Cayuga: A General Purpose Event Monitoring System

Alan Demers[1], Johannes Gehrke[1], Biswanath Panda[1]
Mirek Riedewald[1], Varun Sharma[2][*], and Walker White[1][†]
[1]Department of Computer Science, Cornell University
[2]Computer Science and Engineering, Indian Institute of Technology Delhi

ABSTRACT

We describe the design and implementation of the Cornell Cayuga System for scalable event processing. We present a query language based on Cayuga Algebra for naturally expressing complex event patterns. We also describe several novel system design and implementation issues, focusing on Cayuga's query processor, its indexing approach, how Cayuga handles simultaneous events, and its specialized garbage collector.

1. INTRODUCTION

A large class of both well-established and emerging applications can best be described as *event monitoring applications*. These include supply chain management for RFID (Radio Frequency Identification) tagged products, real-time stock trading, monitoring of large computing systems to detect malfunctioning or attacks, and monitoring of sensor networks, e.g., for surveillance. There is great interest in these applications as indicated by the establishment of sites like `http://www.complexevents.com`, which bring together major industrial players like BEA, IBM, Oracle, and TIBCO. Event monitoring applications need to process massive streams of events in (near) real-time.

Event processing differs from general data stream management in two major aspects of the query workload. First, its has a distinct class of queries, which warrants special attention. In complex event processing, users are interested in finding matches to *event patterns*, which are usually sequences of correlated events. An important class of pattern is what we call a *safety condition*, where we want to ensure that between two events "nothing bad" happens. For example, between leaving the farm (start event) and arriving at the store (end event), fresh produce should not have spent more than 1 hour total above a temperature of 25 °C. Traditional data stream languages are not designed for event monitoring. While it is possible to express event patterns, this is cumbersome and results in queries that are almost impossible to read and to optimize.

Second, in complex event processing, there is usually a large number of concurrent queries registered in the event processing system. This is similar to the workload of publish/subscribe systems. In comparison, data stream management systems are usually less scalable in the number of queries, capable of supporting only a small number of concurrent queries.

[*]Work done while visiting Cornell University.

[†]{ademers,johannes,bpanda,mirek,vs227,wmwhite}@cs.cornell.edu

We have designed and built Cayuga, a general-purpose system for processing complex events on a large scale [8]. Cayuga supports on-line detection of a large number of complex patterns in event streams. Event patterns are written in a query language based on operators which have well-defined formal semantics. This enables Cayuga to perform query-rewrite optimizations. All operators are composable, allowing Cayuga to build up complex patterns from simpler sub-patterns. In addition to an expressive query language, the Cayuga system implements several novel techniques for query processing, indexing, and garbage collection, resulting in an efficient execution engine that can process data streams at very high rates. Cayuga offers applications a unique combination of expressiveness and speed.

In this paper, we describe two core aspects of Cayuga. First, we present a query language designed for naturally expressing complex event patterns and illustrate its use through examples. Then we describe the architecture of the Cayuga system. We discuss important design decisions that enable Cayuga to gracefully handle many input streams, high event stream rates, and also a large number of continuous queries.

The rest of the paper is organized as follows. In Section 2 we present the Cayuga data model and query language. Section 3 describes our automaton processing model and how queries are implemented by automata. In Section 4 we give a detailed discussion of the Cayuga system architecture and implementation. We summarize related work in Section 5. Finally, Section 6 describes three ongoing deployments of Cayuga at Cornell and then concludes the paper.

2. THE CAYUGA QUERY MODEL

The Cayuga data model and query algebra are explained in detail in Demers et. al [8]. Several unique features of the data model lead to important design decisions in the Cayuga system. In this section, we introduce the Cayuga Event Language (CEL), a query language based on the Cayuga query algebra.

2.1 Data Model

Like a traditional relational database system, Cayuga treats data as relational tuples, referred to as events. However, Cayuga is designed to monitor streams of events, not static tables. Thus, rather than sets of tuples, the Cayuga data model consists of temporally ordered *sequences* of tuples, referred to as event streams. Each event stream has a fixed relational schema. Each event in the stream has two timestamps, the start timestamp, denoted as t_0, and the end timestamp, denoted as t_1. Together they represent a duration interval, defined by $t_1 - t_0$. Events are serialized in order of t_1; for this reason, t_1 is also referred to as the "detection time" of a event. Because of the semantic issues discussed in White et al. [16], we consider events with the same detection time to be simultaneous, and guarantee to produce the same result regardless of the order of processing these simultaneous events. This guarantee is realized through *epoch-based processing* in Cayuga, as will be described in Section 4.3.

2.2 Query Language

The Cayuga Event Language is based on the Cayuga algebra [8], designed for expressing queries over event streams. It is a simple mapping of the algebra operators into a SQL-like syntax.

We introduce the query language through several examples. Our application domain for these examples is stock monitoring. We assume a stock ticker stream with the schema Stock(Name, Price, Volume). Each CEL query has the following simple form:

```
SELECT < attributes >
FROM < stream expression >
PUBLISH < output_stream >
```

The SELECT clause in CEL is similar to the SQL SELECT clause. It specifies the attribute names in the output stream schema, as well as aggregate computation. Attributes can be renamed by AS constructs in the SELECT clause. The SELECT clause is optional; omitting it is equivalent to specifying SELECT *. The PUBLISH clause names the output stream, so other queries may refer to it as input. When this clause is omitted, the output stream is unnamed.

The simplest type of query is one that just reads all the events from one stream and forwards them to another, as shown in Example 1.

Example 1. Suppose we want to select every input event from input stream Stock, and output it to a new stream named MyStock. We can formulate this query in CEL as follows.

```
SELECT *
FROM Stock
PUBLISH MyStock
```

We refer to the expression in the FROM clause as a *stream expression*. This expression is the core of each query. A stream expression is composed using a unary construct, FILTER, and two binary constructs, NEXT and FOLD. Each of these constructs produces an output stream from one or two input streams. We introduce them by examples.

The FILTER{predExpr} construct selects those events from its input stream that satisfy the predicate defined by predExpr, as shown in the following example.

Example 2. Suppose we want to select all IBM stock quotes whose price is above \$83. We can formulate this query in CEL as follows.

```
SELECT Price AS IBMPrice
FROM FILTER{Name='IBM' AND Price > 83}(Stock)
PUBLISH IBMStock
```

This query outputs only the Price attribute values of these events, not the stock Name or Volume. Furthermore, it renames the Price attribute to IBMPrice. The output stream is named IBMStock.

As in Cayuga algebra [8], a special attribute denoted as DUR, can be used in the predicate associated with FILTER. A predicate constraint on DUR is referred as *duration predicate*. As is described in Section 2.1, attribute DUR of an event e takes the duration interval of e as the value. For example, an event e satisfies duration predicate DUR $>$ 10min if its duration interval is greater than 10 minutes.

CEL, like SQL, is compositional, allowing sub-queries in the FROM clause. For example, Example 3 gives an equivalent formulation of Example 2 using nested queries.

