Unification of Publish/Subscribe Systems and Stream Databases

The Impact on Complex Event Processing

Joseph Sventek and Alexandros Koliousis

School of Computing Science, University of Glasgow

Abstract. There is increasing demand for complex event processing of ever-expanding volumes of data in an ever-growing number of application domains. Traditional complex event processing technologies, based upon stream database management systems or publish/subscribe systems, are adept at handling many of these technologies. A growing number of hybrid complex event detection scenarios require features of both technologies.
This paper describes a unification of publish/subscribe and stream database concepts to tackle all complex event processing scenarios, with particular emphasis upon hybrid scenarios. The paper describes the architecture for this unified system, the automaton programming language that it supports, and the run-time system that animates automata. Several examples of automata that exploit the system's unified nature are discussed. Raw automata performance is characterised, and its relative performance against Cayuga with respect to stock trend analysis is presented.

Keywords: complex event processing, user-defined functions, stream, automata, publish/subscribe, cache

1 Introduction

There is increasing demand for complex event processing of ever-expanding volumes of data in an ever-growing number of application domains. This explosive growth is fueled by a number of trends in the industry: the availability of inexpensive wireless sensor nodes, the rapid penetration of smart phones in the mobile telephony market, and the growth in availability and sophistication of cloud computing resources. The data deluge resulting from the convergence of these trends dictates that we develop ever more functional and performant complex event processing systems in order to mine the data for information of business or personal importance.

Complex event processing is traditionally achieved using two different technologies: stream database management systems, in which one is able to look backward in time via select statements, and publish/subscribe systems, in which one is able to look forward in time via subscriptions to notifications. Some event

processing scenarios naturally fall into one or the other of these categories; increasingly, there are a number of hybrid scenarios in which both capabilities are required - i.e. the ability to process received notifications is dependent upon access to global and local state representing historical and/or policy information that is crucial to the correct processing of the data.

This paper describes a unification of publish/subscribe and stream database concepts to address these hybrid scenarios. At the same time, the resulting system should also handle scenarios for which the unified nature is not required. The keystone of this unified system is a topic-based, publish/subscribe cache. Topics are organised in memory as either append-only stream tables or static relational tables. Ad hoc select queries, enhanced with time windows, can be presented to the cache at any time. In imperative programming language (the Glasgow Automaton Programming Language, GAPL) is used to program automata to detect complex event patterns over the cached streams and relations. When registered against the cache, each automaton subscribes to chosen topics and receives each event inserted into those topics through the publish/subscribe infrastructure for further processing. Automata can also access (modify) the relational tables, publish new tuples into stream tables, and send events to external processes.

The remainder of the paper is as follows: firstly, we describe related work to place our system in context. This is followed by a discussion of the cache architecture, the automaton programming language, and the automaton execution model. Section 6 evaluates the performance of the system from a number of perspectives, concluding with a performance comparison against Cayuga for several, relevant stock analysis queries. The paper concludes with a discussion of the impact that such a unified system has on future complex event processing and future work.

2 Related work

Codd's relational model structures data into a mathematical object, a relational database, where new information can be extracted using algebraic operators such as projection, selection, union, or join [2]. The ordering of columns (attributes) and rows (tuples) in a relational database is immaterial. On the other hand, data streams are modelled as append-only databases supporting continuous queries for which the relative temporal ordering of tuples is significant.

```
Set  τ = ∞
FOREVER DO
   S = {select attribute from Table [since τ]}
   τ = arg min_{j∈S} t_j
   Return results to user
   Sleep for t seconds
ENDLOOP
```

Fig. 1. The continuous query execution model [1]

```
subscribe event to Topic;
subscribe x to Timer;
window w;
initialization {
   w = Window(sequence, SECS, t);
}
behavior {
   if (currentTopic() == 'Topic')
      append(w, Sequence(event.attribute);
   else
   if (currentTopic() == 'Timer') {
      send(w);
      w = Window(sequence, SECS, t);
   }
}
```

Fig. 2. The continuous query execution model as an automaton

The continuous semantics of queries were first defined in Tapestry [1]. Roughly speaking, a continuous query is a monotonic query – or, equivalently, a non-blocking query [3] – that yields incremental results over a sliding time window whose duration is defined by the current execution time and the last timestamp observed in the previous result set. Figure 1 shows a variant of the basic continuous query execution model, as proposed in [1]. This model inspired the first generation of interactions with our cache, where continuous queries over network flow streams were used to produce real-time visualisations of home networking traffic [4]. Figure 2 is an equivalent implementation in GAPL.

Since TQL, Tapestry's query language, numerous variants of SQL, the language of Codd's relations, have been introduced in the literature, capturing those continuous semantics. CQL, for instance, the continuous query language of the STREAM data management system [5], provides users with a comprehensive list of time-, or count-based sliding window operators to express non-monotonic relations over stream attributes – in other words, stateful relations. Thus, it became apparent to us that the use of sliding windows in stream processing is two-fold. Apart from producing incremental results, sliding windows are also used to maintain the intermediate state necessary for order-agnostic operators – mainly, aggregation and join.

