A General Algebra and Implementation for Monitoring Event Streams

Alan Demers Johannes Gehrke Mingsheng Hong Mirek Riedewald Walker White

Cornell University
{ademers, johannes, mshong, mirek, wmwhite}@cs.cornell.edu

Abstract

Recently there has been considerable research on Data Stream Management Systems (DSMS) to support analysis of data that arrives rapidly in high-speed streams. Most of these systems have very expressive query languages in order to address a wide range of applications.

In this paper, we take a different approach. Instead of starting with a very powerful data stream query language, we begin with a well-known class of languages — event languages. Through the addition of several simple, but powerful language constructs (namely parameterization and aggregates), we add pieces that extend their expressiveness towards full-fledged languages for processing data streams. Our resulting contributions are a novel algebra for expressing data stream queries, and a corresponding transformation of algebra expressions into finite state automata that can be implemented very efficiently. Our language is simple and natural, and it can express surprisingly powerful data stream queries. We formally introduce the language including a formal mapping of algebra expressions to finite state automata. Furthermore, we show the efficacy of our approach via an initial performance evaluation, including a comparison with the Stanford STREAM System.

1 Introduction

Traditional Database Management Systems are built on the concept of *persistent data sets* that are stored on stable storage and queried and updated repeatedly throughout their lifetime. In many application domains, however, data arrives in high-speed streams and needs to be processed continuously [BBD+02, GGR02]. As an example, consider a system that permits financial analysts to configure custom event notification queries over a stream of stock ticks [tra]. An analyst needs to be notified as soon as one of her queries is satisfied. Other applications include transactions in retail chains, ATM and credit card operations in banks, operating system and Web server log records, and many more. These applications *monitor* data streams [CcC+02]: users register long-running *continuous* queries, which compute their results in *real-time* while the data is streaming by. Stream monitoring applications motivate the following desiderata:

- **Scalability.** The system must be scalable in both the arrival rate of the data stream and the number of queries.
- **Expressiveness.** The query language must be expressive enough to meet application requirements. There is a clear tradeoff between query expressiveness and system performance.
- **Well-Understood Formal Semantics.** The meaning of a query expression should be formally defined. Well-defined semantics is a prerequisite for query-rewriting and multi-query optimization.

Several groups are building Data Stream Management Systems (DSMS) with powerful query languages [AAB+05, MWA+03, CCD+03]. There are two main thrusts in existing work. One line of work extends SQL with constructs for data streams, resulting in very powerful query languages with challenging implementation and query optimization problems. Another direction uses a similarly powerful procedural boxes-and-arrows programming model, where the user is required to explicitly select an efficient query plan by deciding which boxes to use and how to combine them. It is fair to characterize most current work as adapting complex systems to apply to data streams.

In this paper, we take a different approach. Instead of starting with a complex system, we extend a much weaker system originally intended for real-time data. We start with event systems, an area that has been the focus of much research over the last decade. Languages for event systems (*event algebras*) can compose complex events from either basic or complex events arriving on a data stream. However, there are significant shortcomings that prevent current event systems to be used for data stream processing.

1.1 From Event Algebras to Data Streams

Our goal is to design an algebra that supports a large class of data stream monitoring applications and is amenable

to scalable implementation. As a running example, we use an application for analyzing stock data. Assume there is a stream of events whose data fields include $(\texttt{name}, \texttt{price}, \texttt{vol}; \texttt{timestamp})$, indicating that `vol` shares of company with name `name` are sold at price `price`. In addition, there is a `timestamp` attribute, indicating when the sale happened.

Existing Constructs: Filtering and Sequencing. We need basic operators to *filter* events and attributes of interest, like selection and projection in the relational algebra. This is essentially the functionality of pub/sub systems, as illustrated by the following:

Query 1. *Notify me when the price of IBM is above $100.*

This query can be evaluated on each incoming event individually. However, an important feature of monitoring systems is the ability to detect *sequences* of events:

Query 2. *Notify me when the price of IBM is above $100, and the* first *MSFT price afterwards is below $25.*

Recall that Query 1 is memoryless – it only compares attribute values of incoming events to constants. For Query 2, however, the system has to maintain *state*; it has to remember whether an event with a stock price of IBM above $100 has happened.

Extension 1: Parameterization. Stream queries often have constraints on events depending on constants from earlier events in the query. Consider the following:

Query 3. *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 $1.05p$.*

In this example, `name` and `price` are parameters that are not specified in the query. Instead, the second stock event is constrained by the name and price of the first stock quote (it has to be the same company at a price at least 5% higher). The act of referring to values of prior events in a multi-event query is called *parameterization*.

Most previously proposed event processing systems either do not support parameterized predicates at all or attempt to simulate them by post-processing the output of another less selective query [PSB03]. For example, Query 3 can be evaluated by generating all adjacent pairs $\langle p, q \rangle$ of stock quotes, and then filtering them by (1) removing all pairs in which $q.\texttt{name} \neq p.\texttt{name}$, and then (2) removing all pairs for which $q.\texttt{price} \leq 1.05p.\texttt{price}$. Such post-processing might be tractable in an event processing system where the base input stream contained only a limited number of predefined primitive events [CKAK94, ZU99]. However, for steam processing applications, the domain from which events are drawn is potentially unbounded. Intuitively, this post-processing approach is analogous to computing a join by first generating the full cross-product, and then filtering based on the join condition.

More importantly, post-processing does not generalize to queries for which a potentially unbounded number of parameter values must actually be checked. Consider the following:

Query 4. *Notify me when the price of IBM increases monotonically for at least 30 minutes.*

In processing this query, each arriving IBM price must be checked against the previous IBM price. To defer this work to a post-processing phase, the output would have to contain the entire event history for the query. This would result in an output stream with events of unbounded size, a design choice that is undesirable, and (since testing for monotonicity only requires comparing adjacent quotes) unnecessary. Furthermore, parameterized *iteration* (Kleene-*) is an important concern, enabling powerful queries such as the following:

Query 5. *Notify me when the price of any stock increases monotonically for at least 30 minutes.*

Extension 2: Aggregation. Another immensely useful feature for an event processing language is support for *aggregation*. Again, most event processing systems either do not support aggregation at all or handle aggregates "externally" (i.e. pass events to another process for computing the aggregate) [CKAK94]. Not including aggregates as part of the algebra severely limits opportunities for query optimization – sound query rewrite rules must take the presence of an aggregation operator into account.

Query 6 is an example of a data stream aggregate. Average is not part of our stock stream schema, and hence must be computed from the events.

Query 6. *Notify me when the next IBM stock is above its 52-week average.*

Extension 3: Well-Defined Semantics *and* Efficient Implementation. Our final contribution is more subtle. When surveying previous work, we observed an unfortunate dichotomy between theoretical and systems-oriented approaches. Theoretical approaches, based on formal languages and well-defined semantics, generally lack efficient, scalable implementations. On the other hand, systems approaches usually introduce their event algebra only informally. The absence of a precise formal specification limits the ability to do query optimization and query rewrites on these systems. Indeed, previous work has shown that the lack of clean operator semantics can lead to unexpected and undesirable behavior of complex algebra expressions [GA02, ZU99]. Our approach was informed by this dichotomy, and we have taken great care to define a language that can express very powerful queries and has a formal semantics, but can be implemented efficiently.

1.2 Contributions of This Work

In this paper, we develop a novel event stream algebra called CESAR (for Composite Event Stream AlgebRa) that can express all example queries of the previous section. CESAR queries are translated into finite state automata that can be implemented efficiently and that are amendable to multi-query optimization in the style of recent automata-based systems for XML filtering[DAF$^+$03]. This implementation is realized in Cayuga, a system architecture for

processing queries in our event algebra. Cayuga leverages pub/sub techniques to achieve high scalability.

The important contributions of this paper are as follows:

- We introduce CESAR, a novel algebra for stream processing, which supports expressive queries including parameterization and aggregation, and hence is applicable to a large class of applications. We discuss the important design decisions and point out subtle challenges when defining operators for data stream processing. (Section 2)

- We identify a subset of our algebra, called *linear-plus* expressions, which while very expressive, is particularly easy to implement. We define a new automaton model such that every linear-plus expression can be implemented by an automaton that is acyclic except for self-loops. (Section 3)

- We introduce Cayuga, a novel system architecture that enables high-speed processing large sets of queries expressed in our event algebra. Key to our scalability is an interesting use of existing pub/sub techniques for processing part of the queries, aggressive merging and novel indices on automata states and instances, an effective form of multi-query optimization. (Section 4)

- In a thorough performance evaluation, we show the scalability of our system both in terms of complexity and number of queries, and we show interesting tradeoffs between expressiveness of queries and their performance. (Section 5)

The technical details of our approach are discussed in Sections 2, 3, 4, and 5. We discuss related work in Section 6, and conclude in Section 7.

2 CESAR: An Event Processing Algebra

The algebra CESAR consists of a data model for event streams plus operators for producing new streams from existing ones. A stream S is a (infinite) set of tuples $\langle \overline{a}, t_0, t_1 \rangle$, which we call *events*. 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 starting and ending timestamps of the event. For example, an input event in the stock monitoring application could be $\langle \text{IBM}, 90, 15000; 9{:}10; 9{:}10 \rangle$. The timestamps are identical, because a sale is an instantaneous event. An example for an event with duration is $\langle \text{IBM}, 90, 85; 9{:}10; 9{:}15 \rangle$, which could indicate that the price of IBM decreased from \$90 to \$85 between 9:10AM and 9:15AM.

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 to avoid well-known problems involving concatenation of complex events [GA02].

To support the functionality discussed in Section 1.1, our algebra has four unary and three binary operators which add the expressive power of regular expressions to pub/sub expressions, making CESAR strictly more expressive than both pub/sub and regular expressions. In the following, S_i refers to an arbitrary stream of tuples, possibly containing events with non-zero duration and overlapping or simultaneous events.

2.1 Unary Operators

The operators introduced in this section are well known from relational algebra. The first unary operator is the standard **projection** operator $\pi_{\mathbb{X}}$, where $\mathbb{X}$ is a set of attributes. Projection can only affect data values; temporal values are always preserved. The second unary operator is the standard **selection** operator σ_θ, where θ is any selection formula. 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 is one of $=, \leq, <, \geq, >$. We also allow selection formulas to contain predicate DUR relop c where c is a constant, and relop is as above. We write $e \models \theta$ when selection formula θ is true of event e. So for any stream S_i,

$$\sigma_\theta(S_i) \;=\; \{\, e \in S \mid e \models \theta \,\}.$$

One last unary operator is the **renaming** operator ρ_f where f is a function taking one set of attributes to another set of attributes. For example, if S is our stock stream from Section 1.1, and f maps `price` $\mapsto$ `oldprice`, then the stream $\rho_f(S)$ has schema (`name`, `oldprice`, `vol`, `timestamp`). It is important to note that a renaming function applied to a stream simply changes the names of the *data attributes* in the stream schema. The timestamps cannot be renamed.

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 usually index the attributes of an event by the ID of the input stream, making renaming implicit. For example, the `name` attribute of events from stream S_1 will be referred to as S_1.`name`. From here on, will only use an explicit renaming operator when we want a specific certain schema.

2.2 Basic Binary Operators

The unary operators enable filtering of events and attributes, and when selection formula are limited to conjunctions of atomic predicates, are the equivalent of a classical pub/sub system. They support queries over individual events, but no composite events and no parameterization. The added expressive power of our algebra, compared to that of pub/sub systems, lies in the binary operators. All of these operators are motivated by a corresponding operator in regular expressions.

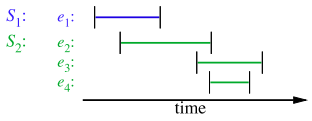


Figure 1: Time intervals of arriving events

The first binary operator is the standard **union** operator $\cup$, where $S_1 \cup S_2$ is defined as $\{ e \mid e \in S_1 \text{ or } e \in S_2 \}$. As in the relational model, we require "union compatible" schemas, achievable by the renaming operator ρ_f.

For the following discussion, we introduce an operator for **concatenation** of events. This operator is not part of CESAR; it will only be used to simplify the definition of CESAR operators. Let $e_0 = \langle \overline{a}, t_0, t_1 \rangle$ and $e_1 = \langle \overline{b}, s_0, s_1 \rangle$ be two *non-overlapping* events ($t_1 < s_0$). We define the concatenated event $e_0 \hat{} e_1 = \langle \overline{a} \cup \overline{b}; t_0, s_1 \rangle$. The values of e_1 overwrite values of e_0 for data attributes that occur in both schemas. For example, let $e_0 = \langle \text{IBM}, 10, 1000; 1, 1 \rangle$ have data schema – that is, not including the timestamps – ($\texttt{name}, \texttt{price}, \texttt{vol}$) and $e_1 = \langle \text{MSFT}, 99; 2, 3 \rangle$ have data schema ($\texttt{company}, \texttt{price}$). Then we obtain $e_0 \hat{} e_1 = \langle \text{IBM}, 99, 1000, \text{MSFT}; 1, 3 \rangle$ with data schema ($\texttt{name}, \texttt{price}, \texttt{vol}, \texttt{company}$). When there is a renaming function f, $e_0 \hat{} _f e_1$ is the same as $e_0 \hat{} e_1$ except that the attributes of e_1 have been renamed by f.

For the remainder of the discussion it is important to understand that, when we refer to $e_0 \hat{} e_1$, it is implicit that the start time of e_1 follows the end time of e_0 (i.e. events can only be concatenated if their time intervals do not overlap).