Example 3. Again suppose we want to select all IBM stock quotes whose price is above \$83. Another way to formulate this query in CEL as follows.

```
SELECT Price AS IBMPrice
FROM FILTER{Name='IBM'}
    (SELECT *
    FROM FILTER{Price > 83}(Stock))
PUBLISH IBMStock2
```

The sub-query in the FROM clause first produces all quotes whose Price is above \$83. The top level query then further filters those quotes to retain only the quotes on IBM.

Although not illustrated in Example 3, we can also publish the outputs of the nested sub-query as a separate stream of its own. We need only add an additional PUBLISH clause to the parenthesized sub-query. This enables one query formulation to produce multiple output streams.

Binary constructs in the stream expression allow us to correlate events over time. The first binary construct is NEXT{predExpr}. When applied to two input streams S_1 and S_2 as S_1 NEXT{predExpr} S_2, this construct combines each event e_1 from S_1 with the next event in S_2 which satisfies the predicate defined by predExpr and occurs after the detection time of e_1. When predExpr as the parameter of NEXT construct is omitted, it is by default set to TRUE. For example, S NEXT S returns a pair of consecutive events from stream S.

Example 4. Suppose we want to match pairs of stock quotes, where the first is an IBM quote with a price above \$83 and the second is the next Microsoft quote to appear in the stream. We can formulate this query in CEL as follows.

```
SELECT IBMPrice, Price AS MSFTPrice
FROM IBMStock
        NEXT{Name='MSFT'} (Stock)
```

Each event in the output stream of this query consists of a pair prices: an IBM price above \$83 and the next MSFT price. In this query, the NEXT construct reads two input streams, where the first stream IBMStock is produced by Example 2, whose schema contains only one attribute IBMPrice, and the second stream is Stock. Alternatively, we could have "inlined" the query from Example 2.

A more powerful use of the NEXT construct exploits what we call "parameterization," the ability of the predExpr to refer to attributes from both its input streams, as in the following example.

Example 5. Suppose we want to match pairs of stock quotes, where the first is an IBM quote with a price above \$83 and the second is the next quote (of any stock) with a price above the IBM price from the first quote. We can formulate this query in CEL as follows.

```
SELECT IBMPrice, Price, Name
FROM IBMStock
        NEXT{IBMPrice < Price} (Stock)
```

Each event in the output stream of this query consists of an IBM price together with the name and price of the next stock to sell at a higher price.

In Examples 4 and 5 the two input streams of the NEXT construct have disjoint schemas. Of course this is not typical – the two input streams of a binary construct frequently contain identically named attributes. This situation happens to the binary join operator in relational algebra or SQL as well, for example in self-joins. When this happens to a binary construct, the reference to an attribute name in the predicate associated with the construct could become ambiguous, since the attribute could be from either one of the two input streams.

To address such reference ambiguity in CEL, we introduce special language constructs, referred to as *decorators*, to identify the streams from which attributes are taken. \$1.foo refers to attribute foo in the first input stream of a binary construct, or the single input stream of a unary construct. Similarly, \$2.foo refers to attribute foo in the second input stream of a binary construct. The decorator of an attribute can be omitted when it is in the schema of only one input stream. For simplicity of CEL, decorators are allowed only in predicate expressions, and cannot be used in the SELECT clause. It is helpful to view the SELECT clause as receiving one input stream whose schema is produced by the FROM clause.

The following example illustrates a simple use of decorators.

Example 6. Suppose we want to match pairs of stock quotes with identical prices, and return the stock name of the second quote of each pair. We can formulate this query in CEL as follows.

```
SELECT Name
FROM (SELECT Price FROM Stock)
        NEXT{$1.Price = $2.Price} (Stock)
```

Each event in the output stream of this query consists the name of the second stock of a pair of quotes with identical prices. Note that the Name parameter occurs only in the schema of the second input to the NEXT construct, so its use is unambiguous.

While NEXT allows us to correlate two events, there are many situations where we need to iterate over an a-priori unknown number of events until a stopping condition is satisfied. This capability is supplied by the FOLD construct. A FOLD construct has the form FOLD{predExpr$_1$, predExpr$_2$, aggExpr}. The three parameters respectively denote (1) the condition for choosing input events in the next iteration; this plays the same role as predExpr in NEXT{predExpr} (2) the stopping condition for iteration, and (3) aggregate computation between iteration steps. Intuitively, FOLD is an iterated form of NEXT that looks for patterns comprising two or more events.

As with NEXT, we use decorators \$1 and \$2 respectively to refer to attributes in the first and the second input streams of FOLD. To refer to attributes in the *last* iteration of FOLD from the second stream, we use decorator \$.

Example 7. Suppose we want to find a monotonically increasing run of prices for a single company, where the run lasts for at least 10 stock quotes, and the first quote has a volume greater than 10000. We can formulate this query in CEL as follows.

```
SELECT *
FROM FILTER{cnt > 10} (
  (SELECT *, 1 AS cnt FROM
  FILTER{Volume > 10000}(Stock))
    FOLD{$2.Name = $.Name, $2.Price > $.Price,
          $.cnt+1 AS cnt}
  Stock)
```

In Example 7 the first input stream of FOLD is produced by a FILTER construct that retains only quotes of volume greater than 10000. A new attribute cnt is added to that stream schema and initialized with value 1. The second input stream of FOLD contains all stock quotes. The first parameter of FOLD ensures that only stock quotes on the same company will be iterated over. The second parameter ensures that iteration will be stopped if the price of the current quote is not greater than that of the price in the last iteration. Finally, the third parameter computes the count aggregate, by adding 1 to the the value of cnt attribute in each iteration. Finally, for each output event of the FOLD construct, we run a filter over it to check whether its duration is greater than 1 hour.

To ensure valid iterations in FOLD construct, we maintain a *schema inclusion invariant* that the schema of its first input stream be a superset of the schema of its second input stream. Queries that violate this invariant will be rejected as being illegal. For more details on the formal semantics of FOLD and the invariant, we refer readers to [8].

Note that when the two input streams of a binary construct have identically named attributes, without proper renaming, the output stream of the binary construct will have duplicate attribute names, making the data semantics ambiguous. In relational algebra or SQL, this situation is addressed by explicitly renaming the output attribute names to make them distinct. Similarly for each NEXT construct, explict renaming can be used to avoid name collisions in its output schema. This is illustrated in Example 4, we had renamed the attribute Price in the first input stream schema of NEXT to IBMPrice.

For the FOLD construct, however, collisions cannot be avoided due to the schema inclusion invariant. For example, without attribute renaming, the output stream schema of Example 7 will contain two attributes named Price, among other duplicate attributes. We use an automatic renaming scheme as follows to make sure the attributes in the output stream schema have distinct names. It applies to both NEXT and FOLD as follows.

For sub-expression R NEXT{predExpr} S, where R and S denote the input streams of the binary construct NEXT, if R and S do not have attribute name collision, the output schema will be a cross product of the two input schemas of R and S, and no renaming is performed. Otherwise, we rename *each* attribute a in R to a_1. For uniformity, even attribute names in R that do not appear in S are renamed this way. However, no renaming is performed on attribute names of S. It is possible that after this renaming operation, there are still duplicate attribute names in the output schema (consider the case when S has an attribute named a_1). In this case, the input query will be rejected as illegal.