Closely related to this work are user-defined aggregate functions, e.g. like those provisioned in Aurora's SQuAl [6]. A user-defined aggregate function consists of three parts: an initialization function that defines (local) state, opening a window within which the computation takes place; an iteration function that updates state; and a termination function that returns state, when the window closes. User-defined aggregates have been proven to be a sufficient extension to SQL for modeling complex patterns over data streams as finite state machines [3].

At this point, two further analogies can be drawn between user-defined functions and our automata. First, an automaton can not only update local state, but also append tuples to other streams, locally (via publish) or remotely (via send). Second, in contrast to other stream database languages, state need not neces-

sarily be local: using associations, an automaton can modify relational tables, whose current state is immediately available to the rest of the system.

Non-deterministic finite automata, a model used in the pattern languages of Cayuga[7, 8] and SASE [9–11], further extend the notion of user-defined aggregates by expressing complex patterns as composites of ordered sequences of events. The FOLD operator of Cayuga, for example, iterates over an *a priori* unknown sequence of events until a terminating predicate is satisfied, maintaining aggregate statistics in the process; SASE's skip until next (any) match operator maintains intermediate state in arrays in order to express Kleene closures, an operator that has recently received considerable attention in complex event detection [12, 13].

For non-deterministic finite automata, the complexity of a pattern lies in determining what comes "next" in event processing [14]. But apart from folding (or skipping) events, it is hard to specify patterns with branching in a Cayuga or SASE automaton. Indeed, a stream has to be replicated and each branch of the pattern must be represented as a different automaton. Finally, it is not always possible to express nested patterns, e.g. a pattern that uses the local state maintained by another.

The Tribeca query language used for network traffic streams [15] is provisioned for demultiplexing and multiplexing packet streams. The demultiplexing of streams, while similar to group by SQL operators, enables processing of substreams beyond mere aggregates. The importance of multiplexing to network analysts has also been stressed in Gigascope, in their discussion of the merge operator.

Gigascope is a high-performance, network monitoring tool [16], and has been designed to process packets on high-speed links, using a two-level query architecture. High performance is achieved by pushing logic (low-level queries) closer to the packet source - i.e. the network interface. Early data reduction is achieved by protocol filtering, projection and aggregation. High-level queries are then used to perform more complex tasks.

A wealth of packet stream processing algorithms focus on the frequency of certain flow attributes, e.g. the throughput to (from) an IP address or a transport port. The problem has been formally characterized as mining the frequent items in data streams [17], and its applications to network monitoring include finding the heaviest bandwidth consumers (heavy-hitters), or finding the heaviest connection initiators (super-spreaders) [18]. These algorithms are expressible in the Homework system.

Modern traffic analysis tools also query the implicit structure of flows in a traffic mix, in an attempt to match application labels to the underlying flow patterns, and vice versa. Indeed, flow monitoring queries are not just mere counters of some traffic volume metric, e.g. of the number of bytes or packets. Besides identifying frequent items, patterns of temporary correlated flows are used for classification or intrusion detection and are usually expressed as sequences of events [19].

3 The Topic-based Publish/Subscribe Cache

There are many situations in which detection of interesting events requires the ability to receive raw events as they occur and the ability to query, as well as modify, global state. In many deployment scenarios, these actions need to be done in real-time. This paper describes a topic-based publish/subscribe cache to facilitate such real-time processing.

A working system consists of a centralised, topic-based, publish/subscribe cache and a varying number of applications that use the cache; the applications and the cache interact through an RPC mechanism. The cache supports the usual SQL commands for creating tables and inserting tuples into tables. The selection operator has been augmented with appropriate time window extensions to reflect the continuous nature of the events.

One can distinguish three different roles that applications can assume with respect to the cache:

- populate tables with raw events via insert commands
- retrieve data from tables via select commands
- register interest to be notified when complex event patterns are detected

There are two types of tables, ephemeral tables (append-only streams for which the primary key is the time of insertion) and persistent tables (time-varying relations for which the primary key is the first defined field of the table schema). Tuples inserted into ephemeral tables are stored in a circular memory buffer,[1] while tuples inserted into persistent tables are stored in the heap. For persistent tables, an `on duplicate key update` modifier to the insert SQL command is used to update, rather than append, a row in the heap, while maintaining the temporal order of events. Thus, when retrieving tuples from the Cache, the default order for either table type is the time of insertion, unless overridden by an `order by` modifier.

Apart from typical `order by` and `group by` operators, *ad hoc* select queries over cached streams have been augmented with time interval expressions that narrow the scope of results to the most recent ones, e.g. `select * from <table> since` τ, with τ being the timestamp of the last retrieved tuple. Typically, applications submit such queries periodically.