For streams S_1 and S_2, and selection formula θ, we define the **conditional sequence** $S_1 \,;_\theta S_2$ as

$$\left\{ e_1 \hat{} e_2 \models \theta \;\middle|\; \begin{array}{l} e_1 \in S_1, e_2 \in S_2, \text{ and } \nexists e' \in S_2 \\ \text{such that } e'.\text{END} < e_2.\text{END}, e \hat{} e' \models \theta \end{array} \right\}$$

Intuitively, this operator computes sequences of consecutive events, filtering out those events from S_2 that do not satisfy θ. We will use $;$ as shorthand notation for $;_{\text{TRUE}}$, i.e., when no events are filtered by the operator.

Assume streams S_1 and S_2 are as shown in Figure 1 and we want to compute $S_1 \,; S_2$. The result of this query is a single composite event, which contains the data fields of e_1 and e_4, and whose starting and ending timestamps are the starting time of e_1 and the ending time of e_4, respectively. Event e_2 cannot be combined with e_1, because the time intervals are not disjunct (condition $s_1 < s_0$ in above definition). Event e_3 does not combine with e_1, because e_4 starts after e_1 ends, and it ends before e_3 ends (see definition above). For the same reason, any other event ending after e_4 cannot be combined with e_1 any more. This is exactly the expected semantics of sequential composition, to combine an event from S_1 with and only with the first matching event from S_2.

Two important design decisions are illustrated by the above example. First, two events $e_1 \in S_1$ and $e_2 \in S_2$ can only be combined if e_2 starts after e_1 has ended. Without this requirement, i.e., by removing conditions $s_1 < s_0$ and $s_1 < r_0$ from the above definition, it can be shown that $S_1 ; (S_2 ; S_3)$ is equivalent to $S_2 ; (S_1 ; S_3)$ [GA02]. This would result in confusing and unexpected query semantics. Notice, however, that if such semantics was desired, our algebra could easily be modified to support it.

Second, out of all events from S_2 that start after $e_1 \in S_1$ has ended, the operator selects the one with the earliest *end* timestamp. In the above example, e_3 is not combined with e_1, even though it started before e_4. This choice of operator semantics is natural, because an event *occurs* at its end time and hence the event order is determined by the end timestamp. From an implementation point of view this semantics is desirable as well, because before the end time of an event, the system has no knowledge if this event will ever occur or not.

$S_1 \,;_\theta S_2$ 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. Notice that traditional event processing systems do not have θ as part of the operator. Adding this feature is essential for parameterization, because θ can refer to attributes of both S_1 and S_2. This enables our algebra to express "group-by" operations such as in Query 3, where we group stock quotes by name via $S_1 \,;_\theta S_2$, and θ is $S_1.\texttt{name} = S_2.\texttt{name}$. Any selection formula θ' in $\sigma_{\theta'}(S_i)$ can only refer to attributes in S_i. Without θ as part of the sequencing operator, we would have to resort to post-processing, which is equivalent to computing a join by first generating the cross-product and then filtering based on the join condition afterwards.

2.3 Iteration

The last binary operator is motivated by the Kleene-+ operator. For example, suppose we want to detect an upward trend in a stock price as was shown in Examples 4 and 5. To express such queries, we introduce the **iteration** operator $\mu_{\mathfrak{F},\theta}(S_1, S_2)$ where

- $\mathfrak{F}$ is a unary operation formed from the composition of selection and projection. This operation is used to "trim" the output after each iteration.
- θ is a selection formula. This formula is used to filter elements of S_2 just as it does in $;_\theta$

Additionally, we require the schema of S_2 be a subset of the schema of S_1. This requirement can be dropped by attaching a renaming function to the μ operator itself, but we will ignore this for the purpose of clarity.

Informally, this operator acts as a fixed point operator (hence the use of the symbol μ), applying the operator $;_\theta$ repeatedly until it produces an empty result. However, at each stage, 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 (which is why we require that the schema of S_2 be a subset of that of S_1), but it will be overwritten by each iteration.

Formally, we define this operator as follows. We set $\mu_{\mathfrak{F},\theta}(S_1, S_2) = \bigcup_{n\geq 1} \mathcal{S}^{[n]}$ where

$$\begin{aligned}\mathcal{S}^{[0]} &= \{\, \langle \overline{a}, \overline{b}, t_0, t_1\rangle \models \theta \mid \langle \overline{a}, t_0, t_1\rangle \in S_1, \overline{b} \subseteq \overline{a} \,\}\\ \mathcal{S}^{[n+1]} &= \pi_{\mathbb{S}_1\cup\mathbb{S}_2} \circ \mathfrak{F}\big((\mathcal{S}^{[n]}) \underset{\theta}{;} S_2\big).\end{aligned}$$

where $\mathbb{S}_i$ is the schema of the stream S_i. Here $\circ$ is the standard composition operator, i.e., for two operators ω_1, ω_2, and input x the expression $\omega_1 \circ \omega_2(x)$ is equivalent to $\omega_1(\omega_2(x))$. We will use this notation to improve readability. Furthermore, to avoid notational clutter, we omitted the necessary renaming functions in the above definition. Notice that the values $\overline{b}$ in $\mathcal{S}^{[0]}$ are obtained by projecting and renaming the attributes of $\overline{a}$. Furthermore, the attributes ATT_i of S_2 are renamed to $\text{ATT}_i.\texttt{last}$ after each iteration, right before applying the projection.

With iteration, we can express Query 5 as

$$\sigma_{\theta_3}\left(\mu_{\sigma_{\theta_2},\theta_1}(S_1, S_2)\right) \tag{1}$$

where both S_1 and S_2 refer to the base stream of stock quotes (think of S_1 and S_2 as having the renaming operator applied to the base stream to distinguish the attributes of the same name, such as $S_1.\texttt{name}$ and $S_2.\texttt{name}$), θ_1 is $S_1.\texttt{name} = S_2.\texttt{name}$, θ_2 is $S_2.\texttt{price} > S_2.\texttt{price}.\texttt{last}$, and θ_3 is the duration constraint $\sigma_{\text{DUR}\geq 30\text{ min}}$. We illustrate the functionality of the μ operator for the following example stream $S_1 = S_2 =$

$$\left\{\begin{array}{l}\langle \text{IBM}, 10; 1, 1\rangle, \langle \text{Dell}, 22; 2, 2\rangle, \langle \text{IBM}, 19; 3, 3\rangle,\\ \langle \text{Dell}, 24; 4, 4\rangle, \langle \text{IBM}, 22; 5, 5\rangle, \langle \text{Dell}, 22; 6, 6\rangle\end{array}\right\}$$

Notice that for this example for the sake of readability we simplified the schema of the stream by removing the volume attribute, which is irrelevant for the query.

The initial set $\mathcal{S}^{[0]}$ is computed by duplicating the relevant attributes $S_1.\text{ATT}_i$ and renaming them to $S_2.\text{ATT}_i$. Hence the resulting schema of $\mathcal{S}^{[0]}$ is $(S_1.\texttt{name}, S_1.\texttt{price}, S_2.\texttt{name}, S_2.\texttt{price}; t_0, t_1)$. For the example stream we obtain $\mathcal{S}^{[0]} =$

$$\left\{\begin{array}{l}\langle \text{IBM}, 10, \text{IBM}, 10; 1, 1\rangle, \langle \text{Dell}, 22, \text{Dell}, 22; 2, 2\rangle,\\ \langle \text{IBM}, 19, \text{IBM}, 19; 3, 3\rangle, \langle \text{Dell}, 24, \text{Dell}, 24; 4, 4\rangle,\\ \langle \text{IBM}, 22, \text{IBM}, 22; 5, 5\rangle, \langle \text{Dell}, 22, \text{Dell}, 22; 6, 6\rangle\end{array}\right\}$$

From this set, the first iteration $\mathcal{S}^{[1]}$ will hold all pairs of adjacent quotes of the same stock (θ_1 filters out quotes from other companies, the same way this happens for $;_\theta$), in which the second quote is higher. The latter is enforced by θ_2, which removes all pairs of quotes with non-increasing price. This iteration has the same schema as the previous one. To achieve this, the attributes $S_2.\text{ATT}_i$ of the last iteration are renamed to $S_2.\text{ATT}_i.\texttt{last}$ before concatenation; after concatenation and selection, the attributes $S_2.\text{ATT}_i.\texttt{last}$ are projected out. This gives us $\mathcal{S}^{[1]} =$

$$\pi_{\mathbb{S}_1\cup\mathbb{S}_2} \circ \sigma_{S_2.\texttt{price} > S_2.\texttt{price}.\texttt{last}}(\mathcal{S}^{[0]} \underset{S_1.\texttt{name}=S_2.\texttt{name}}{;} S_2) =$$

$$\left\{\begin{array}{l}\langle \text{IBM}, 10, \text{IBM}, 19; 1, 3\rangle, \langle \text{Dell}, 22, \text{Dell}, 24; 2, 4\rangle,\\ \langle \text{IBM}, 19, \text{IBM}, 22; 3, 5\rangle\end{array}\right\}$$

The next iteration is computed as $\mathcal{S}^{[2]} =$

$$\pi_{\mathbb{S}_1\cup\mathbb{S}_2} \circ \sigma_{S_2.\texttt{price} > S_2.\texttt{price}.\texttt{last}}(\mathcal{S}^{[1]} \underset{S_1.\texttt{name}=S_2.\texttt{name}}{;} S_2) =$$

$$\{\langle \text{IBM}, 10, \text{IBM}, 22; 1, 5\rangle\}$$

After this point, $\mathcal{S}^{[n]}$ is empty for all $n > 2$. The union $\bigcup_{n\geq 1} \mathcal{S}^{[n]}$ is the result of the μ operator. The final query result is obtained by selecting all those tuples (composite events), which satisfy θ_3, i.e., have a duration of at least 30 minutes.

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 the above example, it was used to make sure that no Dell stock 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. Without this feature, an iteration could produce a large number of irrelevant results, which in turn generates even more irrelevant results in the following iterations, just to be removed by external post-processing.

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 queries. For instance, if S_1 is generated by some complex algebra expression, the μ operator can constrain its iterations by any of the previously generated bindings. Example 5 illustrates a simple usage of this feature by constraining the sequence to consist of a single stock only (θ is $S_1.\texttt{name} = S_2.\texttt{name}$).

2.4 Aggregates

Aggregates fit naturally into our algebra, where aggregation occurs over a sequence of events. Like in SQL, we need to create new attributes where the aggregate values are stored. More formally, an *attribute introduction function* g is a map that takes an attribute x and produces τ_x, an arithmetic combination of attributes and constants. For any event e, we let $g[\![e]\!]$ be e with extra values added according to the rules of g. For example, suppose

$$e = \langle \text{IBM}, 10, \text{IBM}, 19; 1, 3\rangle \in S_1 \underset{\theta}{;} S_2$$

where S_1 and S_2 refer to the stock stream, and θ is the formula $S_1.\texttt{name} = S_2.\texttt{name}$. We let g be the map $\text{AVG} \mapsto \frac{S_1.\texttt{price}+S_2.\texttt{price}}{2}$. Then, $g[\![e]\!] = \langle \text{IBM}, 10, \text{IBM}, 19, 14.5; 1, 3\rangle$ for the new data schema $(S_1.\texttt{name}, S_1.\texttt{price}, S_2.\texttt{name}, S_2.\texttt{price}, \text{AVG})$.

Given an expression $\mathcal{E}$ and introduction function g, the *attribute introduction operator* α_g is defined as

$$\alpha_g(\mathcal{E}) = \{\, g[\![e]\!] \mid e \in \mathcal{E} \,\}$$

Together with μ, we get a natural aggregate. Consider an expression 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 Query 6. If we let $S_1, S_2 = S$ be our stream of stock quotes, this is expressed

$$\mathcal{E} = \sigma_{\text{DUR}=52\text{ weeks}}\left(\mu_{\alpha_{g_2},\text{TRUE}}(\alpha_{g_1}\circ\sigma_\theta(S_1),\sigma_\theta(S_2))\right) \tag{2}$$

where θ is `name` $=$ IBM, g_1 is defined as AVG $\mapsto$ `price`, COUNT $\mapsto 1$, and g_2 is defined as AVG $\mapsto \frac{\text{COUNT.}\texttt{last}\times\text{AVG.}\texttt{last}+\texttt{price}}{\text{COUNT.}\texttt{last}+1}$, COUNT $\mapsto$ COUNT.`last` $+ 1$. Notice that we use `last` feature of μ to compute our aggregate recursively.

Note the average is now a value attached to an attribute and can be used in more complex queries. For example, we express Query 6 as $\sigma_{\theta_2}(\mathcal{E}\,;_{\theta_1} S_3)$ where $S_3 = S$ is our stream of stock quotes, $\mathcal{E}$ is as in (2), θ_1 is S_3.`name` $=$ IBM, and θ_2 is S_3.`price` $>$ AVG.

3 Processing Expressions

Our algebra defines the output of an expression, but does not indicate the most efficient way to compute it. Given the algebra's similarity to regular expressions, finite automata would appear to be a natural implementation choice. However, standard finite automata are not sufficient for several reasons. First, our algebra expressions generate output, hence the automata must be transducers rather than recognizers [HMU00]. Second, because of parameterization, we need a mechanism for keeping track of parameter values. Third, attributes of events can have infinite domains, e.g., text attributes. Thus, the input alphabet of the automaton, which is the set of all possible events, can be infinite as well.

3.1 Automaton Example

At a high level, an automaton that implements an algebra expression (the query) works as follows. Based on the events seen so far in the stream, the automaton maintains all "partially matched sequences". Recall that a query describes a complex sequence of events, which itself is also an event—a *composite* event. The following query illustrates this point:

Query 7. *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 (`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.*

After the first large trade of a stock, the automaton will be looking for a monotonically decreasing sequence, then for a sudden up-move in price. At any given moment in time, there might be several event sequences that satisfy some prefix of the query pattern.