For sub-expression R FOLD{unaryExpr, predExpr, aggExpr} S, for each attribute a that occurs in both R and S, the value of a in R will be stored in attribute a_1 in the output schema of the above sub-expression, and the value of attribute a in the latest iteration of S will be stored in attribute a in the output schema. Each attribute b in R but not in S will still be named b in the output schema. For example, the output schema of Example 7 is (Name_1, Price_1, Volume_1, cnt_1, Name, Price, Volume, cnt). Note that with attribute renaming, we avoided the use of hierarchical decorators such as \$1.\$1.foo to refer to attribute foo in the first stream S of expression (S NEXT S) NEXT S. Hierarchical decorators are therefore not allowed in CEL.

To illustrate the use of decorators and attribute renaming for NEXT construct, we give the following query formulation.

Example 8. Suppose we want to match pairs of IBM stock quotes, where the first quote has a price above \$83 and the second is the next IBM quote whose price is higher than the price in the first

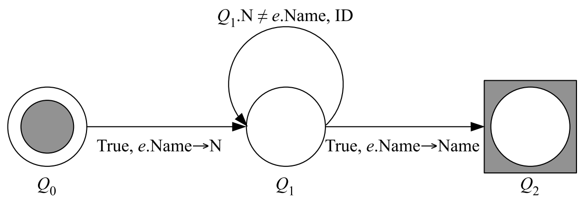

Figure 1: A Cayuga Automaton

quote. We can formulate this query in CEL as follows.

SELECT Price_1 AS IBMPrice1, Price AS IBMPrice2
FROM (FILTER{Name='IBM' AND Price > 83}(Stock))
NEXT{$2.Price > $1.Price}
(FILTER{Name='IBM'}(Stock))

Note that attribute Price_1 in the SELECT clause refers the attribute Price in the first input stream of NEXT, while the attribute Price comes from the second input stream.

We believe our renaming scheme, when used appropriately, makes it easier to write queries by rendering explicit renaming unnecessary, and thus improves the user-friendliness of CEL.

3. PROCESSING MODEL

Demers et al. [8] showed that any left- associated Cayuga algebra expression can be implemented by a variant of a nondeterministic finite state automaton, referred to as a *Cayuga automaton*. Non-left-associated expressions can be broken up into a set of left-associated ones, and will therefore be implemented by a set of corresponding Cayuga automata. Since CEL is based on Cayuga Algebra, these results are applicable to CEL queries as well. In this section, we describe how to process CEL queries with Cayuga automata.

Cayuga automata generalize on traditional NFAs in two ways: (1) instead of a finite input alphabet they read arbitrary relational streams, with state transitions controlled using predicates; and (2) they can store data from the input stream, allowing selection predicates to compare incoming events to previously encountered events.

Each automaton state is assigned a fixed relational schema, as well as an input stream. All the out-going edges of a state read that input stream. Each edge, say between states P and Q, is labeled by a pair $\langle\theta, f\rangle$, where θ is a predicate over $\mathrm{schema}(P) \times \mathrm{schema}(S)$; and f, the *schema map*, is a partial function taking $\mathrm{schema}(P) \times \mathrm{schema}(S)$ into $\mathrm{schema}(Q)$. The Cayuga automata operate as follows. Suppose an automaton instance is in state P with stored data x (note x conforms to $\mathrm{schema}(P)$). Let an event e arrive on stream S such that $\theta(x, e)$ is satisfied. Then the machine nondeterministically transitions to state Q, and the stored data becomes $f(x, e)$.

In Cayuga automata, the self loop edges derived from predicates that filter events (i.e., predExpr in NEXT{predExpr}, and predExpr$_1$ in FOLD{predExpr$_1$, predExpr$_2$, aggExpr}) are called *filter edges*, and we call the associated predicates *filter predicates*; self loop edges derived from predicates that "rebind" attributees (i.e., predExpr$_2$ in FOLD{predExpr$_1$, predExpr$_2$, aggExpr}) are called *rebind edges*, and we call the associated predicates *rebind predicates*; other edges are *Forward* edges. We adopt the convention that filter edges are drawn on top of the states, and rebind edges below the states.

Intuitively, each intermediate state with a filter edge but no rebind edge implements a NEXT construct. Each intermediate state with both a filter and a rebind edge implements a FOLD construct. For example, an automaton implementing SELECT Name FROM (SELECT Name AS N FROM Stock) NEXT{$1.N = $2.Name} Stock is shown in Figure 1. According to the CEL formulation, The schema of state Q_1 has one attribute N, and the schema of Q_2 has one attribute Name. Both Q_0 and Q_1 read input stream Stock.

Predicates in CEL formulations are translated into automaton edge predicates in an obvious way. In particular, Attribute decorators in CEL are translated into prefixes e. or Q. for a given automaton edge predicate, depending on whether the attribute comes from the schema of the current event read by that edge, denote as e, or from the schema of the automaton state, denoted as Q, from which the edge emanates. For example, in Figure 1, the predicate on the forward edge between Q_1 and Q_2 compares the value of attribute N from state Q_1 with the value of attribute N from the input stream Stock.

Predicates in Cayuga automata are associated with edges. However, since there is always one filter edge for each state (except for start and end states), we can associate predicates on filter edges with automaton states without ambiguity. Similarly, since there is at most one rebind edge for each state, associating rebind edge predicates with automaton states is also not ambiguous.

Note that the predicate on a filter edge is the *negation* of the corresponding filter predicate in CEL formulation. For example, in Figure 1, the predicate on the filter edge associated with state Q_1 is Q_1.N $\neq e$.Name, while its corresponding filter predicate in the CEL formulation is $1.N = $2.Name. In the following text, to avoid ambiguity, we avoid using the term filter edge predicate. To be consistent with the notion of a filter predicate in CEL formulations, in the context of automaton edge predicates, we use the term *filter predicate associated with state Q* to refer to the negation of the predicate of the filter edge associated with Q. For example, in Figure 1, the filter predicate associated with state Q_1 is Q_1.N $=$ e.Name. Recall that we refer to the predicate on a rebind (resp. forward) edge as its rebind (resp. forward) predicate.

The schema maps of Cayuga automata are also constructed from the CEL formulations in an obvious way. Note that the schema map associated with a filter edge is always the identity function, and our implementation exploits this fact.

Any Cayuga automaton maintains the following invariants on its edge predicates.

- For any automaton instance I under state P, if the current event together with I satisfy the predicate of a forward edge from state P to Q, then they must together satisfy the filter predicate associated with state P.
- For any automaton instance I under state P, if the current event together with I satisfy the predicate of a forward edge from state P to Q, then they must together satisfy the rebind predicate associated with state P, if there is one.
- For any automaton instance I under state P, if the current event together with I satisfy the rebind predicate associated with state P, then they must together satisfy the filter predicate associated with state P.