Each table supported by the cache corresponds to a publish/subscribe topic with the same name. An application can register an automaton (a user-defined function) against the database; when a tuple is inserted into a table, that event is published in the associated topic; all automata that have subscribed to that topic will receive that event for processing. If, while processing an event, an automaton determines that it has detected a complex event pattern of interest, it may send information about the complex event to the application that registered the automaton; additionally, during normal processing of events, an automaton may publish a tuple into another event stream.

[1] This is the reason that the component is called the Cache.

The unification of the stream database view of events with a publish/subscribe infrastructure is achieved as follows. Every stream database table, whether ephemeral or persistent, also defines a publish/subscribe topic. Whenever a tuple is inserted into a table, this tuple is delivered as an event to any automata that have subscribed to that topic. A typical reaction application (e.g. a policy management engine) registers one or more automata with the Cache.

An automaton is a compiled program that is bound to a separate thread within the Cache. Its thread is awakened and executed whenever an event is delivered into a topic of interest. As part of its processing, an automaton can send derived events to its registering reaction application, as well as insert tuples into other tables, whether ephemeral or persistent. This unity allows for complex patterns to be presented as materialised views in the stream database and, vice versa, materialised views to be used to derive complex patterns.

4 The automata programming language

4.1 Language design principles

Support for complex event matching, requiring consumption of raw publish/subscribe events and access to static relations, dictates the following features of the automaton programming language:

- ability to subscribe to one or more topic streams over which raw events are conveyed
- ability to query persistent tables to access and/or modify static, global relations
- ability to store local state across event deliveries to the automaton
- ability to send information about complex event occurrences to the registering application
- ability to publish raw events into other publish/subscribe topics

Additionally, one requires a small set of basic data types, a small set of aggregate data structuring constructs, and a small set of control constructs. The basic data types of the language are described in Table 1. HAPL also defines aggregate types (e.g. a map) and additional types required to manipulate these aggregate types (e.g. an iterator over a map). These types are described in Table 2. Every aggregate type has a constructor. Note that a sequence instance can contain heterogeneous basic type instances, while each map or window aggregate instance is bound to a particular type (basic or aggregate).

The language supports `if-then-else` and `while` constructs. It also supports a typical set of operators for arithmetic, conditional expressions, and assignment. The C-like syntax for the language was chosen to facilitate the coding of commonly-used algorithms. See §6.4 for an example.

4.2 General Form for an automaton

The general form for an automaton is:

Table 1. Description of basic data types

Type	*Description*
`int`	64-bit integer
`real`	Double-precision floating point
`tstamp`	64-bit unsigned integer (ns since the epoch)
`bool`	True or false
`string`	Variable-length UTF8 array

Table 2. Description of aggregate and supporting data types

Type	*Description*
`sequence`	Ordered set of heterogeneous data type instances
`map`	Maps from an identifier to an instance of the bound type
`window`	Collection of bound type instances that is constrained either to a fixed number of items or a fixed time interval
`identifier`	Key used in maps
`iterator`	Used to iterate over all instances in a map (keys) or window (data values)

- subscription(s)
- association(s)
- declaration(s)
- initialization clause
- behavior clause

Each automaton source starts with binding a local variable to each publish/subscribe topic to which it wishes to be subscribed. Every time an event is delivered on any subscribed topic, the local variable refers to the last received event over that topic. An automaton must always subscribe to at least one topic. The cache provides a built-in topic, Timer, which delivers a tuple every second consisting simply of a timestamp.[2] All other topics must have been created by `create table` calls made by applications (or during cache initialization from a configuration file).

Associations are used to bind a local map variable to a persistent table. The automaton can then access tuples in the associated persistent table through calls to `lookup()` and `insert()` methods on that map variable.

Declarations enable the automaton to declare additional local variables needed for automaton processing. The initialization clause is executed after successful compilation of the automaton. It is usually used to initialise local variables, but can perform any actions supported by the language.

The behavior clause is executed each time an event is delivered to any topic to which the automaton is subscribed.

[2] This is an example of punctuation-carrying heartbeat functionality [22].

4.3 Example hybrid automaton

Households occupied by multiple, unrelated adults (e.g. students sharing a flat) often opt for broadband plans with rapidly escalating charges if a per month bandwidth allowance is exceeded. These households wish to control bandwidth consumption as it nears the monthly allowance. Additionally, it is often the case that a single member of the household is usually the cause of exceeding the monthly allowance; therefore, there is a desire to track the usage of a subset of the members of the household.