For presentation purposes, we use a slightly simplified version of our actual automata for this example. Its purpose is to provide an intuitive understanding of the approach, before introducing the formal definition.

Our event automata are similar to nondeterministic finite automata [HMU00]. Whenever an automaton is in a state where it can traverse more than one edge for an incoming event, it nondeterministically explores all these branches. If it cannot traverse any edge, the corresponding branch "dies". This is equivalent to having multiple *active instances* of the automaton explore the different branches, each branch corresponding to a prefix of the query sequence. Since our automata have to keep track of parameters, an instance of the automaton has to store event attributes and their values.

Let S be the input stream of stock quotes, and assume for the purpose of this example that no two quotes in the stream have the same timestamp. The algebraic expression for Query 7 is then $\sigma_{\theta_5}(\sigma_{\theta_4}(\mu_{\sigma_{\theta_3},\theta_2}(S_1,S_2))\,;_{\theta_2} S_3)$. The S_i are shorthand notation for appropriately renamed and projected versions of S:

$$\begin{aligned} S_1 &\equiv \rho_{f_1}\circ\pi_{\texttt{name},\texttt{price}}\circ\sigma_{\theta_1}(S)\\ S_2 &\equiv \rho_{f_2}\circ\pi_{\texttt{name},\texttt{price}}(S)\\ S_3 &\equiv \rho_{f_3}\circ\pi_{\texttt{name},\texttt{price}}(S). \end{aligned}$$

The corresponding predicates and renaming functions are

$$\begin{aligned} \theta_1 &\equiv \texttt{vol} > 10,000\\ \theta_2 &\equiv \texttt{company} = \texttt{company.last}\\ \theta_3 &\equiv \theta_2 \wedge \texttt{minP} < \texttt{minP.last}\\ \theta_4 &\equiv \theta_3 \wedge \text{DUR} \geq 10\,\text{min}\\ \theta_5 &\equiv \theta_2 \wedge \texttt{price} > 1.05\,\texttt{minP}\\ f_1 &\equiv (\texttt{name},\texttt{price}) \mapsto (\texttt{company},\texttt{maxP})\\ f_2 &\equiv (\texttt{name},\texttt{price}) \mapsto (\texttt{company},\texttt{minP})\\ f_3 &\equiv (\texttt{name},\texttt{price}) \mapsto (\texttt{company},\texttt{finalP}) \end{aligned}$$

The explicit use of renaming is necessary for this example to make the schemas of the intermediate results clear.

The algebra expression is interpreted as follows. S_1 is obtained from S by selecting only large volume trades (θ_1), then projecting out the volume attribute and changing the attribute names (f_1). Hence S_1 contains only large trades and has data schema (`company`, `maxP`). The μ operator searches for a monotonically decreasing sequence (θ_3) for the same stock, ignoring quotes from other companies (θ_2). During each iteration μ compares the current lowest price (temporarily renamed to `minP.last`) to the price of the incoming event, renamed by f_2 to `minP`. If a new minimum price is found, the concatenation overwrites the previously lowest price by the new one, otherwise the monotonic sequence has ended. The μ operator produces output events as soon as the duration constraint in θ_4 is satisfied. Finally, the $;_{\theta_2}$ operator finds the next quote for the same

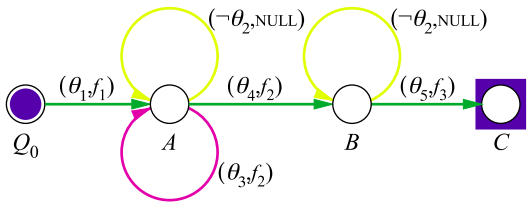


Figure 2: Automaton for query 7

company (θ_2). If the price of that quote satisfies θ_5, the query produces an output event. As before, the renaming operator f_3 ensures, that the final price is added to the result as attribute `finalP`.

The corresponding automaton is shown in Figure 2. Table 1 presents a sample input event stream, and illustrates how the automaton processes these events. The input events are displayed in the first column. For an incoming event, the state of the automaton after processing it is indicated by the active automaton instances in the same row. The table headers show the schema of the instances at a given automaton state. For readability, the timestamp attributes, which cannot be projected out or renamed anyway, are not shown in the schema.

Initially there is no active automaton instance, but the start state is always active by default. When e_1 arrives, the automaton checks if it satisfies θ_1, the predicate on the edge emanating from the start state. This is the case, therefore it applies the projection and renaming function f_1 to the attributes of e_1 and advances the resulting instance to state A. Notice that currently the instance has no `minP` attribute, which is indicated by the NULL value in the corresponding position in I_1.

The next event e_2 does not satisfy θ_1, hence the start state does not create a new instance. For I_1 at state A, the automaton performs the following computation to determine if I_1 can traverse any outgoing edge. First, it applies the projection and renaming to e_2. Then it checks if the composite event, obtained by concatenating I_1 and the projected and renamed e_2, satisfies the predicate of any of the edges emanating from state A. Recall that the μ operator performs a temporary renaming of the attributes of the second operand from ATT_i to ATT_i.`last` during an iteration. Hence, this concatenated event is $\langle \text{IBM}, 90, \text{NULL}, \text{IBM}, 85; 9{:}10; 9{:}15\rangle$ with data schema (`company.last`, `maxP.last`, `minP.last`, `company`, `minP`). This event only satisfies θ_3 on the the rebind edge (self-loop below A). This edge therefore is traversed and instance I_1 is updated by concatenating I_1 and the projected and renamed e_2 (this time without the temporary renaming of I_1). Hence the new values of I_1 are obtained by concatenating the old tuple with $\langle \text{IBM}, 85; 9{:}15; 9{:}15\rangle$ for data schema (`company`, `minP`). The result is shown in Table 1. Notice how the previous NULL value for `minP` is now replaced by e_2's value.

Event e_3 matches θ_1, therefore a new instance I_2 is created at state A. For I_1, the concatenation of I_1 and e_3 only satisfies the predicate of the filter edge (top loop of state A), because `company.last` = IBM and `company` = Dell. Filter edges have special semantics—traversing them never updates the bindings of an instance. This is indicated in Figure 2 by the NULL value for the renaming function.

The arrival of e_4 illustrates the non-determinism of the μ operator. e_4 is filtered for I_2 (the Dell pattern). However, for I_1 both θ_3 and θ_4 are satisfied (duration condition is now true). Hence I_1 non-deterministically traverses both the forward edge from A to B and the rebind edge of state A. This is implemented by cloning I_1 so that there is an instance to traverse each satisfied edge. In the example, clone I_3 traverses the forward edge, concatenating the instance with the renamed and projected e_4.

Events e_5 and e_6 are processed similarly. For e_5 each of the instances traverses the corresponding filter edge. The interesting aspect of e_6 is its affect on instance I_1. I_1 concatenated with e_6 does not satisfy the predicate on any outgoing edge of state A, therefore the instance is deleted. Notice how the nondeterminism ensures correct discovery of the IBM pattern for instance I_3 (events e_1, e_4, e_6 match it), but prevents any later arriving IBM event from generating another matching pattern starting with e_1, because I_1 has failed.

At this point we need to point out two subtleties that need to be addressed by our automata. First, notice that the example automaton cannot properly handle concurrent events. For instance, let there be another event e_6' : $\langle \text{IBM}, 80, 8000; 9{:}24; 9{:}24\rangle$ at the same time as e_6. Even though this event fails θ_5, according to algebra semantics the automaton should still produce the output result with e_6. This suggests that forward edges (and rebind edges as well) are traversed if there *exists* a satisfying event. On the other hand, the same is not true for filter edges. The arrival of an additional event e_3' : $\langle \text{IBM}, 99, 8000; 9{:}17; 9{:}17\rangle$ at the same time as e_3 would cause I_1 to be deleted based on algebra semantics. Hence a filter edge should only be traversed if *all* simultaneously arriving events satisfy the filter predicate.

Second, there is a subtlety related to duration constraints illustrated by the following simple sequence query $\sigma_{\theta_1}(S;\sigma_{\theta_2}(S))$, where the θ_i both are predicates on duration. In this query, θ_2 refers to the duration of input events, while θ_1 refers to the duration of the composite events generated by the sequencing operator. In the automaton in Figure 2, the predicate on an edge refers to the concatenated event (active instance concatenated with input event), hence that automaton cannot support a duration predicate like θ_2 on input events. Our formal automaton model addresses these issues.

3.2 The Formal Automata Model

Now that we have seen both a high-level example and an overview of some of the more subtle issues, we are ready to present a formal description of our translation from expressions to automata. Rather than directly translating arbitrary algebra expressions into automata, we will start with a simpler, but still powerful subset, which we refer to as *linear-*

Input event (name, price, vol)	Instances at state A (company, maxP, minP)	Instances at state B (company, maxP, minP)	Instances at state C (company, maxP, minP, finalP)
$e_1 : \langle$IBM, 90, 15000; 9:10; 9:10$\rangle$	$I_1 = \langle$IBM, 90, NULL; 9:10; 9:10$\rangle$		
$e_2 : \langle$IBM, 85, 7000; 9:15; 9:15$\rangle$	$I_1 = \langle$IBM, 90, 85; 9:10; 9:15$\rangle$		
$e_3 : \langle$Dell, 40, 11000; 9:17; 9:17$\rangle$	$I_1 = \langle$IBM, 90, 85; 9:10; 9:15$\rangle$ $I_2 = \langle$Dell, 40, NULL; 9:17; 9:17$\rangle$		
$e_4 : \langle$IBM, 81, 8000; 9:21; 9:21$\rangle$	$I_1 = \langle$IBM, 90, 81; 9:10; 9:21$\rangle$ $I_2 = \langle$Dell, 40, NULL; 9:17; 9:17$\rangle$	$I_3 = \langle$IBM, 90, 81; 9:10; 9:21$\rangle$	
$e_5 : \langle$MSFT, 25, 6000; 9:23; 9:23$\rangle$	$I_1 = \langle$IBM, 90, 81; 9:10; 9:21$\rangle$ $I_2 = \langle$Dell, 40, NULL; 9:17; 9:17$\rangle$	$I_3 = \langle$IBM, 90, 81; 9:10; 9:21$\rangle$	
$e_6 : \langle$IBM, 91, 9000; 9:24; 9:24$\rangle$	$I_2 = \langle$Dell, 40, NULL; 9:17; 9:17$\rangle$		$I_3 = \langle$IBM, 90, 81, 91; 9:10; 9:24$\rangle$

Table 1: Example computation

plus expressions. The name "linear-plus" is inspired by the linear structure of the corresponding automata, which are acyclic with the addition of self-loops. In Section 3.5, we will show how to generalize the approach to handle arbitrary algebra expressions.

Definition 1. *We define a* linear-plus expression *as follows.*

- *Any base stream S_i is linear-plus.*
- *If $\mathcal{E}$ is linear-plus and $\mathfrak{F}$ is a unary operator formed from selection, projection, renaming, and aggregation, then $\mathfrak{F}(\mathcal{E})$ is linear-plus.*
- *If $\mathcal{E}$ is linear-plus and $\mathfrak{F}$ is a unary operator, then $\mathcal{E}_1 ;_\theta \mathfrak{F}(S_i)$ is linear-plus.*
- *If $\mathcal{E}$ is linear-plus and $\mathfrak{F}_i$ are unary operators, then $\mu_{\mathfrak{F}_1,\theta}(\mathcal{E}, \mathfrak{F}_2(S_i))$ is linear-plus.*
- *If $\mathcal{E}_1$ and $\mathcal{E}_2$ are linear-plus, then so is $\mathcal{E}_1 \cup \mathcal{E}_2$.*

To preserve the intuition from Section 3.1, our automata presentation will be graphical instead of algebraic. In addition, without loss of generality, we will assume that there is only one input stream S; the case of multiple input streams can be handled by distinguishing them via an attribute `streamid`.

For the sake of readability, our initial automata will not be capable of implementing expressions with projection or aggregation (i.e. selection and renaming are the only unary operators allowed). In Section 3.4, we will see how to modify the automata for the additional operators. With this in mind, a *simplified event automaton* is a directed multigraph $\mathcal{A}$ with the following properties:

- Vertices are are marked as initial, final, both, or neither.
- Every edge is marked as having either $\exists$-type or $\forall$-type.
- $\exists$-type edges are labeled $(\exists, \theta_1, \theta_2, f)$ where
  - θ_1 is a selection formula referencing only attributes in S. θ_1 filters input events.
  - θ_2 is any selection formula. θ_2 filters the composite event obtained by concatenating an instance at the edge's source vertex with the input event (see discussion below).
  - f is an attribute renaming function taking the attributes in S to another set of attributes $\mathbb{X}$. f prevents naming conflicts as we generate output.
- $\forall$-type edges are labeled $(\forall, \theta_1, \theta_2)$ where the θ_i are as for the $\exists$-type edges. $\forall$-type edges generate no output, and have no associated attribute renaming function.

An *instance* of an event automaton $\mathcal{A}$ is an event e together with a state $q \in \mathcal{A}$. An instance represents the current state of the automaton together with the non-null contents of the buffer. We will not actually define the buffer of an automaton; its existence will be implicit in our definition of an automaton computation. The output of an automaton will be the set of all events e where, for some final state q_F, (e, q_F) is instance of $\mathcal{A}$ generated by the stream S.

To generate instances from the stream S, we first need to break S up into finite strings. The strings for our automata will be *intervals* of events. For any two time units $t_0 \leq t_1$, the interval $[t_0, t_1]$ is the set of all events e with $t_0 \leq \text{DETECT}(e) \leq t_1$. As time is discrete and there are only finitely many simultaneous events at any time, this is a finite set of events. We order the events by detection time to get our string.