A consequence of these invariants is that for any automaton instance I under state P, if the current event together with I does not satisfy the filter predicate associated with state P, then none of the predicates on the rebind or forward edges associated with state P will be satisfied. Therefore the instance I must traverse the filter edge of P and is unmodified (due to the identity schema map of the filter edge). In this case we say instance I is *not affected* by the current event. Otherwise, if the current event together with I satisfies the filter predicate of P, we say I is *affected* by the current event.

These invariants can be easily realized in the implementation by predicate conjunctions. For example, the first invariant could be realized by attaching the filter predicate of state P as a conjunct to the predicate of each forward edge leaving P, and to the rebind predicate associated with P, if there is one. With an understanding of these invariants, to simplify the presentation, in the automaton figures we usually do not duplicate filter predicates on forward or rebind predicates. For example, in the automaton shown in Figure 1, the predicate of the forward edge from Q_1 to Q_2 has semantics Q_1.N = e.Name. However, we decide not to show the filter predicate of Q_1 as a conjunct in this forward predicate, and therefore denote the forward predicate as TRUE.

Complete details of the NFA construction can be found in [8].

3.1 Automaton Example

To illustrate how Cayuga automata process a query, we present an extended example.

Example 9. Suppose we want to record the quotes for any stock s, for which there is a monotonic decrease in price for at least 10 minutes, and which started at a large trade (Volume $>$ 10,000). Furthermore, suppose we are only interested in those monotonic runs that which are followed by one last stock quote whose prices is 5% above the previously seen (bottom) price. In other words, the stock has been steadily decreasing, but now shows signs of rising again. We can formulate this query in CEL as follows.

```
SELECT Name, MaxPrice, MinPrice, Price AS FinalPrice
FROM
  FILTER{DUR≥ 10min} (
    (SELECT Name, Price_1 AS MaxPrice, Price AS MinPrice
    FROM FILTER{Volume > 10000}(Stock))
      FOLD{$2.Name = $.Name, $2.Price < $.Price, }
    Stock)
    NEXT{$2.Name = $1.Name AND
          $2.Price >1.05*$1.MinPrice}
  Stock
```

In this formulation, the FOLD construct searches for a monotonically decreasing sequence for the same stock, ignoring quotes from other companies. During each iteration, the current lowest price is compared to the price of the incoming event. If a new minimum price is found, the concatenation overwrites the previously lowest price by the new one, otherwise the monotonic sequence has ended. When the duration constraint FILTER{DUR≥ 10min} evaluated on the output of FOLD is satisfied, a complex event is output denoting a mononotically decreasing sequence that lasts for no less than 10 minutes. Finally, the NEXT construct finds the next quote for the same company. If the price of that quote is %5 above the previous price, the query produces an output event. The output event retains only four attributes Name, MaxPrice, MinPrice, and FinalPrice.

The corresponding automaton is shown in Figure 2. We associate each edge with an edge predicate θ_i and an attribute mapping F_j. They are defined as follows.

$$\begin{aligned} F_1 &= e.Name \mapsto Name\_1, e.Price \mapsto Price\_1, \\ &\quad e.Name \mapsto Name\_n, e.Price \mapsto Price\_n \\ F_2 &= ID : Schema(A) \mapsto Schema(A) \\ F_3 &= A.Name\_1 \mapsto Name\_1, A.Price\_1 \mapsto Price\_1, \\ &\quad e.Name \mapsto Name\_n, e.Price \mapsto Price\_n \\ F_4 &= e.Name \mapsto Name, A.Price\_1 \mapsto MaxPrice, \\ &\quad e.Price \mapsto MinPrice \end{aligned}$$

$$\begin{aligned} F_5 &= ID : Schema(B) \mapsto Schema(B) \\ F_6 &= e.Name \mapsto Name, B.MaxPrice \mapsto MaxPrice, \\ &\quad B.MinPrice \mapsto MinPrice, e.Price \mapsto FinalPrice \end{aligned}$$

Note that these mappings are not true functions; F_1 maps Name to both Name_1 and Name_n. This is to support the value duplication necessary for initializing an iteration loop. Given these attribute mappings, we define the edge predicates as follows.

$$\begin{aligned} \theta_1 &\equiv e.Volume > 10,000 \\ \theta_2 &\equiv e.Name = A.Name\_n \\ \theta_3 &\equiv e.Price < A.Price\_n \\ \theta_4 &\equiv e.t_1 - A.t_0 \geq 10\,\text{min} \\ \theta_5 &\equiv e.Name = B.Name \\ \theta_6 &\equiv e.Price > 1.05 * B.MinPrice \end{aligned}$$

The predicates on automaton edges are much the same as the predicates in CEL formulation, except that the attribute names are decorated with the sources where they come from. By $e.t_1$, we mean the end time of the event traversing the edge with this predicate; $A.t_0$ is the start time of the instance stored at state A.

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

For this example, we suppose that we have the event stream illustrated in Figure 3. Figure 4 illustrates how the automaton processes these events. 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 data schema of the instances at a given automaton state. For readability, the timestamp attributes 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 attribute mapping function F_1 to the attributes of e_1 and creates the resulting instance I_1 under state A.

The next event e_2 does not satisfy θ_1, hence the start state does not create a new instance. For instance I_1 at state A, to determine if I_1 can traverse any outgoing edge, predicates θ_2, θ_3 and θ_4 on the outgoing edges of A are evaluated with respect to e_2. This event satisfies θ_3 on the rebind edge associated with state A. Therefore, the event traverses this edge and instance I_1 is updated by the mapping F_3[1]. The result is shown in Table 4.

Event e_3 matches θ_1; therefore a new instance I_2 is created at state A. For I_1, I_1 and e_3 together do not satisfy θ_2, the filter predicate associated with state A, because in I_1 the Name_n attribute

[1]Technically, an automaton instance traversing the rebind edge creates a new instance, and then the original instance will be deleted after processing this event. We use the term "update" here to simplify the presentation.

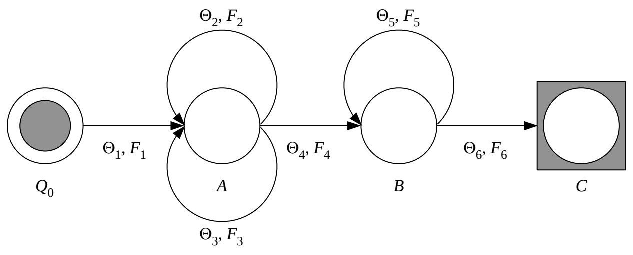

Figure 2: Automaton for Example 9

Event	(Name,Price,Volume)
e_1	$\langle$IBM, 90, 15000$\rangle$ 9:10, 9:10
e_2	$\langle$IBM, 85, 7000$\rangle$ 9:15, 9:15
e_3	$\langle$Dell, 40, 11000$\rangle$ 9:17, 9:17
e_4	$\langle$IBM, 81, 8000$\rangle$ 9:21, 9:21
e_5	$\langle$MSFT, 25, 6000$\rangle$ 9:23, 9:23
e_6	$\langle$IBM, 91, 9000$\rangle$ 9:24, 9:24

