# Cayuga: A High-Performance Event Processing Engine*

[Demonstration Paper]

Lars Brenna†, Alan Demers, Johannes Gehrke, Mingsheng Hong, Joel Ossher,
Biswanath Panda, Mirek Riedewald, Mohit Thatte, Walker White
Cornell University
Ithaca, New York
{larsb,ademers,johannes,mshong,jpo5,bpanda,mirek,mmt38,wmwhite}@cs.cornell.edu

## ABSTRACT

We propose a demonstration of Cayuga, a complex event monitoring system for high speed data streams. Our demonstration will show Cayuga applied to monitoring Web feeds; the demo will illustrate the expressiveness of the Cayuga query language, the scalability of its query processing engine to high stream rates, and a visualization of the internals of the query processing engine.

## Categories and Subject Descriptors

H.2 [**DATABASE MANAGEMENT**]: Query processing

## General Terms

Experimentation, Design, Performance

## Keywords

Publish/Subscribe, Complex Event Processing, Continuous Query Processing

## 1. INTRODUCTION

An ever increasing amount of data arrives as high speed event streams. Stock ticker data, network traffic data, data gathered in sensor networks, RFID streams are just a few examples of such streams. In addition, as the amount of dynamic content on the Web (blogs, news, auctions) grows rapidly, the web is becoming a source of many interesting and complex event streams.

An emerging class of enterprise applications such as healthcare, environmental monitoring, supply chain management, and compliance checking, as well as consumer applications such as personalized news feeds and information aggregation, all require filtering, correlation, and aggregation of individual events. The resulting complex event queries go beyond publish/subscribe in that the queries involve correlations among multiple events instead of simple predicates on individual events.

At Cornell, we have built the Cayuga System, a high-performance system for complex event processing [1, 2]. Cayuga combines a simple query language for composing stateful queries with a scalable query processing engine based on nondeterministic finite state automata with buffers. An important feature of Cayuga is its ability to scale not only with the arrival rate of events in the stream, but also with the number of queries; the latter dimension of scalability is especially important for consumer applications where millions of users could have queries registered with the system.

In this demo, we show an application of Cayuga to monitor event streams from a web feed aggregator for personalized news aggregation and notification. We first give a high-level overview of the Cayuga System, introducing some of the Cayuga queries that we will show in the demo (Section 2). We then describe the actual demonstration in detail (Section 3).

## 2. CAYUGA

In Cayuga, each event stream has a fixed relational schema, and events in the stream are treated as relational tuples. Each event has two timestamps, a start time and a detection time, modeling the fact that events can have a non-zero but finite duration. Events are serialized in the order of their detection times; events with the same detection time are considered to happen simultaneously, and there is no guarantee about their serialization order. Several interesting theoretical issues have led to this design [3].

The Cayuga query language is derived from an event algebra [1]; it is a simple mapping of the algebra operators into a SQL like syntax, similar in spirit to the complex event language in SASE [4]. Each query has the following form:

SELECT $\langle attributes \rangle$
FROM $\langle algebra\_expression \rangle$
PUBLISH $\langle output\_stream \rangle$

*This material is based upon work supported by the National Science Foundation under Grant 0621438 and by the Air Force under Grant AFOSR FA9550-06-1-0111. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

†The author is also affiliated with the University of Tromsø, Norway, but this work was done while visiting Cornell University.


*SIGMOD'07,* June 11–14, 2007, Beijing, China

| | |
|---|---|
| item_url | http://news.zdnet.com/2509-1_22-0-... |
| feed_url | http://news.zdnet.com/2100-1040_22... |
| title | Louisiana joins battle over violent video g... |
| summary | Similar laws in other states found unconst... |
| timestamp | Mon, 19 Jun 2006 13:45:00 PDT |
| streamid | webfeeds |
| starttime | 0 |
| endtime | 0 |

**Table 1: Event Schema**

The SELECT clause specifies the attributes in the output stream schema, the FROM clause specifies a Cayuga event pattern, and the PUBLISH clause gives the output stream a name. The event pattern can be built with three different operators. The FILTER{$\theta$} operator selects those events from the input stream that satisfy the predicate $\theta$.

**Demo Query 1:** In the demo, we will illustrate the FILTER operator with the following query that finds all news items published by Google news in the overall stream "webfeeds" with a schema that is shown in Table 1.

```
SELECT * FROM
  FILTER {feed_url='http://news.google.com/'}(webfeeds)
PUBLISH google_news_items
```

A powerful construct that allows us to correlate events over time is the sequencing operator NEXT{$\theta$}. When applied to two input streams $S_1$ and $S_2$ as $S_1$ NEXT{$\theta$} $S_2$ the operator combines each event from $S_1$ with the next event in $S_2$ that satisfies the predicate $\theta$ and occurs after the detection time of the event in $S_1$.