Unfortunately, as events can have simultaneous detection time, this set is not necessarily totally ordered. Instead of a string, we get a *multistring*. That is, each position contains a (nonempty) set of events instead of just a single event. For example, consider the following interval of length 4:

$$\begin{array}{ccccc} & & e_{1,0} & e_{2,0} & \\ e_{0,0} & & & e_{2,1} & e_{3,0} \\ & & e_{1,1} & e_{2,2} & \end{array}$$

Position 0 has a single event $e_{0,0}$, while position 1 has two events $e_{1,i}$ where $e_{1,0}.\text{END} = e_{1,1}).\text{END} > e_{0,0}.\text{END}$. Note that despite the use of double indices in positions 1 and 2, simultaneous events are not ordered.

To define the computation of an automaton on an interval, recall from Section 2.2 that $e_0 \hat{} e_1$ is the concatenation of two events, and that $e_0 \hat{}_f e_1$ is the concatenation of two events with the attributes in e_1 renamed via the function f. To make the construction simpler, we also introduce the empty event ϵ, for which $\epsilon \hat{} e = e \hat{} \epsilon = e$ for any event e. Suppose now that we have an automaton $\mathcal{A}$ and an interval $[t_0, t_1]$ of length n. For each $i \leq n$, we define the set

of valid instances for $\mathcal{A}$ at position i of $[t_0, t_1]$. The set of valid instances at position 0 is $\{ (e, q_S) \mid q_S \text{ a start state} \}$. Algorithm 1 computes the set I_i of valid instances at position i for $i > 0$, given the set of events E_{i-1} at position $i-1$.

Algorithm 1 Generating Instance Sets

Require: I_{i-1} is defined
1: $I_i = \emptyset$
2: **for all** instances $(e_0, q) \in I_{i-1}$ **do**
3: **for all** edges $(\theta_1, \theta_2\texttt{type}, f)$ outgoing from q **do**
4: let q' be the destination state of this edge
5: **if** $\texttt{type} = \forall$ **then**
6: **for all** events $e \in E_{i-1}$ **do**
7: test that either $e.\text{START} \leq e_0.\text{END}$ or $e \not\models \theta_1$ or $e_0 \hat{} e \not\models \theta_2$
8: **if** event $e \in E_{i-1}$ passes the test **then**
9: $I_i = I_i \cup \{ e_0 \}$
10: **else**
11: **for all** events $e \in E_{i-1}$ **do**
12: **if** $e.\text{START} > e_0.\text{END}$ and $e \models \theta_1$ and $e_0 \hat{} e \models \theta_2$ **then**
13: $I_i = I_i \cup \{ e_0 \hat{} _f e \}$
14: **return** I_i

From this algorithm we see that $\forall$ edges check all of the simultaneous events against a single instance, but do not alter the instance event; $\exists$ edges on the other hand, spawn a new instance for each satisfying event, thus recording the data from that event. This solves the simultaneity issues from Section 3.1. Algorithm 1 may appear to be expensive because it has several nested loops. However, in practice the algorithm is handled via multiple subscriptions to a pub/sub system, and so the two external loops can be handled fairly efficiently.

Given Algorithm 1, the output of $\mathcal{A}$ on an interval $[t_0, t_1]$ of length n is the set of all events e such that $(e, q_F) \in I_n$ for some final state q_F. Then the *output of* $\mathcal{A}$ is the set of events output by $\mathcal{A}$ on any interval of S. We are now ready to state our primary theorem.

Theorem 1. *Let $\mathcal{E}$ be any linear-plus expression without projection or aggregation. There is an simplified event automaton such that $\mathcal{E}$ is the output of $\mathcal{A}$.*

3.3 Proof of Theorem 1

Our proof proceeds by induction on the definition of a linear-plus expression. Throughout this proof, we will make use of the following lemma, which gives us a convenient order to perform renaming and selection.

Lemma 2. *Let $\mathfrak{F}$ be any unary operation formed from selection and renaming, including multiple instances of each operator in any order. Then $\mathfrak{F}$ can be rewritten as $\mathfrak{F} = \rho_f \circ \sigma_\theta$.*

For the base case, consider the expression $\rho_f \circ \sigma_\theta(S_i)$. We implement this expression as a two-state event automaton with a single edge labeled $(\theta, \text{TRUE}, \exists, f)$ (see Figure 3). This automaton will be referred to as the *base* automaton. That this automaton is correct should be clear from Algorithm 1.

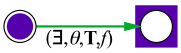


Figure 3: Automaton for $\rho_f \circ \sigma_\theta(S_i)$

Next we consider the case $\rho_{f_2} \circ \sigma_{\theta_3}(\mathcal{E}_1 \,;_{\theta_2} \mathcal{E}_2)$. As this expression is linear-plus, $\mathcal{E}_2 = \rho_{f_1} \circ \sigma_{\theta_1}(S_i)$. Our automaton combines the two automata $\mathcal{A}_1$, $\mathcal{A}_2$ for $\mathcal{E}_1$, $\mathcal{E}_2$, respectively, as shown in Figure 4. We identify the start state of $\mathcal{A}_2$ with the terminal state of $\mathcal{A}_1$; we call this state q_1. We add a loop edge $(\theta_1, \theta_2, \forall, f_1)$; this edge will remove events that do not qualify as successor. We call this type of self-loop edge on which the predicate correspond to the selection formula θ in $;_\theta$ a *filter* edge. In comparison, an edge that goes from one node to another is called a *forward* edge. We always draw a filter edge on top of a node.

We let q_2, the final state of $\mathcal{A}_2$, be the final state of this composite automaton. Finally, the forward edge from q_1, to q_2, we replace the second formula TRUE from the base automaton with $\theta_2 \wedge \theta_3$, and compose the final renaming function f_2 with f_1.

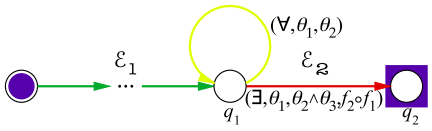


Figure 4: Automaton for $\rho_{f_2} \circ \sigma_{\theta_3}(\mathcal{E}_1 \,;_{\theta_2} \mathcal{E}_2$

To see that the output of this automaton is exactly $\mathcal{E}$, take any event e output by $\mathcal{A}$. Then there is some interval $[t_0, t_1]$ of length n such that (e, q_2) is an instance at position n of this automaton. By definition, there are events e_1, e_2 such that $e = e_1 \hat{} _f e_2$ and (e_1, q_1) is an instance at some position $k < n$. Let k be the least such and let $t' = e_1.\text{END}$. Then $[t_0, t']$ must be an interval of length k such that e_1 is an element of the set in the final position. As k is least, this instance is not produced by traversing the loop edge of q_1. So, e_1 is output by $\mathcal{A}_1$ on $[t_0, t']$, and thus $e_1 \in \mathcal{E}_1$.

The loop edge of q_1 does not add any new data values as it produces new instances; it only forwards the instance to the next stage. The only edge that can possibly add new data values is the forward edge from q_1 to q_2. Hence, $e_2 \in S_i$. By definition, $e \models \theta_1$, and so $e_2 \in \mathcal{E}_2$ (this is the step where we need two formulas on each edge, and not just one). Furthermore, $e_1 \hat{} e_2 \models \theta_2 \wedge \theta_3$, and by definition of concatenation, $e_2.\text{START} > e_1.\text{END}$. Finally, the filter edge guarantees that for any e' with $e'.\text{END} < e_2.\text{END}$, either $e'.\text{START} \leq e_1.\text{END}$, $e' \not\models \theta_1$, or $e_1 \hat{} e' \not\models \theta_2$. Hence $e \in \mathcal{E}$, and so we have shown the output of $\mathcal{A}$ is contains in $\mathcal{E}$. Running this argument in reverse gives the equivalence

of $\mathcal{E}$ and $\mathcal{A}$.

The construction for $\rho_{f_3} \circ \sigma_{\theta_4}(\mu_{\rho_{f_2} \circ \sigma_{\theta_3}, \theta_2}(\mathcal{E}_1, \mathcal{E}_2))$ is shown in Figure 5. Again, we are given that $\mathcal{E}_2 = \rho_{f_1} \circ \sigma_{\theta_1}(S_i)$. We exploit the fact that $\mathcal{E}_2$ is a unary operator applied to a base stream S_i, and thus can be recognized by a single edge. We implement the fixed-point implicit in the μ operator by a loop for this edge. We call this type of self-loop edge on which the predicate corresponds to selection formula θ_3 in the above expression a *rebind* edge, since it controls whether an event can "rebind" to the automaton instance by modifying some of its existing values (as opposed to concatenating new ones). We usually draw the rebind edge below a node. Note that the renaming function f_1 for the rebind edge is the same as for the forward edge; this automaton makes use of the fact that the concatenation $e_0 \hat{}_f e_1$ overwrites old values with attributes in $\text{range}(f)$.

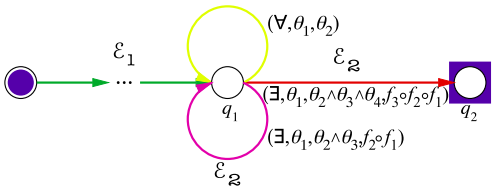


Figure 5: Automaton for $\rho_{f_3} \circ \sigma_{\theta_4}(\mu_{\mathcal{F},\theta_2}(\mathcal{E}_1, \mathcal{E}_2))$

Two more details are needed for the automaton to be correct. First of all, the formulas θ_2 and θ_3 make reference to attributes of the form $\text{ATT}_i.\texttt{last}$. As the composite renaming function $f_2 \circ f_1$ appears on both the rebind and filter edge, we need to rename all such attributes in θ_2 and θ_3 according to this function. In addition, these two predicates are attached to the rebind edge. However, the first instance (e, q_1) to reach this state will have no values for the attributes in $\text{range}(f_2 \circ f_1)$, and hence the rebind edge will always evaluate to FALSE. To solve this, we rewrite the formula $\theta_2 \wedge \theta_3$ as a disjunction that either uses the attributes in $\text{range}(f_2 \circ f_1)$, or if those values are NULL, use the corresponding attributes from $\mathcal{E}_1$. The proof that $\mathcal{A}$ is equivalent to $\mathcal{E}$ should now be clear, following the same steps from the construction for $;_\theta$.

Finally, the construction for $\rho_f \circ \sigma_\theta(\mathcal{E}_1 \cup \mathcal{E}_2)$ is given in Figure 6. Again we start with the event automata for $\mathcal{A}_i$ for each expression $\mathcal{E}_i$, but we combine them differently this time. We identify the start states with each other, and similarly identify the terminal states. In addition, we add θ to the second formula of each edge entering the final state, and compose its renaming function with f. This completes our construction.

3.4 Adding Projection and Aggregation

In order to complete the proof of Theorem 1, we need to extend the definition of our automata to include linear-plus expressions with projections and aggregates. Both of cases can be handled by the following theorem, which is an extension of Lemma 2.

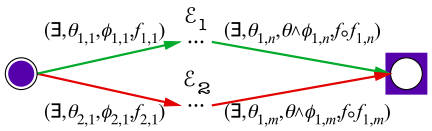


Figure 6: Automaton for $\rho_f \circ \sigma_\theta(\mathcal{E}_1 \cup \mathcal{E}_2)$

Theorem 3. *Let $\mathfrak{F}$ be any unary operation formed from selection, projection, renaming, and aggregate, including multiple instances of each operator in any order. Then $\mathfrak{F}$ can be rewritten as $\mathfrak{F} = \pi_{\mathbb{X}} \circ \alpha_g \circ \rho_f \circ \sigma_\theta$.*

This theorem allows us to delay projection and aggregation until after the selection step for each edge of the automata. All we have to do is modify line 13 of Algorithm 1 to introduce new attributes, and to project out old ones. This is accomplished by two modifications to our $\exists$-type edges. First, our $\exists$-type edges are now labeled $(\exists, \theta_1, \theta_2, \mathbb{X}, f)$, where $\mathbb{X}$ is the set of attributes to preserve at this state. Any value in the instance added at line 13 whose attribute is not in $\mathbb{X}_i$ is removed (i.e. the buffer location is set to NULL).

To implement aggregation, we must modify the attribute renaming function f on the $\exists$-type edges. Note that if f is an aggregate renaming function, then f^{-1} is an attribute *introduction* function, albeit a trivial one that copies the value from one attribute to another. So instead of labeling the edge with a renaming function, we label it with an introduction function that instructs how to compute the value for each buffer location at that stage of the computation. The introduction function labeling that edge is a composition of f^{-1}, where ρ_f is the renaming operator for this edge, and g, where α_g is the aggregate for this edge.

We refer to the automata with the expanded $\exists$-type edges as event automata. Combining Theorem 3 with the proof of Theorem 1, we get the following.

Theorem 4. *Let $\mathcal{E}$ be any linear-plus expression. There is an event automaton such that $\mathcal{E}$ is the output of $\mathcal{A}$. Furthermore, the number of states of this automaton is linear in the size of $\mathcal{E}$.*

3.5 General Algebra Expressions

Since we limited ourselves to linear-plus expressions in Theorem 1, our automata are all simple. As the operators in our algebra are similar regular expressions, one might think that it is possible to construct more complex automata to implement general expressions. However, as we shall demonstrate in the following example, this is not necessarily true.

Let $S_1, S_2, S_3 = S$ be our stream of stock prices, and consider the expression $\mathcal{E} = S_1 ;_\theta (S_2 ; S_3)$, where θ is $S_3.\texttt{price} > S_1.\texttt{price}$. $\mathcal{E}$ is one of the simplest examples of an expression that is not linear-plus. However, it is the composition of two linear-plus expressions: $S_1 ;_\theta S_4$ and $S_2 ;_{\text{TRUE}} S_3$. The automaton $\mathcal{A}$ for $S_2 ; S_3$ has a distinct start and terminal state, so naively we should be able

to replace the S_4-edges in the automaton for $S_1 ;_\theta S_4$ with $\mathcal{A}$. This produces the automaton in Figure 7 (the filter edge for the two copies of $\mathcal{A}$ has been removed as it is never traversed).