Figure 3: Example Event Sequence

Event	Instances at State A (Name_1,Price_1, Name_n, Price_n)	Instances at state B (Name,MaxPrice, MinPrice)	Instances at state C (Name, MaxPrice, MinPrice, FinalPrice)
e_1	$I_1 = \langle$IBM, 90, IBM, 90$\rangle$ 9:10, 9:10		
e_2	$I_1 = \langle$IBM, 90, IBM, 85$\rangle$ 9:10, 9:15		
e_3	$I_1 = \langle$IBM, 90, IBM, 85$\rangle$ 9:10, 9:15 $I_2 = \langle$Dell, 40, IBM, 40$\rangle$ 9:17, 9:17		
e_4	$I_1 = \langle$IBM, 90, IBM, 81$\rangle$ 9:10, 9:21 $I_2 = \langle$Dell, 40, IBM, 40$\rangle$ 9:17, 9:17	$I_3 = \langle$IBM, 90, 81$\rangle$ 9:10, 9:21	
e_5	$I_1 = \langle$IBM, 90, IBM, 81$\rangle$ 9:10, 9:21 $I_2 = \langle$Dell, 40, IBM, 40$\rangle$ 9:17, 9:17	$I_3 = \langle$IBM, 90, 81$\rangle$ 9:10, 9:21	
e_6	$I_2 = \langle$Dell, 40, IBM, 40$\rangle$ 9:17, 9:17		$I_3 = \langle$IBM, 90, 81, 91$\rangle$ 9:10, 9:24

Figure 4: Example computation

has value IBM, while in e_3 the Name attribute has value Dell. I_1 is therefore not affected by this event.

The arrival of e_4 illustrates the non-determinism of the FOLD construct, implemented by state B. e_4 is filtered for I_2 (the Dell pattern). However, for I_1 both θ_3 and θ_4 are satisfied (the duration condition in θ_4 is now true). Hence I_1 non-deterministically traverses both the forward edge from A to B and the rebind edge of state A. In the example, we update to content of I_1 with the content of e_4, and create an new instance I_3 under state B by applying F_3 to I_1 and the current event e_4.

Events e_5 and e_6 are processed similarly. For e_5, each of the instances traverses the corresponding filter edge, and are thus not affected. The interesting aspect of e_6 is its effect on instance I_1. I_1 together with e_6 satisfies the filter predicate associated with state A, but does not satisfy the forward or rebind predicates 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.

3.2 Additional Subtleties

While the previous example gives an good high level understanding of our automata, there are several subtleties that it fails to illustrate. First of all, there might be simultaneous events in a stream, produced either by an external stream source or by a Cayuga query. Simultaneous events can pose several difficulties in ensuring correctness of automaton processing. Consider our example above, but let there be another event $e'_6 = \langle$IBM, 80, 8000$\rangle$ 9:24, 9:24 which has the same detection time as e_6, and is processed by the automaton between e_5 and e_6. Even though this event fails to satisfy θ_5, we cannot delete I_3 immediately, since according to the Cayuga query semantics, the output event produced by I_3 together with e_6 should not be affected by the presence of e'_6. This suggests that after processing the current event e, we cannot delete those automaton instances that did not traverse their associated filter edges immediately, since if any following events are simultaneous with e, these automaton instances should remain visible while processing these events. We handle simultaneous events correctly by using epoch-based query processing, to be described in Section 4.3.

As stated earlier, not every Cayuga query can be implemented by a single automaton. In order to process arbitrary queries, Cayuga supports *resubscription*. Resubscription is similar to pipelining – the output stream from a query is used as the input stream to another query. Because of resubscription, query output must be produced in real time. Each tuple output by a query has the same detection time as the last input event that contributed to it, and thus its processing (by resubscription) must take place in the same epoch in which that event arrived. This decision motivates the Cayuga Priority Queue, described in Section 4.2 and the "pending instance lists" described in Section 4.3.

4. THE CAYUGA SYSTEM

In this section, we describe the Cayuga system architecture, and we explain how we efficiently implement large number of automata.

4.1 Architecture

The Cayuga system architecture is shown in Figure 5. We first describe the control and data flow in Cayuga, and how components interact at a high level. We then focus on describing the major components of the system in more detail.

External events arriving at the Cayuga system are received by Event Receivers (ERs), each of which runs in a separate thread, receiving events from a particular source. The ER threads are responsible for deserializing arriving events, assigning timestamps if necessary, internalizing them in the Cayuga Heap and inserting them on the input Priority Queue (PQ). The ERs and PQ are described in Section 4.2.

The Cayuga Query Engine is a single thread that is responsible for most of the query processing work. The engine dequeues events from the PQ in detection time order. It performs all indi-

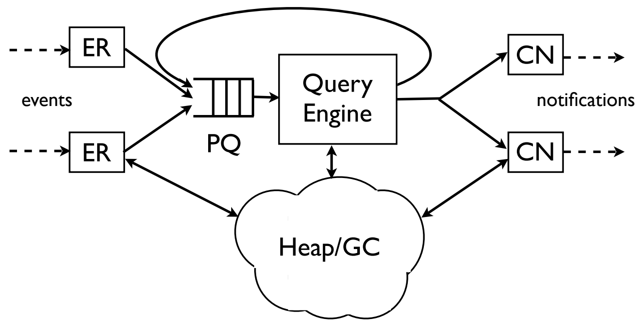

Figure 5: Cayuga System Architecture

cated automaton state transitions, using several index structures to achieve high throughput. For each automaton instance reaching a final state, it enqueues a new event on the PQ if required for resubscription, and passes events to the appropriate Client Notifier threads (CNs). The engine is described in Section 4.3, and CNs are described in Section 4.4.

Cayuga uses a customized Memory Manager with a Garbage Collector (GC) to facilitate efficient object sharing for high performance and a small memory footprint. The Memory Manager is described in Section 4.5.

4.2 Event Receivers and Priority Queue

Cayuga has multiple Event Receiver threads. Each ER thread converts events arriving from an external data source (such as a TCP stream or a sensor device) into a sequence of internalized Cayuga events. Since the external data sources may encode data in different ways, there are multiple ER classes with deserialization methods specific to the data sources.

Internalized events have a common format, designed to allow sharing of complex data such as large string bodies or user-defined types. Shared data resides in the garbage-collected Cayuga Heap, discussed in Section 4.5.1. This design enables us to manipulate events and automaton instances using "shallow copy" operations and exploit efficient block move procedures whenever possible.

Newly-internalized events are inserted on the Priority Queue (PQ) which is ordered by event detection time. The events are later dequeued and processed by the Query Engine, described in Section 4.3.

As mentioned above, the Cayuga query model requires that events be processed in detection time order. Thus, the system must correct for clock skew between data sources, as well as for network delay and reordering. We address this problem as follows. Let T be an *a priori* bound on the out-of-orderedness of the input streams. That is, when an incoming event with timestamp t arrives at the Cayuga system, it is guaranteed that the system's local clock is at most $t + T$. With T so defined, proper ordering can be achieved by buffering events in the PQ until they appear to be at least T time units old before allowing them to be dequeued. Specifically, a dequeue operation will block (or return *empty*) until the smallest detection time of any event in the queue is less than $c - T$, where c is the current Cayuga system clock value.