The required tables to support this functionality are shown in Fig. 3. The `Allowances` table is populated with a monthly download byte-limit for monitored IP addresses using a network management utility. The `BWUsage` table records accumulated usage; a network management utility is used to reset the accumulated usage to 0 at an appropriate frequency.

```
create table Flows (protocol integer, srcip varchar(16), sport
     integer, dstip varchar(16), dport integer, npkts integer,
     nbytes integer)
create persistenttable Allowances (ipaddr varchar(16) primary
     key, bytes integer)
create persistenttable BWUsage (ipaddr varchar(16) primary key,
     bytes integer)
```

Fig. 3. Bandwidth usage tables

```
subscribe f to Flows;
associate a with Allowances;
associate b with BWUsage;
int n, limit;
identifier ip;
iterator it;
sequence s;
string st;
behavior {
   ip = Identifier(f.daddr);
   if (hasEntry(a, ip)) {
      limit = seqElement(lookup(a, ip), 1);
      if (hasEntry(b, ip))
         n = seqElement(lookup(b, ip), 1);
      else
         n = 0;
      n += f.nbytes;
      s = Sequence(f.daddr, n);
      if (n > limit)
         send(s, limit, 'limit exceeded');
      insert(b, ip, s);
   }
}
```

Fig. 4. Bandwidth usage consumption

The automaton of Figure 4 tracks the bandwidth usage for each monitored IP address, generating a notification to the registering application (in our case, a policy-based management system) when a limit has been exceeded. The automaton subscribes to `Flows` events. It associates `a` and `b` with the persistent tables `Allowances` and `BWUsage`, respectively. Upon receipt of each `Flows` event, it does the following:

- generates an identifier from the destination address in the `Flows` event
- if no entry for this IP address in `Allowances`, stop processing
- lookup allowance for this IP address
- if entry for IP address in `BWUsage`, fetch accumulated usage, otherwise 0
- increment usage by number of bytes in this `Flows` tuple
- if limit exceeded, send event information to registering application
- update usage for this IP address

5 Automaton execution model

When an application registers an automaton against the Cache, it provides the source code for the automaton along with data required for the cache to create an RPC channel back to the registering application. The source code is compiled into instructions for a stack machine; if a compilation error is detected, information about the error is communicated back to the registering application, and the RPC channel is closed.

Upon successful compilation, a new PThread is created to implement the automaton, and an identifier is returned to the registering application; this identifier can be used to manage the automaton at a later time.

When the PThread is created, the byte code sequences resulting from the compilation of the initialization and behavior clauses are bound to an instance of the stack machine interpreter. The initialization sequence is executed, and the thread then enters a continuous loop, awaiting an event on one of its subscribed topics; the runtime guarantees that tuples are delivered to an automaton in strict time-of-insertion order. Upon receipt of an event, the behavior sequence is executed. If an automaton executes a `send()` in the behavior sequence, an RPC containing the `send()` arguments is made to the registering application. If the automaton executes a `publish()` in the behavior sequence, a tuple is inserted into the table/topic specified in the `publish()` arguments, potentially triggering other automata to execute.

The default PThread scheduling algorithm is used by the cache. Appropriate conditional critical regions are used to guarantee safe execution. The language runtime implements an aggressive garbage collection policy as soon as it knows that heap allocated storage is no longer in use; the `delete()` procedure can be invoked by code to advise when storage is no longer in use.

5.1 Optimizations enabled by the execution model

Many complex event processing systems based upon the stream database model (streams and relational operators) require the creation of multiple temporary

event streams to enable the operators to perform the requisite aggregation and disaggregation operations demanded by the pattern matching logic. This leads to a very large number of operators that must be scheduled, and a very large number of additional tuples that need to be delivered to the tree of operators that represent the query [7].

The imperative structure of the automaton language, together with the ability to declare and manipulate automaton-local state, enable the aggregation of multiple operators into a single automaton, thus reducing the scheduling stress on the event processing system. The following example, documented more fully in [23], demonstrates this effect.

The DEBS 2012 Grand Challenge posed two queries with regards to monitoring for complex events from manufacturing equipment. Here we discuss the first query, partially illustrated in Fig. 5. The query consisted of 15 operators in total, only five of which are shown. The other ten are replications of those illustrated.

Operators 1 and 4 compute a state transition by correlating attributes of the initial stream S_0; once a pattern is detected, an event is directed to the operator 7. Operator 7 looks for events S_5 followed by events S_8. This code is sequential, since logically the latter operator awaits events in order. The sequential nature of the automaton language allowed to put them together under one automaton – i.e. one execution thread.

Operator 10 and 11 operate on the state, a 24-hour window of events generated by operator 7. The former computes a least squares fit over the stored events; and the latter looks for an increase in the equipment delay and raises an alarm. If these operators are treated separately, then exactly the same state must be maintained twice. With our automata, this duplication can be avoided.

Finally, all code can be merged into a single automaton. And, finally simply append the next two groups of operators one after the other since they are independent.

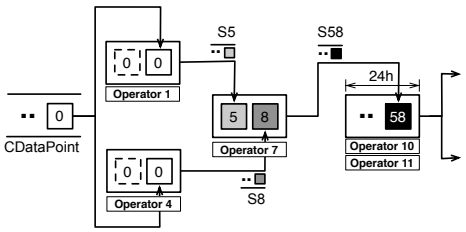

Fig. 5. Exemplar operators from the DEBS 2012 Grand Chellange

6 Evaluation

Experiments are run on two AMD Athlon 64 dual core 2.7GHz processors with 4GB of RAM running Ubuntu Linux 2.6 and Window 7, respectively. The cache

is implemented as a multi-threaded process. The main thread serially processes RPC requests from other processes. Upon successful compilation of an automaton, a new thread is created to animate that automaton. The remainder of this section documents the performance of automata in this environment.