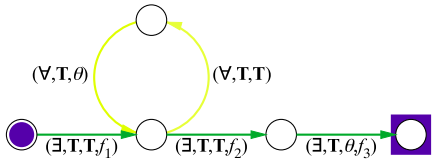


Figure 7: Naive Automaton for $S_1 ;_\theta (S_2 ; S_3)$

However, the automaton in Figure 7 is not correct. The filter "edge" is now a loop consisting of two edges, the first of which is never satisfied (as it is ∀-type, an event must *not* satisfy TRUE to traverse this edge). This is a minor problem which we can solve by changing the first half of the filter to (∀, FALSE, FALSE); this change delays all selection to the end of the filter loop.

Unfortunately, this fix causes another problem. Consider the stream $S =$

$$\left\{ \begin{array}{l} \langle \text{IBM}, 10; 1, 1\rangle, \langle \text{Dell}, 22; 2, 2\rangle, \langle \text{IBM}, 9; 3, 3\rangle, \\ \langle \text{Dell}, 24; 4, 4\rangle, \langle \text{IBM}, 11; 5, 5\rangle \end{array} \right\} \quad (3)$$

The modified automaton in Figure 7 will not output $\langle \text{IBM}, 10, \text{IBM}, 9, \text{Dell}, 24; 1, 4\rangle$ even though it is part of the expression $\mathcal{E}$. The event $\langle \text{Dell}, 22; 2, 2\rangle$ is at fault here. If this event traverses the forward edge from the second state then then this instance of this machine will be eliminated a the next event. On the other hand, if this event traverses the first half of the filter, there is no ∃-type edge to record the event $\langle \text{IBM}, 9; 3, 3\rangle$.

A correct automata implementation must be able to handle the overlap of events in $S_2 ; S_3$. This requires a forward ∃-type edge from the intermediate state of the filter loop. For example, the automaton in Figure 8 processes the stream in (3) correctly.

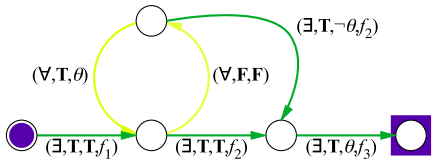


Figure 8: Modified Solution for $S_1 ;_\theta (S_2 ; S_3)$

However, this automaton does not handle simultaneous events properly. Consider the stream $S' =$

$$\left\{ \begin{array}{l} \langle \text{IBM}, 10; 1, 1\rangle, \langle \text{Dell}, 22; 2, 2\rangle, \langle \text{IBM}, 9; 3, 3\rangle, \\ \langle \text{Dell}, 23; 3, 3\rangle, \langle \text{Dell}, 24; 4, 4\rangle, \langle \text{IBM}, 11; 5, 5\rangle \end{array} \right\}$$

For this stream, the automaton in Figure 8 will output $\langle \text{IBM}, 10, \text{Dell}, 23, \text{Dell}, 24; 1, 4\rangle$, which is not correct. We need the ∃-type edge from the filter loop to act as both a ∀-type edge, which processes all simultaneous events together, and a ∃-type edge, which records them separately. Hence a direct construction of this expression appears to require a fundamental change in our automata model.

Instead of changing our automata model, we choose to implement general expressions is through *resubscription*. In resubscription, an automaton is allowed subscribe to the output of another automaton instead of just the base stream. To implement a general expression with resubscription, we break it up into a sequence of linear-plus expressions, like we did with the example in Section 3.5.

As we went through the trouble of ensuring that our automata could deal with simultaneous events of nontrivial duration, there is no problem with treating the output of other automata as data streams. The only issue is how to specify which stream the automaton should use. To do this, we extend our data model to allow for multiple streams, and assign an index for each base stream and for each automaton. Then, to each edge of an automaton, we add the index of the stream to use for that edge. The proof of Theorem 1 is the same as before.

The only issue to worry about is that we cannot have any circular references. We cannot have two automata $\mathcal{A}_1$ and $\mathcal{A}_2$ that subscribe to each other, as all automata are evaluated simultaneously. Fortunately, if we only allow the algebraic expressions to refer to base streams, this can never happen.

Resubscription has the benefit that it can implement all possible algebraic expressions. However, resubscription is more complicated than the single automaton implementation of Section 3. An interesting method of optimization would be to use rewrite rules to convert a general expression to a minimal sequence of linear-plus expressions, and then use resubscription to combine the expressions together. This is an area of future work.

4 Cayuga: The Implementation

For each incoming event, the system must determine which automaton instances need to be modified. At any time, the *total* number of active instances can be very large, but typically the number of instances *affected* by an event is orders of magnitude lower. In the stock monitoring application, for example, a query that matches a sequence of IBM prices can ignore events for any other company. Rather than sequentially testing each instance, as suggested by Algorithm 1, we can use indexes that efficiently identify the instances that are affected by the incoming event.

Note that an instance is *unaffected* by an input event if and only if that event makes the instance traverse its *filter* edge. Traversing a forward or rebind edge modifies bindings, affecting the instance; and if no edge can be traversed, the instance is affected by being deleted. Thus, to find all affected instances efficiently we simply index each instance by the predicate on the filter edge of its current state. This is the problem addressed by pub/sub systems: Index a large number of predicates, such that for each in-

coming event all satisfied predicates are found efficiently. Hence we can leverage existing and proven technology for this task [FJL+01]. The for-all semantics of filter edges makes dealing with *simultaneously* arriving events easy—the affected instances are obtained as the union of the results returned by the index for each of the simultaneous input events.

Two challenges need to be addressed when using pub/sub engines. First, most pub/sub systems assume that predicates are conjunctions of atomic formulas. We can handle an arbitrary boolean formula by transforming it to *disjunctive normal form* (DNF), a disjunction of conjunctions of atomic formulas, and then registering each conjunction as a separate subscription in the pub/sub engine. The second challenge is parameterized predicates like ATT_1 relop ATT_2, because pub/sub systems expect static predicates of the form ATT relop CONST. To address this, in our current prototype implementation, we index only static predicates to avoid high index maintenance cost. Since all sub-formulas are conjunctions, the index returns a superset of the affected instances, which is postprocessed based on the parameterized predicates. In the remainder of this section, we introduce notation and then describe the Cayuga implementation in more detail.

4.1 Notation

A *static atomic predicate* is an atomic predicate of the form ATT relop CONST, e.g., `price` > 10. Other atomic predicates are referred to as *dynamic*. A dynamic predicate ATT_1 relop ATT_2 that compares an attribute value of the incoming event with an attribute of an earlier event is referred to as a *parameterized atomic predicate*. In the following discussion we consider only static and parameterized predicates; dynamic predicates of the form ATT_1 relop ATT_2, where ATT_1 and ATT_2 are both attributes of the incoming event, are treated by postprocessing as in [FJL+01]. We also assume all selection predicates are supplied in DNF.

Each conjunct P can be rewritten as $P = \bigwedge_i \text{ATT}_i$ relop $\text{CONST}_i \wedge \bigwedge_j \text{ATT}_j$ relop ATT_{k_j} by grouping the static atomic predicates and the parameterized dynamic atomic predicates together and then canonicalizing them, e.g. by sorting them lexicographically by attribute names. We refer to $\bigwedge_i \text{ATT}_i$ relop CONST_i and $\bigwedge_j \text{ATT}_j$ relop ATT_{k_j} as the *static* and *dynamic part* of P, respectively. If either part is empty, it is equivalent to the TRUE predicate.

A node of an automaton is *active* if there are active instances at this node, otherwise it is *inactive*. The start node is active by default. Similarly, we say all the outgoing edges of an active node are active as well.

4.2 Merging Automata

Our automata have a structure similar to the automata of YFilter [DAF+03] or linear finite automata in general. Hence we can use a similar procedure for merging common prefixes of different automata. Our procedure is slightly more general, since the union operator creates a DAG (directed acyclic graph), rather than a tree, and since there is a greater variety of edge types and edge labels. The final result of the merging process is a DAG of all automaton states.

Formally, the merging of automata is based on the following notion of *equivalent states*.

Definition 2. *Let n and m denote automaton nodes (states), and E_n and E_m denote the sets of edges entering n and m respectively. We define a nested sequence $\{\equiv_i \mid i = 0, 1, \ldots\}$ of equivalence relations on states as follows.*

- *$n \equiv_0 m$ for all n, m.*
- *$n \equiv_{i+1} m$ if and only if there exists a bijection $\sim$ between the entering edge sets E_n and E_m such that for each mapped pair $e_n \sim e_m$*
  - *e_n and e_m have identical edge labels, and*
  - *$e_n.source \equiv_i e_m.source$*

States n and m are equivalent, *written $n \equiv m$, if and only if $n \equiv_i m$ for all i.*

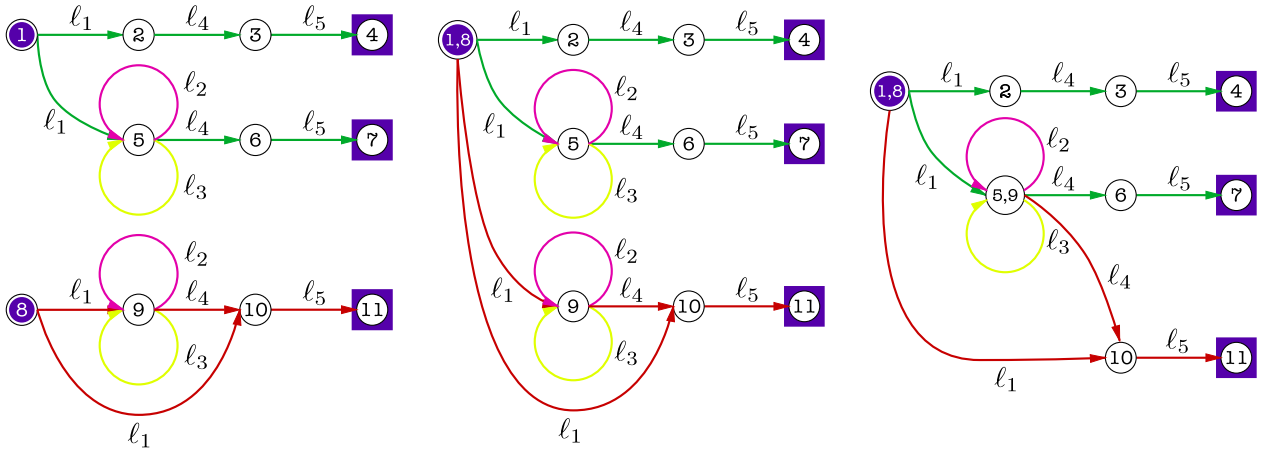
It is possible to compute $\equiv$ using a slight generalization of the traditional techniques for state minimization of finite automata [Hop71]. Our current implementation takes a simpler approach, merging only *prefixes* of paths of equivalent automaton states, as in [DAF+03]. The following example illustrates how we do this in a single pass. We merge automata one-by-one. The top diagram in Figure 9(a) shows the current merged automata DAG (it is fairly easy to prove by induction that it is a DAG with a single root node), while the bottom diagram in Figure 9(a) is the new query to be inserted.

In the first step, we can trivially merge the start states (see Figure 9(b)). Then we proceed with the other nodes of the new query in *topological order*, until no more merges are possible. In the example, we determine that node 9 is equivalent to node 5 (but not node 2, because of the self-loop edges). Hence we can merge the two nodes as well, resulting in the DAG shown in Figure 9(c). Node 10 is not equivalent to any of the other nodes at level 2 (root node is at level 0), hence the merging process terminates. Note that we cannot merge nodes 10 and 6, because of the additional edge with label ℓ_1.

We can identify equivalent nodes efficiently by computing a hash signature of the set of predicates for each incoming edge, and using this signature to prune the search space. This is fairly straightforward and not discussed here.

4.3 Efficient MQO Implementation

The overall system architecture is shown in Figure 10. Its core component is the *State Machine Manager*, which manages the merged query DAG and the active instances at the automaton nodes. It also maintains several indices for efficiently determining which automaton instances are affected by incoming events.



(a) Original DAGs to Merge (b) After merging start nodes (c) After merging nodes 5 and 9

Figure 9: State Merging Example

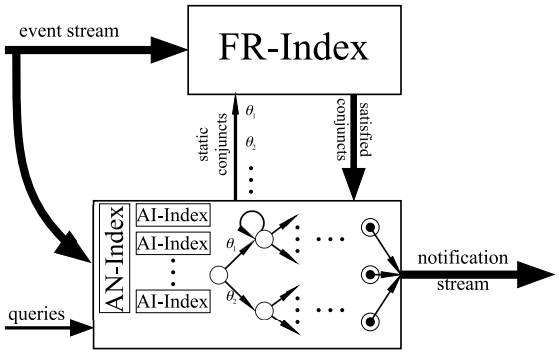


Figure 10: Cayuga architecture

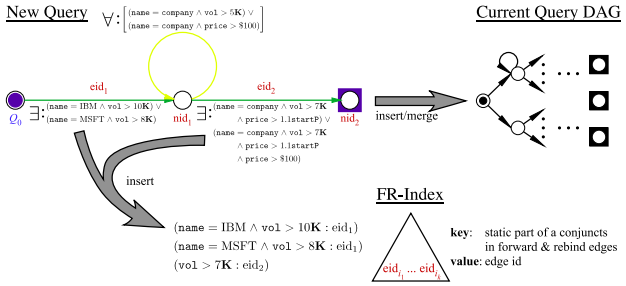


Figure 11: Insertion of a query

The Active Node (AN) index contains (key: value) pairs, where the key is the *static* part of a conjunction from the *filter* predicate of a node, and the value is a pointer to that node. The AN-index contains entries only for *active* nodes.