**Demo Query 2.** This query finds all news items that are published by some site, followed by an article on Google news that refers to it. Note that this query takes the output of Query 1 as one of its input streams. The user-defined function contains performs substring matching; $1 and $2 refer to the two input streams of the NEXT operator.

```
SELECT $2.summary, $1.item_url FROM
  (webfeeds) NEXT {contains($2.item_url,$1.item_url)=1}
  (google_news_items)
PUBLISH reffed_by_google_news
```

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 operator. Intuitively, FOLD is a generalization of the NEXT operator because it looks for patterns comprising two or more events. We will describe FOLD using the next example.

**Demo Query 3.** This query sends out a notification whenever an iPod is popular in the news (i.e., there are at least ten articles talking about the iPod in a certain time duration). The FOLD operator contains three expressions: (1)condition for the iteration (2)stopping condition for iteration (3)mapping between iteration steps.

```
SELECT * FROM
  (SELECT *, cnt AS 1 FROM
    FILTER {contains(summary,"iPod")=1}(webfeeds))
      FOLD {TRUE, cnt>10 AND dur<1 day, cnt AS cnt+1}
    (SELECT * FROM
      FILTER {contains(summary,"iPod")=1}(webfeeds))
PUBLISH ipod_popularity
```

Cayuga processes events in epochs; during an epoch all events with the same detection time are processed. The queries are implemented as nondeterministic finite state automata with buffers and self-loops that work as follows. Each state in an automaton is assigned a fixed relational schema. Each edge, say between states $P$ and $Q$, is labeled by a triple $\langle S, \theta, f \rangle$, where $S$ identifies an input stream; $\theta$ is a predicate over schema($P$) $\times$ schema($S$); and $f$, the *schema map*, is a function taking schema($P$) $\times$ schema($S$) into schema($Q$). The NFAs operate as follows. Suppose an NFA instance is in state $P$ with stored data $x$ (note $x$ conforms to 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 $f(x, e)$. Cayuga supports *resubscription*, a concept similar to a query plan in a relational database system: the output event stream from one query can be used as the input stream to one or more other queries. Resubscription enables very complex event pattern queries, and it significantly extends the expressiveness of the query language. More details of the system can be found in our recent publications [1, 2].

# 3. DEMONSTRATION OUTLINE

In this section we provide an overview of how we demo the queries shown above, taking Demo Query 2 as the running example. Our demo has the following three components: (i) A Web based front end for users to enter queries; (ii) a trace visualizer which displays the internal state of the Cayuga engine and the automata based query representation; and (iii) an output visualizer which displays the results of query evaluation. During the demonstration, we will run queries on a stream of events from a Web feed aggregator. Let us now present each of the above components by walking through the execution of Query 2. This query was processed against a stream of events containing 225365 feed items from 418 channels recorded between June and October 2006.

## 3.1 Submitting a Cayuga Query

The Web-based frontend is running on a custom Python Web server, with AJAX-based controls for asynchronous communication and user-friendly interfaces in the browser. Cayuga has a web-based frontend where users can enter persistent queries and register them with a running Cayuga engine. For convenience, users are given a choice of predefined templates for the described query language from a drop-down menu. Then, they can modify these to their need, or write one from scratch. Since Cayuga queries can easily span multiple lines, we supply a multi-line input box bigger than typically seen in web-based query interfaces. The query selected for editing in Figure 1 is a screenshot of our query frontend which displays Query 2. Note that this version of the query is inlined and without resubscription.

Upon submitting a query, the query will be compiled and buffered outside Cayuga. Users will get a message back with a query id or an error message that the query did not compile and must be re-submitted after changes. Users can choose to batch up a number of queries before they press EXECUTE to register their queries with Cayuga. A batch can be cleared by pressing Clear.

## 3.2 Trace Visualizer

Cayuga can output a continuous trace of how its internal state changes between events. This trace is written to a file, which contains a description of the query automata, followed by events and state instances. Our visualizer reads the trace