6.1 Cost of Built-in functions

It is important to characterize the costs of invoking built-in functions in the language. The automaton template in Fig. 6 was used to measure the execution costs to invoke different built-in functions.

```
subscribe t to Timer;
int i;
int limit;
tstamp start;
int diff;
# built-in specific declarations
initialization {
   limit = 100000;
   # built-in specific initialization
   print('===== Start of <built-in> test =====');
}
behavior {
   i = 0;
   start = tstampNow();
   while (i < limit) {
      # invoke built-in
      i += 1;
   }
   diff = tstampDiff(tstampNow(), start);
   print(String('<built-in>: ', float(diff)/100000000.0));
}
```

Fig. 6. Built-in cost template automaton

The built-in specific declarations, initialization, and invocation are incorporated into each automaton as appropriate. The `print()` built-in is used to print the string argument on the standard output; the number printed is the number of microseconds required for each invocation of the built-in. Each automaton was executed for 2 minutes on an unloaded machine. Figure 7 shows the minimum, 25^{th}, 50^{th}, 75^{th} percentiles, and maximum of execution times recorded for each built-in, with nothing indicating the per-iteration overhead due to the while loop. [Note that limit for publish() was 50000, and for send() was 1000.]

Several things are apparent from this data:

- the stack machine interpreter acts like a processor with a $\sim 3\mu s$ instruction cycle;
- identifier generation, which requires access to the heap and copying of strings, takes a bit over 2 instruction cycles;
- publishing an event to another topic takes ∼3 instruction cycles; and
- sending an event to an external process takes $\sim 200\mu s$.

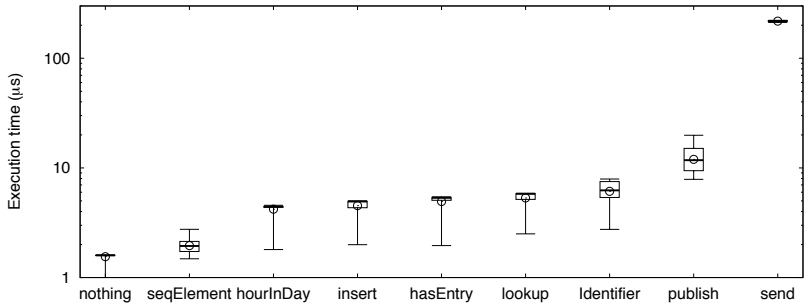

Fig. 7. The execution cost of built-in functions

6.2 Performance at Scale

It is important to understand how the Cache performs as the number of automata and also the frequency of tuple insertion scale up. To stress the system, we vary the number of automata that subscribe to the Flows topic. Independently, we vary the frequency of tuple insertion into the Flows table. An important measure of the ability for the system to handle the increased scale is the delay between when a tuple is inserted into the table/topic, and when each subscribed automaton processes the event. This is measured using the automaton in Fig. 11.

```
subscribe f to Flows;
real min, max, ave, r;
int count, nsecs;
string id;
initialization {
      min = 1000.;
      max = 0.;
      ave = 0.;
      id =  A ;
      count = 0;
}
behavior {
      count = count + 1;
      nsecs = tstampDiff(tstampNow(), f.tstamp);
      r = float(nsecs) / 1000000.;
      ave = ave + (r - ave) / float(count);
      if (r > max)
         max = r;
      if (r < min)
         min = r;
      if (count >= 1000) {
         print(String(id, ': ', ave, , , min, ', ', max));
         count = 0;
         min = 1000.;
         max = 0.;
         ave = 0.;
      }
}
```

Fig. 8. Performance at scale template automaton

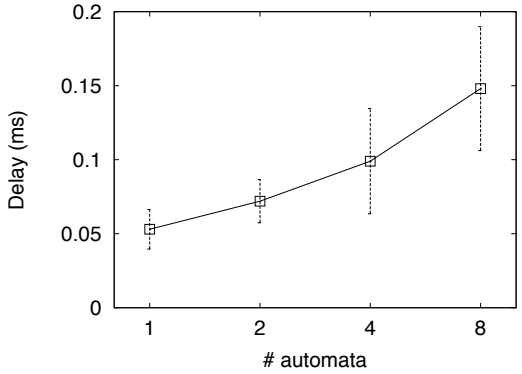

Fig. 9. Delay vs. # automata, $\Delta t = 8\mu s$

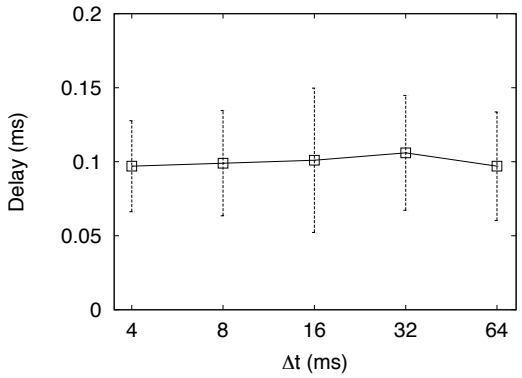

Fig. 10. Delay vs. event inter-arrival rate with 4 automata running

For each automaton, a different value is assigned to `id`; the subsequent log is analysed for mean and standard deviation for the average delay observed across all automata, as well as for minimum and maximum delays observed. The independent parameters for the experimental runs are the number of automata simultaneously subscribed and the cycle period of tuple insertion into Flows, Δt.