Each automaton node in the query DAG has an additional local index, the Active Instance (AI) index. This index contains the active automaton instances at the node, indexed by the *dynamic* part of the node's filter predicate. More precisely, for each conjunction of the DNF of the filter predicate and for each active instance at the node, there is an entry in AI-index with the dynamic part of the conjunction as key and the corresponding instance as value. Note there is no particular problem with indexing the dynamic part of a predicate associated with an *active* instance – while the instance is active, the attribute values associated with previous events are already bound, and can be treated as if they were constant.

Outside the State Machine Manager, there is the Forward/Rebind (FR) index. It indexes all forward and rebind edges by the *static* part of their edge predicates. More precisely, for each forward and rebind edge, and for each conjunction of the DNF of the edge's predicate, FR-index contains a (key: value) pair. The key is the static part of the conjunct and the value is the unique ID of the edge.

There are two types of updates that Cayuga needs to handle—insertion/deletion of queries and arrival of input events. A new query is inserted by first merging it into the query DAG in the State Machine Manager. This is shown in Figure 11 for a simple example query. For simplicity we assume that only the start states can be merged. Then, for each conjunction in the DNF of each forward and each rebind edge, an entry is added into FR-index. This entry has the static part of the conjunction as the key and the ID of the edge as the value. In the example, each of the two conjuncts of edge eid_1 results in a separate entry. AN-index and AI-index are not affected, because they maintain only active nodes and instances. When the query is deleted, the insertion process is simply reversed. Only nodes and edges that are not shared with other queries are physically removed from the DAG.

Incoming events are sent to both the State Machine Manager and the FR-index. Figure 12 illustrates how an event is processed right after the new query was inserted. In the example we omit the timestamp attributes for readability. Probing the FR-index produces a set of edge IDs as the result. This is the set of edges whose static predicate parts are satisfied by the event. Note the index returns both eid_1 and eid_2, based on their static predicate parts. For efficiency during later probing, the resulting set of edge IDs

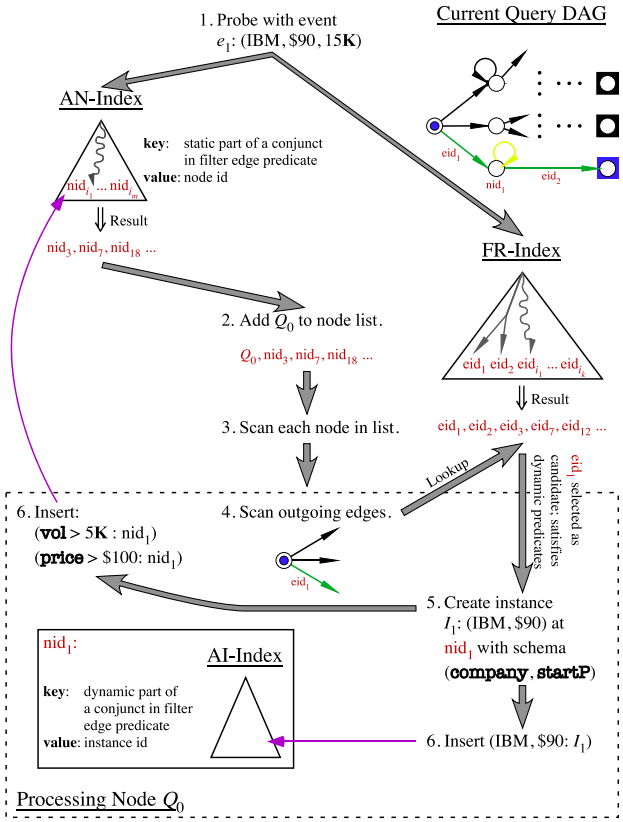


Figure 12: Processing first event

is stored in a hash table.

At the State Machine Manager, first AN-index is probed. It returns the set of active nodes whose filter edges are not traversed, based on the static parts of the filter predicate conjuncts. This immediately prunes a large number of active nodes, whose instances all traverse the filter edge. In the example in Figure 12 some arbitrary nodes are shown, but nid_1 is not among them, because it is not yet active. The system adds Q_0 to the result, because the start node is always active and relevant by default.

For each node in the result list of AN-index, the system determines which instances at the node are affected by the event. We discuss this step first for the start node only. When processing Q_0, we look up each of its outgoing edges in the result of FR-index (recall that it is stored in a hash table for fast lookups). If there is a hit, the corresponding edge is a *candidate* for a traversal. Recall the FR-index only uses the static parts of edge predicates; therefore, to eliminate false positives, we must test whether a candidate also satisfies the *dynamic* parts of the edge conjunctions. If it does, then the edge "fires" – we create a new instance and advance it to the destination node. In the example, the event satisfies the predicate on edge eid_1. Hence the corresponding instance is inserted into node nid_1's AI-index.

Figure 13 shows how the next incoming event is processed. AN-index and FR-index are probed as before. Processing Q_0 results in creation of a new instance at node nid_1 as before, but this time for the MSFT stock price.

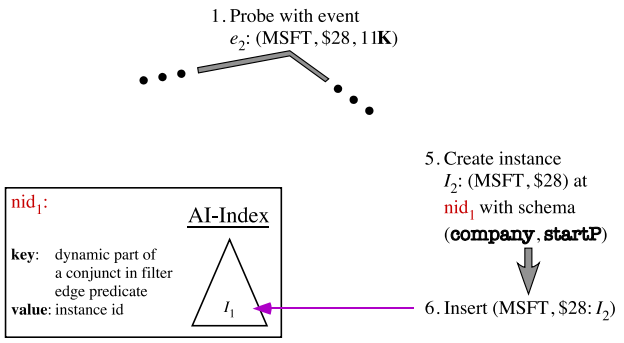


Figure 13: Processing second event

Note that AN-index returns node nid_1, because its static part `vol` $>$ 5K of the filter predicate is satisfied by the event. However, when processing the node, AI-index returns an empty result, because the event is for a different company (`company` $\neq$ `name`). This is exactly the expected behavior of the filter edge—it is traversed because the MSFT event did *not* satisfy the edge predicate for the IBM instance.

In Figure 14, we show how the next incoming event is processed. This event is for IBM. AN-index and FR-index are probed as before. Q_0 is also processed as before, but this time eid_1 is not traversed (FR-index filters it out). When processing node nid_1, probing its AI-index produces only I_1 as the result, because the *dynamic* part of its filter predicate is satisfied. I_2 is for MSFT and hence its dynamic filter predicate part is not satisfied and it traverses the filter edge. For I_1 we look up all outgoing forward and rebind edges of nid_1, only eid_2 in the example, in the result of FR-index. eid_2 is found and therefore its dynamic predicate parts are tested. Since they are satisfied by the event, instance I_1 traverses edge eid_2 and advances to the final state.

The diagram in Figure 15 summarizes the Cayuga event processing steps. On arrival of an event, the following happens:

1. FR-index generates a set of edges whose static predicate parts are satisfied by the event. This set is stored in a hash-index on edge ID.

2. AN-index generates a set of relevant active nodes.

3. For each node in the set we do the following. We first obtain the set of relevant active instances for which the filter condition is satisfied from AI-index. Then we determine for each relevant instance the candidates of satisfied edges by a lookup of the output of FR-index, followed by a verification of the edge predicates based on their dynamic parts.

In the above example and discussion, events have been assumed to arrive individually. However, simultaneous events pose no serious problems for our implementation, as long as all simultaneously arriving events are processed

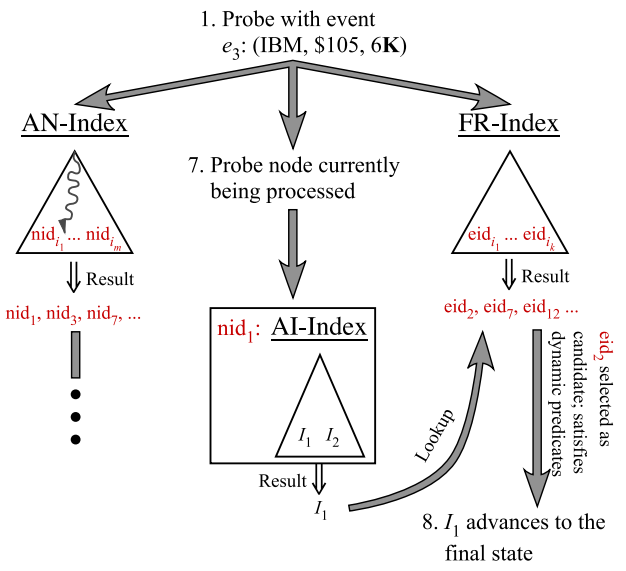


Figure 14: Processing third event

together. As mentioned earlier, the for-all semantics of filter edges implies that affected instances can be computed as the union of the affected instances for each of the simultaneous input events. The nondeterministic there-exists semantics of forward and rebind edges is naturally handled by first accumulating the set of new instances (generated by firing candidate edges) for all the simultaneous input events, then deleting old instances and installing the new ones atomically.

5 Performance Evaluation

We built an initial prototype implementation of Cayuga in C++. For standard data structures such as hash indices and lists we relied on the C++ Standard Library implementations. We believe that using specifically tailored implementations would lead to a considerable gain in system performance. However, even with the current prototype implementation we show that, with no more than a standard, off-the-shelf PC, we can process thousands of events per second, for hundreds of thousands of concurrently active sequence queries.

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 at least as fast as our system can process them. For this setup we measured the *total* runtime for matching all incoming events with all sequence queries in the system. For each experiment we perform several runs, clearing the cache between 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.

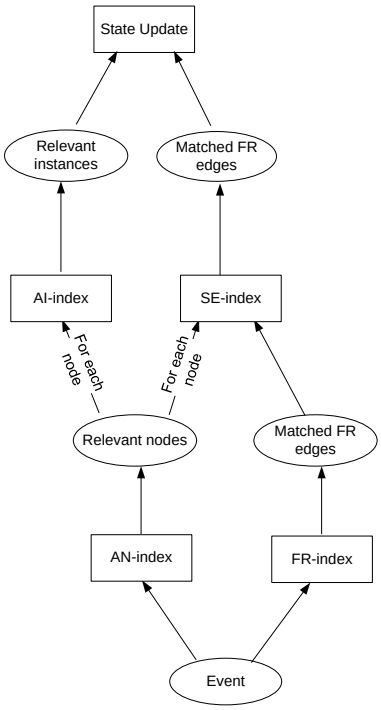


Figure 15: Event Processing Diagram

5.1 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 Instead of the stock stream example, we generated 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 both the stream and the associated queries are shown in Table 2.

We generated queries according to five different templates: `LinearStat`, `LinearDyn`, `Filter`, `NonDeterministic`, and `NonDeterministicAgg`. All queries are over a single input stream S; hence, as

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 queries	200,000
Number of atomic predicates (discrete + continuous)	2 + 2
Domain size of discrete attribute	100
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 2: Parameters (default values)

described in Section 2.1, S_i refers to an appropriately renamed occurrence of S in the algebraic expression.

`LinearStat` queries 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 event if it satisfies $\theta_1 \wedge \theta_2 \wedge \theta_3$. For example, if such a template where applied to our stock stream example, then our template might generate the following query.

Query 8. *Notify me when there are three consecutive stock quotes representing IBM below $10, followed by IBM above $15, and finally IBM below $15.*

As our input stream is not the stock stream, but a synthetic stream of eight attributes, 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 Query 8 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 $zipf_1$ 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` template

$$\sigma_{\theta_3}(\sigma_{\theta_2}(\sigma_{\theta_1}(S_1);S_2);S_3)$$

The difference between this template 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.

We measure the overhead of evaluating filter predicates with the `Filter` template

$$\sigma_{\theta_3}(\sigma_{\theta_2}(\sigma_{\theta_1}(S_1) \underset{\theta_4}{;} S_2) \underset{\theta_5}{;} S_3)$$

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 t is as shown in Table 2 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. To illustrate this idea with our stock stream example, suppose we took Query 8 and made θ_4 the filer predicate DUR $\leq 10\text{min} \wedge S_2$.`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 second filter formula θ_5 is similar to θ_4; we merely replace S_2.ATT with S_3.ATT.

The effect of non-determinism in our automata is measured by the `NonDeterministic` template

$$\sigma_{\theta_3} \circ \mu_{\text{ID},\theta_5}(\sigma_{\theta_2} \circ \mu_{\text{ID},\theta_4}(\sigma_{\theta_1}(S_1), S_2), S_3)$$

where ID is the identity unary operation. This query is much more powerful than the previous ones. An analogy using our example Query 8 would be a query that not only searches for patterns of *consecutive* IBM stock quotes, but one that can find *any 3-tuple* of IBM stock quotes that satisfies the duration constraints and selection criteria, ignoring all stock quotes (including other quotes for IBM) in between. Hence the output of this query will be a superset of the `Filter` query with exactly the same formulas θ_i.