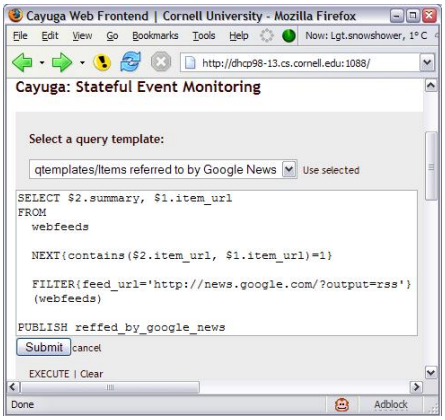


**Figure 1: Web Frontend initialized with Query 2.**

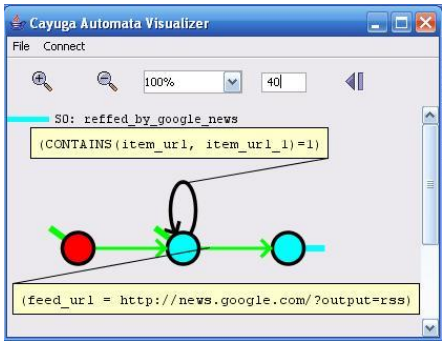


**Figure 2: An empty automaton, annotated with predicates.**

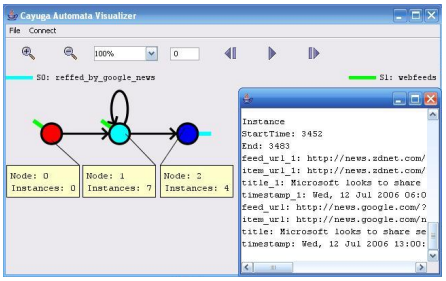


**Figure 3: Automaton with populated nodes.**

file and uses a Java Swing-based GUI to display how events are matched to predicates in the query automata.

To see precisely how the loaded queries work against the events in the input stream, users can choose to play forward, pause at any given point, or go through the trace step by step. This allows users to investigate in detail how any single event affects the internal states of the query automata. The controls seen on the toolbar of the Visualizer in Figures 2 and 3 are (in order from left to right) to zoom the automaton in and out, adjust the vertical distance from the toolbar, and for playback. The input- and output streams are named and given different colors in the animation. At any time, if the user clicks at one of the states in the automaton, a text box will pop up to show the instances in this state. Clicking on an edge will show the predicate on that edge.

Figure 2 shows the automaton for Query 2 with the transition edges annotated with their predicates. In Figure 3, the automaton has processed some events. The stream has so far produced 7 instances of state 1, and 4 instances of state 2. The text box on the right shows the content of state 2, scrolled down to see the last instance to reach this state. From the text we can see that a story published by `http://news.zdnet.com` on Wed, 12 Jul 06:00 was linked to by `http://news.google.com` at 13:00 the same day.

## 3.3 Witness Output

When witness events (i.e., query results) are produced, Cayuga currently dumps them in a file that the Web frontend is tailing. These are then put in a buffer where a frequently polling AJAX function can get them and display in the HTML page on the web browsers of end users, sorted by queries. Figure 4 is a screenshot showing the first two witnesses that appeared for Query 2.

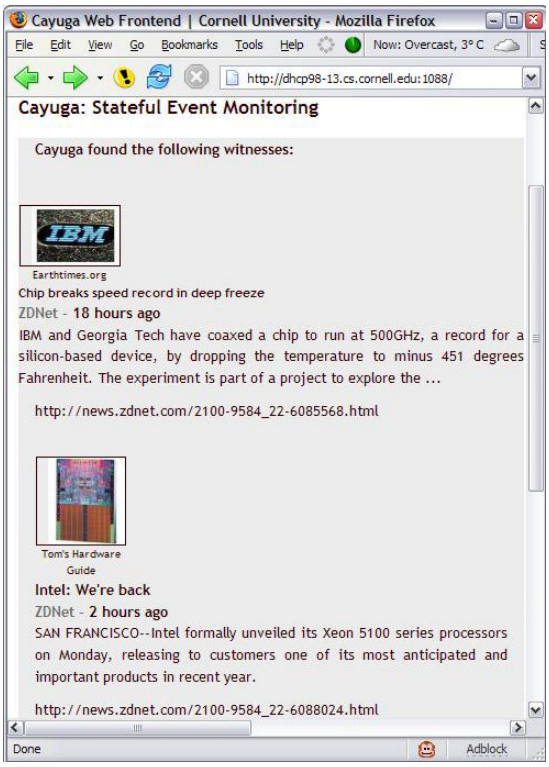


**Figure 4: Web Frontend with witness events.**

# 4. CONCLUSIONS

Cayuga is a mature event processing system with an interesting processing model. We believe a demo of its capabilities will be of interest to the database community.

# 5. REFERENCES

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

[2] A. Demers, J. Gehrke, B. Panda, M. Riedewald, V. Sharma, and W. White. Cayuga: A general purpose event monitoring system. In *Proc. CIDR*, pages 412–422. www.cidrdb.org, 2007.

[3] W. White, M. Riedewald, J. Gehrke, and A. Demers. What is "next" in event processing? In *Proc. PODS*, 2007.

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