Figure 9 displays the measured delay parameters for $\Delta t = 8ms$. It is clear that the average delay grows linearly as the number of automata scales from 1 to 8. In the deployments to date, the typical number of tuples inserted are approximately 100 events/sec; $\Delta t = 8ms$ corresponds to an insertion rate of 125 tuples/second.

Figure 10 shows the measured delay parameters for 4 automata as Δt scales from $4ms$ to $64ms$ (insertion rates of 250 events/sec to 16 events/sec). The average and variance of the delay remain essentially constant across this range of packet insertion rates.

The system scales well with number of automata and frequency of tuple insertion. The linear growth in average and standard deviation of delay with number of automata is consistent with scheduling of increasing numbers of PThreads. The constancy of average and variance against insertion frequency indicates that there is plenty of execution capacity in the Cache for the loads presented.

It is important to note that it is quite uncommon in our experience to have several automata subscribed to high frequency topics like Flows.

6.3 Performance at stress

Another important measure of the capacity of the system is the maximal rate at which it can absorb and generate RPCs. To measure this, we executed the following automaton to measure 1-way and 2-way stress performance, with a single application performing insert calls into a `Test` table as rapidly as possible. Note that to measure 2-way stress, simply uncomment line 15 in Figure 11.

```
subscribe t to Timer;
subscribe s to Test;
int count;
initialization {
   count = 0;
   print('===== Start of stress test =====');
}
behavior {
   if (currentTopic() == 'Timer') {
      if (count > 0)
         print(String('stress1way: ', count));
      count = 0;
   } else {
      count += 1;
      send(s); # uncomment for 2-way test
   }
}
```

Fig. 11. Performance at stress template automaton

The performance as the number of integer fields in the Test schema varies from 1 to 16 is shown in Fig. 12.

Figure 13 shows the performance as the number of characters in a schema consisting of a single varchar field varies from 1 to 10,000. Note that the RPC system performs fragmentation/reassembly at 1024-byte boundaries, so the linear drop with buffer size is to be expected.

6.4 Finding frequent items

Figure 14 shows the "frequent" algorithm [17]. It stores $k-1$ out of n items. The final set contains at least those that have occurred n/k times. The execution time is dominated by $O(1)$ and $O(k)$ operations. As k increases, the more the

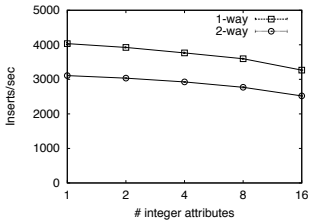

Fig. 12. Integer stress test

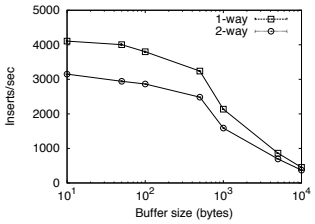

Fig. 13. Character string stress test

$O(1)$ operations, the more expensive the $O(k)$ ones. The mean (μ) decreases but the standard deviation (σ) increases.

The input data are 264,745 out-going HTTP requests to 5,572 unique hosts, as logged by the Homework router running in a small office environment at the University of Glasgow. Figure 15 shows the Zipfian frequency distribution of the data set, where hosts are ranked by their popularity. This is a well-known characteristic of Web traffic.

Another approach is to introduce a built-in procedure if an implementation in the automaton language is insufficiently performant. E.g.,

```
subscribe e to Urls;
behavior { frequent(T, Identifier(u.host), k); }
```

Figure 16 shows the coefficient of variation (σ/μ) for both the automaton and built-in implementation of the algorithm.

6.5 Comparison with Cayuga

The examples from Cayuga are usually described using stock market queries. Most of the examples, as well as the datasets provided with the distribution, are related to complex analysis for stock investors. The largest dataset used contains 112,635 anonymized stock events.

We evaluate three Cayuga queries against the Cayuga engine. The Cayuga engine best compiles in Microsoft's Visual Studio, thus the experiments were run on a Window platform. Figure 18 shows the results from comparing the execution time of those queries with equivalent implementations in the automaton programming language.

The Cayuga execution times are the elapsed time after all events have been loaded into memory and until all events have been processed. The Cache was never provisioned for post-hoc analysis of in-memory data: all events are processed in real-time. For a fair comparison, we derive our timings by first appending all events in a window, and then iterate over the window and execute the queries.