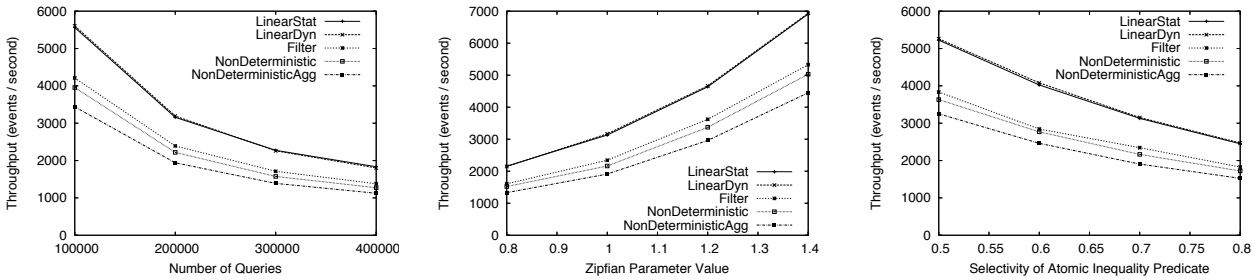
Finally, template `NonDeterministicAgg` implements aggregation. It extends `NonDeterministic` by computing the sum of the values of the continuous attributes, for the three events that satisfy the query pattern.

In processing these queries, 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. Different distributions only affect results by changing the selectivity of the edge predicates. The same effect is achieved by adjusting the query constants, and so we did not investigate this further.

5.1.1 Results

Figure 16 illustrates the results of various throughput experiments. Figure 16(a) shows how the system throughput changes with the number of subscriptions. Even for 400K concurrently active queries, 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 despite a challenging workload. Each event on average matches about 100 static predicates in the pub/sub engine. 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. Overall the `Filter` workload generated between 41 (100K queries) and 171 (400K queries) sequence matches, `NonDeterministic` and `NonDeterministicAgg` had a few more matches, and the linear workloads generated virtually no matches.



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

Figure 16: Throughput Measurements

This outcome is not surprising. Even though each single edge predicate by itself is not very selective, the overall pattern is very restrictive; the automata frequently advance to the first state, but only rarely reach the second or last state.

Note also that, despite the skewed query distribution, the merged query DAG is very large. For instance, before merging states the DAG for 100K queries 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 16(b), we compare the effect of parameter $zipf_1$ on system performance. Lower skewness makes the subscriptions less similar, hence reduces the possibilities for multi-query optimization. This can be observed in the graphs. 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 queries is virtually unaffected by the Zipf parameter, because there is no correlation between event values and query constants.

Finally, we examined the effect of edge predicate selectivity on the performance. Figure 16(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.

5.2 Comparison to Other Approaches

To justify our approach, we compare the performance of Cayuga to two other systems. In the first case, we compare Cayuga to a similar system without the multi-query optimization of Section 4, in order to understand the benefit of that optimization. In the second case, we compare Cayuga to STREAM, a substantially more expressive system, in order to demonstrate the performance benefits of our weaker language.

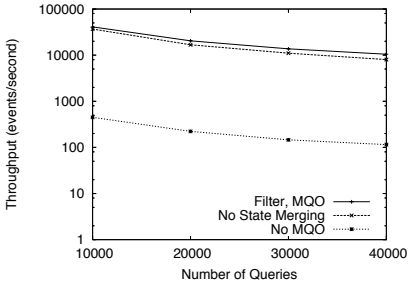


Figure 17: Effect of multi-query optimization

5.2.1 Naive Extension of Pub/Sub

In order to see the benefits of our multi-query optimization, we compare Cayuga with a naive extension of a pub/sub engine. This system uses a simple value-based pub/sub engine to filter incoming events. Users then post-process the output of the engine to recognize complex patterns. This post-processing can be treated as a black box, but for the sake of comparison, we use our automaton model to implement this black box. Hence the difference between this implementation and Cayuga is that, as each user runs her own individual automaton, it cannot have any multi-query optimization.

To simulate this setting, we take Cayuga and turn off state merging, as well as any indices that span multiple automata other than the FR-index (see Section 4). The FR-index is retained as its functionality could be provided in the naive extension by an external centralized pub/sub process where all users register their static predicates.

Figure 17 shows the performance of Cayuga compared to the two systems described above on our technical benchmark. To keep the runtime of the naive system manageable, we reduced the number of concurrently active queries to 10K-40K, compared to 100K-400K in our other experiments. The throughput measurements in Figure 17 are for the `Filter` workload. The curve "Filter, MQO" denotes the performance of Cayuga, while the "No MQO" curve represents the throughput of the naive extension. Note that the y-axis is a *log scale*; hence with multi-query optimization the system is faster by a factor of 100.

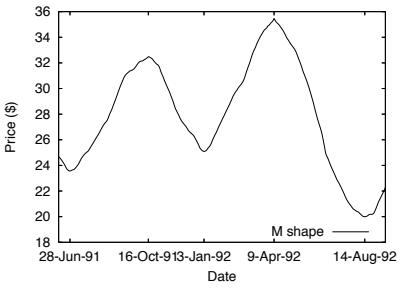


Figure 18: Double-Top pattern for Dell quotes

In order to explore the effectiveness of different multi-query optimization techniques, we also explored a Cayuga system where state merging was turned off, but in which the automaton indices worked properly. The performance of this system is the "No State Merging" curve in Figure 17. It is clear from this curve that most of the performance gain comes the indexing of active node, and not from merging automata states. In particular, this indexing dramatically reduces the cost of "joining" the pub/sub engine's result with the set of active automata edges.

5.2.2 Comparison to STREAM

The CESAR algebra used by Cayuga is not as expressive as many of the more traditional languages. We chose to introduce a less expressive language in order that we may implement it more efficiently. In order to illustrate this trade-off, we compare Cayuga to the Stanford STREAM system. STREAM is a general stream processing system with a relatively mature implementation, capable of processing the queries used in our experiments. Furthermore several of its operators are very similar to the ones used in Cayuga.

Since multi-query optimization has not been fully integrated into STREAM, we restrict the comparison to a single query at a time. We report results for a variant of the well-known Double-Top (or M-Top) pattern used for stock analysis (see for instance `http://www.stockcharts.com/education/ChartAnalysis/doubleTop.html`). The original Double-Top pattern is easy to express in Cayuga and can be processed efficiently, but it would be expensive for a system that was not optimized for non-deterministic matching.

We use a modified Double-Top pattern that consists of five consecutive local extrema of stock prices, starting with a minimum at price A, rising monotonically to reach a maximum price B, such that $B \geq 1.2A$, then falling monotonically to reach a minimum C, which is within 10% of the starting price A. Afterwards the stock rises monotonically to reach a new high of D, which is within 10% of the previous high B. Finally the stock falls monotonically to a new minimum E below A. Figure 18 shows an example of the pattern, found in a real sequence of stock closing prices.

The Double-Top query is naturally expressed in our algebra as a *linear-plus expression* with five μ operators (one for each monotonic sequence). It is essentially an extension to the `NonDeterministic` template in our technical benchmark, which contains only two μ operators.

It is possible to express Double-Top in our algebra without the μ operator. While the resulting query is not linear-plus, we can implement it using the idea of resubscription from Section 3.5. To see how, first we create the following expression for detecting local maxima.

$$\sigma_{\theta_1}(\sigma_{\theta_2}(S_1 \underset{\theta_3}{;} S_2) \underset{\theta_4}{;} S_3)$$

Here θ_1 is $(S_2.\texttt{price} > S_3.\texttt{price})$, θ_2 is $(S_2.\texttt{price} > S_1.\texttt{price})$, θ_3 is $(S_2.\texttt{name} = S_1.\texttt{name})$, and θ_4 is $(S_3.\texttt{name} = S_1.\texttt{name})$. Note that local minima can be detected similarly. We then define stream E to be the union of all local minima generated by the above expression. Given this stream E of local minima, we now use resubscription to express Double-Top as an expression linear-plus in E. The expression is

$$\sigma_{\theta_1}(\sigma_{\theta_2}(\sigma_{\theta_3}(\sigma_{\theta_4}(E_1 \underset{\theta_5}{;} E_2) \underset{\theta_6}{;} E_3) \underset{\theta_7}{;} E_4) \underset{\theta_8}{;} E_5)$$

where the selection formula are as follows:

$$\begin{aligned}
\theta_1 &\equiv (E_5.\texttt{price} \leq E_1.price) \\
\theta_2 &\equiv (0.9E_2.\texttt{price} \leq E_4.\texttt{price} \leq 1.1E_2.\texttt{price}) \\
\theta_3 &\equiv (0.9E_1.\texttt{price} \leq E_3.\texttt{price} \leq 1.1E_1.\texttt{price}) \\
\theta_4 &\equiv (E_2.\texttt{price} \geq 1.2 * E_1.\texttt{price}) \\
\theta_5 &\equiv (E_2.\texttt{name} = E_1.\texttt{name}) \\
\theta_6 &\equiv (E_3.\texttt{name} = E_1.\texttt{name}) \\
\theta_7 &\equiv (E_4.\texttt{name} = E_1.\texttt{name}) \\
\theta_8 &\equiv (E_5.\texttt{name} = E_1.\texttt{name})
\end{aligned}$$

STREAM's CQL query language lacks the μ operator. To efficiently find this pattern in CQL, the query is decomposed into manipulations on several levels of views (see Figure 19(a)). [1] Hence this implementation is is similar to the algebra expression with resubscription in our system. The STREAM implementation first computes a stream of "up" and "down" trends between consecutive quotes of the same stock. Then it detects local extrema in that stream. Finally, every sequence of five consecutive extrema is examined to determine whether the constraints on the price attribute are satisfied. Note that, while nicely optimized, this query had to created manually, and it requires considerable expertise to craft the query in this way.

A more direct way to formulate this query in CQL, denoted as CQL2, is to use self-joins. This approach is shown in Figure 19(b). First a 3-way self-join is used to discover local extrema. Then, on the resulting stream of extrema, we find the actual pattern using a 5-way self-join. In order to properly quantify the performance degradation caused by a single n-way self-join, in CQL2 we express the first part of the query in the standard 3-way self-join fashion, but use

[1] We would like to thank Arvind Arasu for crafting this CQL query formulation for us.

```
# Stock stream
table : register stream Stock (time integer, name integer, price float);

# Add difference to previous stock price to each tuple
vquery : Rstream (Select S.time, S.name, S.price, (S.price – P.price) From Stock [Now] as S,
      Stock [Partition By P.name Rows 2] as P Where S.name = P.name and S.time > P.time);
vtable : register stream StockDiff (time integer, name integer, price float, pdiff float);

# Generate stream of extrema
vquery : Rstream (Select P.time, P.name, P.price, P.pdiff From StockDiff [Now] as S,
      StockDiff [Partition By P.name Rows 2] as P
      Where S.name = P.name and (S.pdiff * P.pdiff) < 0.0);
vtable : register stream Extrema (time integer, name integer, price float, pdiff float);

# Assign unique sequence numbers to extrema points
vquery : Select name, count(*) from Extrema Group By name;
vtable : register relation ExtremaCounter (name integer, seqNo integer);

# Attach sequence numbers to extrema
vquery : Rstream (Select E.name, E.price, E.pdiff, C.seqNo, C.seqNo – 1
      From Extrema [Now] as E, ExtremaCounter as C Where E.name = C.name);
vtable : register stream ExtremaSeq (name integer, price float, pdiff float,
      seq integer, prevSeq integer);

# State A: minimum
vquery : Select name, price, seq from ExtremaSeq Where pdiff < 0.0;
vtable : register relation stateA (name integer, price float, seq integer);

# State B: maximum, B > 1.2 A
vquery : Rstream (Select E.name, E.price, A.price, E.seq From ExtremaSeq [Now] as E,
      stateA as A Where E.name = A.name and E.prevSeq = A.seq and E.price > (A.price*1.2));
vtable : register relation stateB (name integer, bprice float, aprice float, seq integer);

# State C: minimum, 0.9 A < C < 1.1 A
vquery : Rstream (Select E.name, E.price, B.bprice, B.aprice, E.seq From ExtremaSeq [Now] as E,
      stateB as B Where E.name = B.name and E.prevSeq = B.seq and E.price > (B.aprice * 0.9)
      and E.price < (B.aprice * 1.1));
vtable : register relation stateC(name integer, cprice float, bprice float, aprice float, seq integer);

# State D: maximum, 0.9 B < C < 1.1 B
vquery : Rstream (Select E.name, E.price, C.cprice, C.bprice, C.aprice, E.seq
      From ExtremaSeq [Now] as E, stateC as C Where E.name = C.name and E.prevSeq = C.seq
      and E.price > (C.bprice * 0.9) and E.price < (C.bprice * 1.1));
vtable : register relation stateD (name integer, dprice float, cprice float, bprice float, aprice float,
      seq integer);

# The final query: D < A
query : Rstream (Select E.name, E.price, D.dprice, D.cprice, D.bprice, D.aprice
      From ExtremaSeq [Now] as E, stateD as D Where E.name = D.name and E.prevSeq = D.seq
      and E.price < D.aprice);
```

(a) Formulation in STREAM (CQL1)

```
# Stock stream
table : register stream Stock (time integer, name integer, price float);

# Generate stream of extrema

vquery : Istream (Select S2.time, S2.name, S2.price, (S2.price–S1.price)
From Stock [Partition By S1.name Rows 3] as S1,
   Stock [Partition By S2.name Rows 3] as S2,
   Stock [Partition By S3.name Rows 3] as S3
Where S1.name = S2.name and S2.name = S3.name and
   (S2.price–S1.price) * (S3.price–S2.price) < 0.0 and
   S1.time < S2.time and S2.time < S3.time);
vtable : register stream Extrema (time integer, name integer, price float, pdiff float);


# Assign unique sequence numbers to extrema points
vquery : Select name, count(*) from Extrema Group By name;
vtable : register relation ExtremaCounter (name integer, seqNo integer);

# Attach sequence numbers to extrema
vquery : Rstream (Select E.name, E.price, E.pdiff, C.seqNo, C.seqNo – 1
      From Extrema [Now] as E, ExtremaCounter as C Where E.name = C.name);
vtable : register stream ExtremaSeq (name integer, price float, pdiff float,
      seq integer, prevSeq integer);

# State A: minimum
vquery : Select name, price, seq from ExtremaSeq Where pdiff < 0.0;
vtable : register relation stateA (name integer, price float, seq integer);

# State B: maximum, B > 1.2 A
vquery : Rstream (Select E.name, E.price, A.price, E.seq From ExtremaSeq [Now] as E,
      stateA as A Where E.name = A.name and E.prevSeq = A.seq and E.price > (A.price*1.2));
vtable : register relation stateB (name integer, bprice float, aprice float, seq integer);

# State C: minimum, 0.9 A < C < 1.1 A
vquery : Rstream (Select E.name, E.price, B.bprice, B.aprice, E.seq From ExtremaSeq [Now] as E,
      stateB as B Where E.name = B.name and E.prevSeq = B.seq and E.price > (B.aprice * 0.9)
      and E.price < (B.aprice * 1.1));
vtable : register relation stateC(name integer, cprice float, bprice float, aprice float, seq integer);

# State D: maximum, 0.9 B < C < 1.1 B
vquery : Rstream (Select E.name, E.price, C.cprice, C.bprice, C.aprice, E.seq
      From ExtremaSeq [Now] as E, stateC as C Where E.name = C.name and E.prevSeq = C.seq
      and E.price > (C.bprice * 0.9) and E.price < (C.bprice * 1.1));
vtable : register relation stateD (name integer, dprice float, cprice float, bprice float, aprice float,
      seq integer);

# The final query: D < A
query : Rstream (Select E.name, E.price, D.dprice, D.cprice, D.bprice, D.aprice
      From ExtremaSeq [Now] as E, stateD as D Where E.name = D.name and E.prevSeq = D.seq
      and E.price < D.aprice);
```

(b) Formulation in STREAM (CQL2)

Figure 19: Double-Top query formulation

the more efficient state-like expressions of CQL2 for the latter part.