If an arriving event has a timestamp smaller than the timestamp of some event previously dequeued from the PQ, the event is ignored. Thus, coping with high variance in arrival times requires a large value for T. Note that increasing T adds latency but does not affect throughput.

Our use of a fixed global parameter T could be overly conservative in some cases. A more sophisticated (but less efficient) approach is described in [15].

Some data sources do not provide timestamps. For such data sources, the ER assigns to each arriving event a *point timestamp* – a 0-length interval – with detection time equal to the current Cayuga system clock. Clearly, a stream with such locally-generated timestamps can inter-operate with other streams whose timestamps are provided by the data sources.

Note that if *all* streams have locally-generated timestamps, then a suitable value for the global parameter T is 0. In this case, the PQ behaves as a FIFO queue except for resubscription processing as described in Section 4.3.

4.3 The Cayuga Query Engine

The Cayuga Query Engine processes internalized Cayuga events drawn from the Priority Queue, by executing state transitions of Cayuga automata and generating output events whenever final states are reached. Output events can be fed back into the PQ for resubscription, or they can be passed to CN threads and forwarded to subscribers as described in Section 4.4. In the following subsections we describe the engine in detail.

4.3.1 Query Representation

The external representation of a Cayuga query uses an XML format we call Automaton Intermediate Representation (AIR). The AIR encoding of automaton states and edges is basically straightforward; however, the following features are worthy of note:

- Final states are explicitly identified for subscribing clients and for resubscription.
- Filter edges are explicitly identified, as they require different treatment from FR edges.
- Edge predicates and schema maps (as discussed in Section 3) are encoded as programs for a specialized bytecode interpreter discussed below.
- Predicate conjuncts that should be indexed are flagged appropriately (our indexing strategy is described in Section 4.3.3).

The AIR format is intentionally rather low-level, and can represent arbitrary collections of automaton states. This design decision enhances modularity – the Query Engine deals only with automaton execution, and is not concerned with the translation of algebra expressions into automata, or with multi-query optimizations involving automaton rewriting or sharing of sub-automata by resubscription. The only form of query optimization performed by the Engine is to merge manifestly equivalent states during AIR file loading.

When Cayuga loads an AIR file, the resulting internal data structure has an explicit representation of automaton states and edges. The Cayuga automaton model is nondeterministic. Thus, during query evaluation there may be many active instances of query automata, each in a different state and with different stored data. Our internal representation associates a list of *instance objects* with each automaton state, as depicted in Figure 6(a). An instance object associated with state Q represents an instance of the query automaton in state Q. The stored data of the automaton instance (conforming to the schema of Q) is contained in the instance object.

Edge predicates and schema maps are represented internally by bytecode interpreter programs, essentially copied from their AIR representation. The Cayuga bytecode interpreter is a straightforward stack machine specialized for efficient "short-circuit" evaluation of conjunctions of atomic formulas, and for efficient construction and copying of instance objects. The interpreter (as well as the

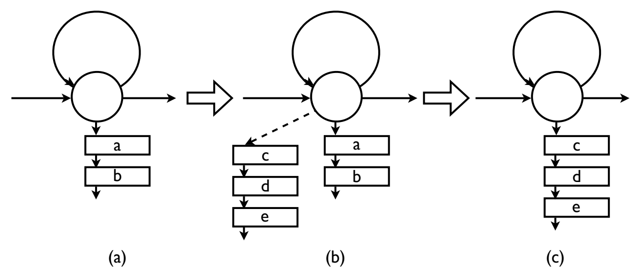

Figure 6: Instance Lists

AIR format) is designed to be extensible, so it can easily support new user-defined predicates (UDFs) and event schemas including new user-defined types (UDTs).

4.3.2 Simultaneity and Epochs

Cayuga's data model allows events with simultaneous detection times. To handle such events correctly, we use an *epoch-based* processing strategy. During epoch t, all events with detection time t are processed, but it is critical that no new automaton instance created during epoch t can be visible to other events detected in the same epoch. To achieve this effect, we put newly-created instances under a state into a separate *pending instance* list. At the end of each epoch, we perform *instance installation*, atomically merging each state's pending instance list with the state's surviving instances. This process is depicted in Figure 6(a-c).

Although instance installation is deferred to epoch boundaries, processing of new events for instances reaching final states is *not* deferred. Client notification, as well as insertion of new events into the PQ to support resubscription, both happen immediately. This scheme is a simple and efficient way to handle simultaneity and resubscription. It also makes it possible to construct some powerful recursive queries (arguably a good thing), or to construct an automaton that can loop forever in an epoch, generating unboundedly many events with the same detection timestamp (a bad thing). Fortunately, automata generated from Cayuga algebra expressions are guaranteed not to exhibit this bad behavior.

4.3.3 Evaluation and Indexing

We can now describe the flow of processing for each incoming event. The system components involved are shown in Figure 7. Our design is specially driven by the invariants of Cayuga automata described in Section 3.

Event processing is done in two stages. In the first stage, the Filter Evaluator is invoked. As its name suggests, the Filter Evaluator evaluates filter edge predicates with respect to the current input event. It identifies the set of *affected* instances and the states they are associated with. (Recall from Section 3 that an instance is *affected* if it does not traverse its filter edge.) All affected instances will be marked for deletion at the end of the current epoch.

In the second processing stage, for each affected state marked by the Filter Evaluator, we invoke the Forward-Rebind Evaluator. The FR Evaluator evaluates all the FR edge predicates of a state, and creates new instances under the destination state of each edge it its predicate is satisfied. Whenever an instance is created under a final state, an output event is generated, which is inserted on the PQ for resubscription and/or sent to the appropriate Client Notifiers.

In our implementation we convert edge predicates to Disjunctive Normal Form and treat the top-level disjuncts separately. Thus, for

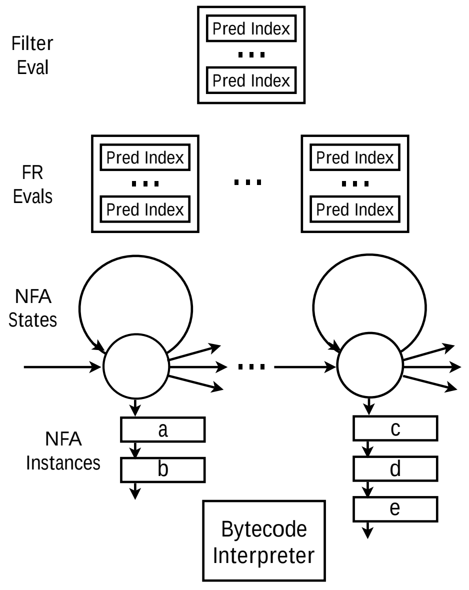

Figure 7: Query Evaluation Architecture

the remainder of this discussion we assume each predicate is a conjunction of atomic predicates. We classify atomic predicates as either *static* or *dynamic*. A static atomic predicate compares an event attribute to a constant. Any other atomic predicate is considered dynamic.