The first query is an example of a basic operator. In Cayuga, the query has the form `SELECT * from Stocks PUBLISH T`. The equivalent automaton is one that subscribes to stream `Stocks` and publishes an event to another stream `T`. The performance improvement against Cayuga is an order of magnitude. This

```
subscribe e to Urls;
map T;
iterator i;
identifier id;
int count;
int k;
initialization {
   k = k;
   T = Map(int);
}
behavior {
   id = Identifier(e.host);
   if (hasEntry(T, id)) {
      count = lookup(T, id);
      count += 1;
      insert(T, id, count);
   } else if (mapSize(T) < (k-1))
      insert(T, id, 1);
   else {
      i = Iterator(T);
      while(hasNext(i)) {
         id = next(i);
         count = lookup(T, id);
         count -= 1;
         if (count == 0)
            remove(T, id);
         else
            insert(T, id, count);
      }
   }
}
```

Fig. 14. The "frequent" algorithm [17]

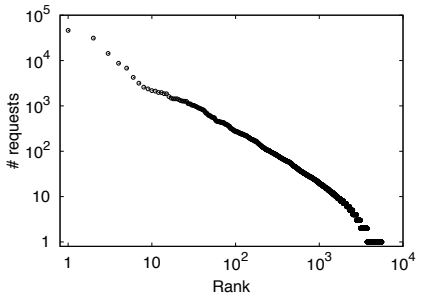

Fig. 15. Number of requests per Web page ordered by popularity

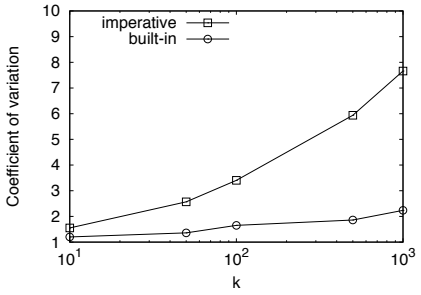

Fig. 16. Imperative *vs.* built-in execution time of the frequent algorithm

strengthens the argument of the efficacy of the automaton execution model and the efficient unification of publish/subscribe with the data stream management code.

The second query is an example of using multiple states. The query detects price movements over time. In particular, it asks for notifications whenever there is double-top formation in the price chart of any stock – this is a well-known pattern amongst analysts, otherwise known as M-shaped pattern. The automaton detects the pattern twice as fast as Cayuga. This pattern is best illustrated with an example. Fig. 17 shows one of the M-shaped patterns detected in the data set. Due to space limitations, the Cayuga query is omitted. It is available at `www.cs.cornell.edu/bigreddata/cayuga/`.

Our implementation maintains states A-F in a map of stocks; each entry represents a small state machine. Once all states A-F are true, then the pattern is detected. Depending on the current stock price, the algorithm backtracks to previous states or proceeds to the next (ascending or descending) accordingly. Note here that our solution is algorithmic (if-then-else). This is another example of the ability to implement multiples state machines under a single execution thread, which contributes to the substantial performance enhancement.

The final query is a prime example of the use of the FOLD operator; it is used to maintain aggregates and windows under time- or attribute-based conditions.

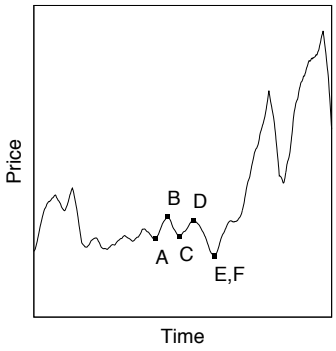

Fig. 17. Q2 illustrated

The desired behaviour is to be able to detect continuous runs of increasing prices for each stock, and to be able to display the sequence of events that constituted each run. This has been done simply and effectively using two automata that are substantially faster ($\times 50$) than the Cayuga equivalent.

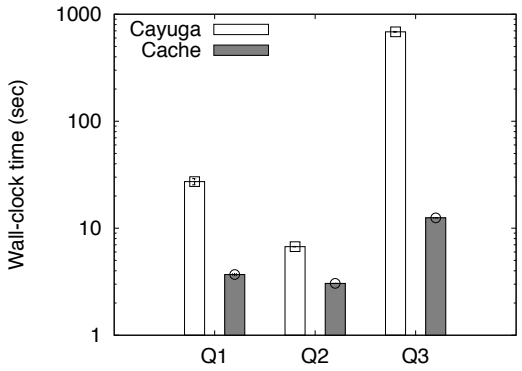

Fig. 18. Benchmarking against Cayuga

7 The impact on complex event processing

We have proposed an algorithmic, imperative approach to complex event processing.

The first impact is Turing completeness, something that was proposed from a theoretical basis on [3]. Some event languages have a solid background in event calculus and result in one line expressions that may be compact implementations but are not apt to user optimizations – apart from physical query plans. For

example, there is a large body of complex event language research on Kleene closures [9]. Although not explicitly documented here, we have implemented SASE's kleene closure operator (e.g. based on partition contiguity) with a map of windows.