Figure 20 shows the performance difference between the two equivalent queries in Cayuga, and the two equivalent queries in STREAM. We run a single instance of the Double-Top query on a stream of 112,635 real daily closing stock prices for 24 different companies listed at the NYSE. The effect of different degrees of smoothing (length of window for computing a running average) is examined. Note that stronger smoothing reduces the number of local extrema, and hence benefits the resubscription and STREAM query formulation.

The "Mu Formulation" in Cayuga corresponds to the natural linear-plus expression. This formulation clearly outperforms the equivalent "Resubscription" formulation in Cayuga, as well as the two CQL formulations in STREAM. This result supports our focus on linear-plus expressions in Section 3. It is important to note that the Cayuga resubscription formulation and CQL1 perform almost identically; this should not be too surprising as their

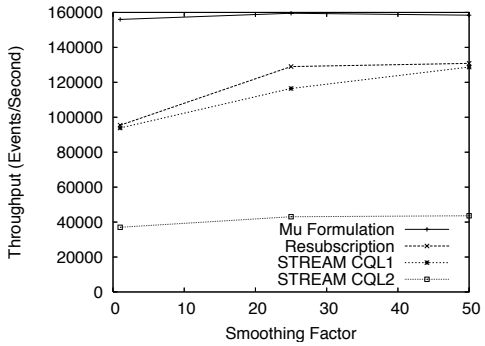


Figure 20: Comparison to STREAM

formulation is essentially identical. However, CQL1 is essentially a hand-optimized version of the Double-Top query. The more natural formulation of CQL2 had less than half of the throughput of CQL1, and a fourth of the throughput of the μ formulation. Hence there is potential for some synergy between Cayuga and STREAM, where we can use our algebra to optimize such pattern-search queries.

6 Related Work

To date, interest in building a DSMS has concentrated principally at the extremes of the expressiveness spectrum. At the low end of the spectrum lie pub/sub systems [ASS+99, YSG03, FJL+01]. These systems sacrifice expressiveness to achieve high performance. For example, Le Subscribe [FJL+01] is a very high-performance scalable pub/sub system that performs aggressive multi-query optimization. Work in this area includes scalable trigger mechanisms [HCH+99, SPAM91, TLP03].

Somewhat higher in the expressiveness spectrum is work from the Active Database community [WC96] on languages for specifying more complex event-condition-action rules. The composite event definition languages of SNOOP [CKAK94, AC03] and ODE [GJS92] 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 nondeterministic finite automaton model. The automaton construction of [GJS92] supports a limited form of parameterized composite events defined by equality constraints between attributes of primitive events.

Our own work can be viewed as extending this style of system with full support for parameterized composite events and support for aggregate queries. Despite the significant added expressiveness, our queries can still be evaluated by nondeterministic finite automata amenable to multi-query optimization using a combination of state merging and indexing techniques.

Still higher in the spectrum, several groups have described or are building systems with very expressive query languages [CcC+02, MWA+03, CCD+03, AAB+05]. Sistla and Wolfson [SW95] describe an event definition and aggregation language based on Past Temporal Logic. The TREPLE language [MZ97b] is a Datalog-based system with a precise formal specification; it extends the parameterized composite event specification language of EPL [MZ97a] with a powerful aggregation mechanism that is capable of explicit recursion. Perhaps the most powerful formal approach is STREAM's CQL query language [MWA+03], which extends SQL with support for window queries. Like SQL itself, CQL is declarative and admits of a formal specification [ABW03]; and there are some initial results characterizing a sub-class of queries that can be computed with bounded memory [SW04, ABB+02]. The STREAM system is quite mature, though it lacks multi-query optimization. A similarly powerful approach is represented by Aurora and Borealis [CcC+02, AAB+05]. These two systems, however, use a procedural boxes-and-arrows paradigm which is much less amenable to formal specification in our style. In [LWZ04] it is shown that SQL lacks expressive power for continuous queries on data streams, and Wang et al. extend SQL with features to support data mining and data streams [WZL03].

In general, the semantics of some of the more expressive event languages is not well-defined [GA02, ZU99], 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 continuous queries.

Efficient filtering and dissemination of information is a very active and diverse field of research. Due to lack of space, we only list selected approaches without claiming completeness. IR-style approaches to document filtering [cFG00, FD92, YGM99] typically rely on similarity measures between incoming documents and stored user profiles, but otherwise are conceptually similar to what we refer to as simple attribute-value pub/sub in this paper. Our use of a pub/sub engine to implement selection is similar to the idea of context-based subscriptions, although our algebra is much more expressive than the languages proposed in previous work [ASS+99, YSG03]. There have been several systems for large-scale filtering of streaming XML documents [AF00, DAF+03, NACP01, CFGR02, GS03, GMOS03, BGKS03]. Their query languages usually are fragments of XPath, which is more expressive than pub/sub, but not as powerful as STREAM's CQL. Specifically, XML filtering systems do not address parameterization.

Related to our implementation, Sellis [Sel88] is one of the first to address general multi-query optimization in databases. Traditionally this is performed by sharing operators and query results [BBD+02, CcC+02, CCD+03, KFHJ04, MSHR02, CDTW00, LPT99]. Our multi-query optimization is fundamentally different and aggressively exploits the relationship of our stream query algebra to automata.

7 Conclusions and Future Work

We presented CESAR, a novel algebra for processing data streams, and Cayuga a prototype implementation of this algebra. CESAR extends previous work on event processing in several directions. It adds built-in support for parameterization, aggregates, selection over infinite domains, and support for arbitrary streams of events, including simultaneous events and events with non-trivial duration. We developed a new automaton model for implementing algebra expressions efficiently. We discussed the challenges of implementing this automaton model, together with several strategies multi-query optimization. Finally, we presented several initial performance results showing the efficacy of our approach. We plan to extend this work by developing a complete optimization framework, including query rewrite

rules and more effective MQO strategies.

Apart from CESAR-specific optimizations, we see our work as a step towards understanding the fundamental tradeoffs involved in data stream processing. More precisely, how much scalability do we trade off for increasing expressiveness? CESAR is very different from Aurora's boxes-and-arrows approach and SQL-based languages like STREAM's CQL [ABW03]. It will be interesting to formally compare the expressiveness of the different languages by mapping them to a common powerful calculus, and to see how much expressiveness (i.e., new operators) we can add to CESAR, while still maintaining scalability. Notice that CESAR's operators and implementation are closer in spirit to XML filtering than to the above mentioned DSMSs. 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.

References

[AAB+05] 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.

[ABB+02] 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.

[ABW03] A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: Semantic foundations and query execution. Technical report, Stanford University, 2003.

[AC03] R. Adaikkalavan and S. Chakravarthy. Snoopib: Interval-based event specification and detection for active databases. In *Proc. ADBIS*, pages 190–204, 2003.

[AF00] M. Altinel and M. J. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *Proc. VLDB*, pages 53–64, 2000.

[ASS+99] 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.

[BBD+02] 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.

[BGKS03] N. Bruno, L. Gravano, N. Koudas, and D. Srivastava. Navigation- vs. index-based XML multi-query processing. In *Proc. ICDE*, pages 139–150, 2003.

[CcC+02] 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.

[CCD+03] 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.

[CDTW00] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *Proc. SIGMOD*, pages 379–390, 2000.

[cFG00] U. Çetintemel, M. J. Franklin, and C. L. Giles. Self-adaptive user profiles for large-scale data delivery. In *Proc. ICDE*, pages 622–633, 2000.

[CFGR02] 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.

[CKAK94] 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.

[DAF+03] 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.

[FD92] P. W. Foltz and S. T. Dumais. Personalized information delivery: An analysis of information filtering methods. *CACM*, 35(12):51–60, 1992.

[FJL+01] 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.

[GA02] A. Galton and J. C. Augusto. Two approaches to event definition. In *Proc. DEXA*, pages 547–556, 2002.

[GGR02] M. N. Garofalakis, J. Gehrke, and R. Rastogi. Querying and mining data streams: You only get one look. In *Proc. SIGMOD*, 2002.

[GJS92] 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.

[GMOS03] T. J. Green, G. Miklau, M. Onizuka, and D. Suciu. Processing XML streams with deterministic automata. In *Proc. ICDT*, pages 173–189, 2003.

[GS03] A. K. Gupta and D. Suciu. Stream processing of XPath queries with predicates. In *Proc. SIGMOD*, pages 419–430, 2003.

[HCH+99] E. N. Hanson, C. Carnes, L. Huang, M. Konyala, L. Noronha, S. Parthasarathy, J. B. Park, and A. Vernon. Scalable trigger processing. In *Proc. ICDE*, pages 266–275, 1999.

[HMU00] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 2nd edition, 2000.

[Hop71] J. Hopcroft. An n log(n) algorithm for minimizing the states in a finite automaton. In Z. Kohavi, editor, *The Theory of Machines and Computations*, pages 189–196. Academic Press, 1971.

[KFHJ04] S. Krishnamurthy, M. J. Franklin, J. M. Hellerstein, and G. Jacobson. The case for precision sharing. In *Proc. VLDB*, pages 972–986, 2004.

[LPT99] L. Liu, C. Pu, and W. Tang. Continual queries for internet scale event-driven information delivery. *IEEE TKDE*, 11(4):610–628, 1999.

[LWZ04] Y. Law, H. Wang, and C. Zaniolo. Query languages and data models for database sequences and data streams. In *Proc. VLDB*, pages 492–503, 2004.

[MSHR02] S. R. Madden, M. A. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *Proc. SIGMOD*, 2002.

[MWA+03] 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.

[MZ97a] I. Motakis and C. Zaniolo. Formal semantics for composite temporal events in active database rules. *Journal of Systems Integration*, 7(3-4):291–325, 1997.

[MZ97b] I. Motakis and C. Zaniolo. Temporal aggregation in active database rules. In *Proc. SIGMOD*, pages 440–451, 1997.

[NACP01] B. Nguyen, S. Abiteboul, G. Cobena, and M. Preda. Monitoring XML data on the Web. In *Proc. SIGMOD*, pages 437–448, 2001.

[PSB03] P. R. Pietzuch, B. Shand, and J. Bacon. A framework for event composition in distributed systems. In *Proc. ACM/IFIP/USENIX International Middleware Conference*, pages 62–82, 2003.

[Sel88] T. K. Sellis. Multiple-query optimization. *ACM TODS*, 13(1):23–52, 1988.

[SPAM91] U. Schreier, H. Pirahesh, R. Agrawal, and C. Mohan. Alert: An architecture for transforming a passive DBMS into an active DBMS. In *Proc. VLDB*, pages 469–478, 1991.

[SW95] A. P. Sistla and O. Wolfson. Temporal conditions and integrity constraints in active database systems. In *Proc. SIGMOD*, pages 269–280, 1995.

[SW04] U. Srivastava and J. Widom. Memory-limited execution of windowed stream joins. In *Proc. VLDB*, pages 324–335, 2004.

[TLP03] W. Tang, L. Liu, and C. Pu. Trigger grouping: A scalable approach to large scale information monitoring. In *NCA*, pages 148–155, 2003.

[tra] Traderbot financial search engine. http://www.traderbot.com/.

[WC96] J. Widom and S. Ceri, editors. *Active Database Systems: Triggers and Rules For Advanced Database Processing*. Morgan Kaufmann Publishers, 1996.

[WZL03] H. Wang, C. Zaniolo, and C. Luo. ATLAS: A small but complete SQL extension for data mining and data streams. In *Proc. VLDB*, pages 1113–1116, 2003.

[YGM99] T. W. Yan and H. Garcia-Molina. The sift information dissemination system. *ACM TODS*, 24(4):529–565, 1999.

[YSG03] A. Yalamanchi, J. Srinivasan, and D. Gawlick. Managing expressions as data in relational database systems. In *Proc. CIDR*, 2003.

[ZU99] D. Zimmer and R. Unland. On the semantics of complex events in active database management systems. In *Proc. ICDE*, pages 392–399, 1999.