In a straightforward implementation of a Filter Evaluator with no indexing, each input event would need to be checked against the static filter atomic predicates of all the states; then the dynamic filter atomic predicates would need to be checked for all instances present at the nodes whose static filter components were satisfied, thereby identifying the affected instances and nodes. All these predicate evaluations would be performed by the Bytecode Interpreter.

Similarly, in an unindexed implementation of an FR Evaluator each input event would need to be checked against the static atomic predicates of every FR edge of every affected state; then, for those FR edges whose static components were satisfied, the dynamic atomic predicates would need to be checked for all affected instances at the associated node.

Performance is greatly improved by the use of indexing. Our current indexing structure addresses Static and Dynamic equality Atomic predicates. Each Conjunctive Clause is segregated into four components: Indexed Static, Unindexed Static, Indexed Dynamic, and Unindexed Dynamic.

Static predicate indices for filter predicates are maintained at the global level to efficiently identify (a superset of) the set of affected States. Static predicate indices for FR predicates could be maintained at the global level (eliminating duplicate evaluation of indexed static atomic predicates) or on a per-node basis (eliminating unnecessary evaluation for unaffected nodes). Either choices could be viable, depending on workload; we decided in favor of the second. Dynamic predicate indices for filter predicates are maintained on a per-node basis, since they index node instances rather than the global constants in the static filter predicate, and each index is associated with a particular filter predicate clause. We currently maintain only one dynamic predicate index per node to avoid excessive maintenance overhead as instances are added and deleted at the end of each epoch.

Upon event arrival, the following process is carried out for filter predicate evaluation

1. The filter evaluator probes the relevant indices to enumerate which nodes have the static indexed component of one of their filter predicate clauses satisfied. (The clause number is also output). The static unindexed component is evaluated for these clauses, and if is is satisfied we move to step 2.

2. Once the node and clause numbers have been identified, the corresponding dynamic filter predicate index is probed to get the set of instances that satisfy the indexed dynamic component of that clause.

3. For each such instance, the unindexed dynamic component is evaluated by the interpreter to output the set of affected instances at that node.

For Nodes that do not have filter predicate indexing enabled, the index output degenerates to a sequential list consisting of all instances present at the node, unindexed predicate evaluation then proceeds on this list. The process ultimately creates a list of affected nodes, each containing a list of affected instances of its own. The FREvaluator then probes the FR indices associated with each affected node to output the FR Edges having the static component of one of their clauses satisfied (the clause number is also output). Dynamic FR indices are currently not supported in our system. Since the number of affected nodes/instances is generally small, we believe the performance advantage would be overcome by the maintenance overhead. The unindexed component is evaluated on the node's affected instances to finally identify (edge, instance) pairs. New instances are created and added to pending instance lists to effect the state transition.

4.4 Client Notifiers

Client Notifier threads (CNs) are roughly analogous to ERs: ERs receive events from external streams, while CNs send event notifications to subscribing clients. There is one CN per connected Cayuga client. Whenever the Query Engine detects that a query has produced a match (i.e., an automaton instance reaches a final state), the QE sends the corresponding event to all the CNs representing the subscribing clients for that event. Each CN serializes the event appropriately and delivers it to the client.

4.5 Memory Management

4.5.1 Garbage Collector

During Cayuga event processing, large string bodies and objects of complex user-defined data types are created, copied and destroyed frequently. To reduce the space and time overhead for such objects, we share the objects as much as possible and rely on a custom memory management scheme to reclaim them. In this section we describe the memory manager and discuss some of our design decisions.

The principle dynamic data structures in Cayuga are associated with events and automaton instances. Here we discuss the treatment of automaton instances; the treatment of events is similar, and in fact most of the code is shared.

At the top level, automaton instances are allocated and freed manually. Scalar data resides in each instance, while string bodies and complex data structures reside in a garbage-collected heap and are always shared among instances. Note this implies that the size of an instance is determined by the schema of the corresponding automaton state, and instances are fairly small. Copying an instance is done by "shallow copy" – only the fixed-length instance object is copied, increasing the degree of sharing of the sub-objects.

To manage instance objects, we use a simple size-segregated collection of free pools.

To manage complex shared objects, we initially considered a C++ "smart pointer" implementation based on reference counts. Apart from reference counting's inability to deal with cyclic structures – a restriction we could probably have lived with for our application – this approach has considerable elegance. However, Cayuga's nondeterministic state transitions require frequent copying and deletion of instances, each involving reference count manipulation and possibly synchronization overhead as well. So we implemented a garbage-collected heap for shared objects. This allows us to do a shallow copy of an instance with no locking or reference count manipulation – a simple and efficient block move operation is safe.

Our GC design uses some familiar techniques in a somewhat unusual combination driven by the needs of our application. Discussion of most of these techniques can be found in [17].

First, we remark that in the absence of pathologic query selectivities Cayuga's object lifetime distribution is highly bimodal. Most automaton instances fail filter predicates and die very early. A few instances, such as those associated with a long-running μ operator, live for a very long time. But hardly any instances have "medium" lifetimes. Because of this observation, we chose a simple generational scheme with two generations [4]. The first generation uses copying garbage collection [3]; objects that survive more than a few copies are "promoted" to the second generation, which uses non-copying collection.

Although copying collection is somewhat controversial [5, 14], Cayuga's object lifetimes are sufficiently short and skewed that we believe copying should have a considerable performance advantage. With a copying GC, object allocation is essentially free – basically, just incrementing a limit pointer – and the cost of a collection is linear in the number of bytes of live data, but is *independent* of the number of bytes reclaimed. Thus, if the vast majority of allocated objects do not survive until the next garbage collection, the total cost of allocation using a copying collector can be lower than the cost of manual memory management using malloc() and free() [3].

To avoid the need to update all client reference variables when an object is copied, we use a handle-based design similar to some Java VMs. Our handle space data structure supports unit-cost reclamation of the entire set of dead handles during a reclamation, preserving the asymptotic cost of the copying collection algorithm.

We now briefly discuss issues of root finding and concurrency. Logically, the first step of a garbage collection is to find the values of all "root variables" – program variables that might contain references to objects in the heap. The collector then traces through the heap, identifying all objects that are transitively reachable from any of the roots. In a multi-threaded system like Cayuga, the GC must either stop all other threads (a "stop-the-world" collector) or be prepared to operate correctly in the presence of concurrent changes to the heap made by other threads.

In general, these tasks represent considerable development effort. Precise root finding requires either (non-portable) compiler support or (error-prone) application support. Stopping the world is also non-portable, and is undesirable in any case; while concurrent collection is extremely delicate and error-prone [refs].

We address these issues by exploiting our application design, in which the majority of work is done by a single Query Engine thread. Most GC systems collect as a side-effect of object allocation. The Cayuga GC collects only when explicitly invoked by the

Engine thread. This happens only at "GC-safe" points in the Engine code[2]. At such points, all of the Engine's live data is guaranteed to be reachable from (1) events waiting to be processed in the PQ, or (2) automaton instances contained in the query automata data structure. These are the only Engine root variables that are needed, and enumerating them requires only a few lines of code.

Of course, this approach guarantees only that the Engine thread is at a GC-safe point when a collection occurs. What about the ER and CN threads? We guarantee that *all* points in such threads are GC-safe by requiring them to access the heap using a stylized API similar to a Java "native methods" API. This API is somewhat less convenient than the direct API used in Engine code; but use of the heap by the ER and CN threads is simple enough that the approach is tolerable.

Finally, note that an allocation request may ask for more space than is currently available in the heap. The usual response in this situation is to invoke garbage collection, and to fail only if the collection does not produce sufficient free space. However, we have explicitly ruled out invoking garbage collection as a side effect of an allocation request. Fortunately, this is not a serious issue in our two-generation design. When insufficient memory is available in the first-generation heap to satisfy a request, we immediately "promote" the request to the second generation, which uses the system allocator and does not fail. [3]

4.5.2 Internal String Table

The Internal String Table is a component that manages read-only string objects stored (internalized) in the Cayuga Heap. The purpose of the Internal String Table is to ensure that there is at most one copy of any string value in the heap – if multiple clients internalize the same string value, the system returns identical references to the same shared string body in the heap. This has two beneficial effects:

- It reduces space consumption. If the same string appears in multiple input events or automaton instances, storage will be shared even if the events or instances were constructed independently.

- It "canonicalizes" the strings, enabling equality test to be performed by a single pointer comparison rather than byte-by-byte character comparison. For certain workloads, in which successful string comparisons occur frequently, this can yield significant speedups.

The "obvious" implementation for the Internal String Table is to place all internalized strings in a hash table, eliminating duplicates. Unfortunately, this implementation does not work, because it makes no provision for ever reclaiming an internalized string object s even if no client holds a reference to it – the Internal String Table *itself* holds a reference to s that keeps it from being reclaimed. Thus, the Internal String Table can grow without bound even if Cayuga's space requirements *at any particular time* are modest.

A Canonical String Table like ours is not a new idea; solutions to the unbounded growth problem typically involve some form of "weak references," a GC feature which, while still slightly arcane, dates back to the 1980's and is sufficiently mainstream to exist in Java and C-#. Informally, a weak reference to an object enables a client to use that object, but does not prevent the object from being reclaimed. The Cayuga GC provides a very simple form of weak reference: when all ordinary references to an object disappear, the object is reclaimed, and all remaining weak references to the object are atomically cleared to **null**. With this feature, we can easily implement the Internal String Table as a hash table whose buckets contain weak references to canonical string objects; code in the hash table maintenance methods lazily deletes **null** references from the table.

[2]In fact, Cayuga contains only one call to the collector, at the top of the Query Engine's main loop. Whether the call actually does a collection, or just returns immediately, is a policy decision internal to the GC.

[3]More precisely, it does not fail recoverably.

5. RELATED WORK

There is a lot of previous work relevant to complex event monitoring. Due to space constraints, we discuss only a representative subset here. The work falls into three broad classes:

Publish-subscribe systems such as [2, 9] are characterized by very limited query languages, allowing simple selection predicates applied to individual events in a data stream. Such systems trade expressiveness for performance – when well engineered, they exhibit very high scalability in both the number of queries and the stream rate. However, their inability to express queries that span multiple input events makes them unsuitable for complex event processing.

Stream databases such as [13, 7, 12, 6] lie at the opposite end of the spectrum, trading performance for expressiveness. Such systems have very powerful query languages, typically subsuming SQL with provisions for sliding windows stream grouping features. Though powerful, these query languages can be awkward for expressing the kinds of sequential patterns that occur frequently in our target applications. Moreover, the systems have yet to demonstrate the scalability of the other approaches.

Complex event systems such as SNOOP [1], ODE [11] and SASE [18] are closest in spirit to our own work. These systems describe composite events in a formalism related to regular expressions and use some variant of a NFA model. While many support some form of parameterized composite events, none is as general as Cayuga. In addition, the semantics of some of the more expressive event systems is not well defined [10, 19].

6. OUTLOOK

We believe that Cayuga incorporates interesting novel design decisions that will have impact on the community of researchers and practitioners that are currently working on building complex event systems. At Cornell, we currently have three ongoing deployments of Cayuga. We are collaborating with researchers from the Cornell Parker Center for Investment Research on real-time analysis and correlation of external data streams and stock data streams. We are also working on an integration with CorONA, a distributed high-performance publish-subscribe system for Web MicroNews running on PlanetLab (http://www.cs.cornell.edu/People/egs/beehive/corona/). Our vision is to devise an infrastructure for stateful monitoring of the Blogosphere. Our third deployment is an ongoing collaboration with system administrators at CTC, Cornell's high-performance computing center, on distributed infrastructure monitoring through OS event log processing.

While our work here describes a working system that is currently under deployment, our users have started to demand other features. We are currently adding support for user defined types and functions in Cayuga; this also allows us to extend Cayuga to XML for deployment in service-oriented architectures. For an even more scalable architecture, we want to distribute the event processing task by setting up a hierarchy of Cayuga servers, where servers at higher levels resubscribe to the output of servers at lower levels.

7. ACKNOWLEDGEMENTS

We wish to acknowledge the many contributions of our colleague Mingsheng Hong, whose work has been crucial for all aspects of Cayuga. Only the CIDR conference rule prohibiting multiple submissions prevented him from being included among the authors of this paper.

This work was supported by the National Science Foundation under Grants IIS-0636259, IIS-0621438, and CRCD-0203449; by AFOSR under grant FA9550-05-1-0055; by a Sloan Foundation Fellowship; and by a gift from Xerox. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

8. REFERENCES

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

[2] 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.

[3] Andrew W. Appel. Garbage collection can be faster than stack allocation. *Information Processing Letters*, 25(4):275–279, 1987.

[4] Andrew W. Appel. Simple generational garbage collection and fast allocation. *Software Practice and Experience*, 19(2):171–183, 1989.

[5] Hans Boehm. Mark-sweep vs. copying collection and asymptotic complexity. ftp://parcftp.xerox.com/pub/gc/complexity.html.

[6] 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.

[7] 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.

[8] A. Demers, J. Gehrke, M. Hong, M. Riedewald, and W. White. Towards expressive publish/subscribe systems. In *Proc. EDBT*, 2006.

[9] 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.

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

[11] 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.

[12] 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.

[13] P. Seshadri, M. Livny, and R. Ramakrishnan. Sequence query processing. In *Proc. SIGMOD*, pages 430–441, 1994.

[14] Frederick Smith and Greg Morrisett. Comparing mostly-copying and mark-sweep conservative collection. In *ISMM '98: Proceedings of the 1st international symposium on Memory management*, pages 68–78, New York, NY, USA, 1998. ACM Press.

[15] U Srivastava and J. Widom. Flexible time management in data stream systems. In *Proc. PODS*, pages 263–274, 2004.

[16] W. White, M. Riedewald, J. Gehrke, and A. Demers. What's "next"? Technical Report TR2006-2033, Cornell University, 2006.

[17] Paul R. Wilson. Uniprocessor garbage collection techniques. In *Proc. Int. Workshop on Memory Management*, Saint-Malo (France), 1992. Springer-Verlag.

[18] E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *Proc. SIGMOD*, 2006.

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