Our experiences thus far are that the imperative programming style of GAPL allows it to be used in many domains: home network management [4], industrial applications [23], and, given the latest endeavours comparing to Cayuga, stocks.

Apart from expressiveness, the imperative programming model is substantially faster. It may be viewed as an assembly language to which higher-level complex event languages can subsequently compile.

8 Conclusions

It is clear that the automaton language, as integrated into the cache, provides a very high-performance complex event processing capability. It can be criticized for its imperative, C-like structure, in terms of usability by individuals wanting to deploy their own automata. We have started to investigate compilation of stream expressions for complex event patterns, such as Cayuga's, into equivalent automata. An alternative approach is to compile stream expressions directly into instructions for the stack machine that underlies the cache.

In comparing with Cayuga, we have determined that we need to be able to create streams on the fly. This will enable exploration of the dynamic demultiplexing of streams, as lately discussed in [24]. We continue our comparative endeavours with the Linear Road Benchmark [25].

References

1. Terry, D., Goldberg, D., Nichols, D., Oki, B.: Continuous queries over append-only databases. In: Proceedings of the 1992 ACM SIGMOD international conference on Management of data. SIGMOD '92, New York, NY, USA, ACM (1992) 321–330
2. Codd, E.F.: A relational model of data for large shared data banks. Commun. ACM **13** (June 1970) 377–387
3. Law, Y.N., Wang, H., Zaniolo, C.: Query languages and data models for database sequences and data streams. In: Proceedings of the Thirtieth international conference on Very large data bases - Volume 30. VLDB '04, VLDB Endowment (2004) 492–503
4. Sventek, J., Koliousis, A., Dulay, N., Pediaditakis, D., Rodden, T., Lodge, T., Sharma, O., Sloman, M., Bedwell, B., Glover, K., Mortier, R.: An Information Plane Architecture Supporting Home Network Management. In: Integrated Network Management, IFIP/IEEE (May 2011)
5. Arasu, A., Babu, S., Widom, J.: The cql continuous query language: semantic foundations and query execution. The VLDB Journal **15** (June 2006) 121–142
6. Abadi, D.J., Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., Zdonik, S.: Aurora: a new model and architecture for data stream management. The VLDB Journal **12** (August 2003) 120–139

7. Demers, A., Gehrke, J., Hong, M., Riedewald, M., White, W.: Towards expressive publish/subscribe systems. In: Proceedings of the 10th international conference on Advances in Database Technology. EDBT'06, Berlin, Heidelberg, Springer-Verlag (2006) 627–644
8. Brenna, L., Demers, A., Gehrke, J., Hong, M., Ossher, J., Panda, B., Riedewald, M., Thatte, M., White, W.: Cayuga: a high-performance event processing engine. In: Proceedings of the 2007 ACM SIGMOD international conference on Management of data. SIGMOD '07, New York, NY, USA, ACM (2007) 1100–1102
9. Agrawal, J., Diao, Y., Gyllstrom, D., Immerman, N.: Efficient pattern matching over event streams. In: Proceedings of the 2008 ACM SIGMOD international conference on Management of data. SIGMOD '08, New York, NY, USA, ACM (2008) 147–160
10. Wu, E., Diao, Y., Rizvi, S.: High-performance complex event processing over streams. In: Proceedings of the 2006 ACM SIGMOD international conference on Management of data. SIGMOD '06, New York, NY, USA, ACM (2006) 407–418
11. Gyllstrom, D., Agrawal, J., Diao, Y., Immerman, N.: On supporting kleene closure over event streams. In: Proceedings of the 2008 IEEE 24th International Conference on Data Engineering. ICDE '08, Washington, DC, USA, IEEE Computer Society (2008) 1391–1393
12. Sadri, R., Zaniolo, C., Zarkesh, A., Adibi, J.: Expressing and optimizing sequence queries in database systems. ACM Trans. Database Syst. **29**(2) (June 2004) 282–318
13. Mozafari, B., Zeng, K., Zaniolo, C.: From regular expressions to nested words: unifying languages and query execution for relational and xml sequences. Proc. VLDB Endow. **3**(1-2) (September 2010) 150–161
14. White, W., Riedewald, M., Gehrke, J., Demers, A.: What is "next" in event processing? In: Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems. PODS '07, New York, NY, USA, ACM (2007) 263–272
15. Sullivan, M., Heybey, A.: Tribeca: a system for managing large databases of network traffic. In: Proceedings of the annual conference on USENIX Annual Technical Conference. ATEC '98, Berkeley, CA, USA, USENIX Association (1998) 2–2
16. Cranor, C., Johnson, T., Spataschek, O., Shkapenyuk, V.: Gigascope: a stream database for network applications. In: Proceedings of the 2003 ACM SIGMOD international conference on Management of data. SIGMOD '03, New York, NY, USA, ACM (2003) 647–651
17. Cormode, G., Hadjieleftheriou, M.: Finding the frequent items in streams of data. Commun. ACM **52** (October 2009) 97–105
18. Sekar, V., Reiter, M.K., Zhang, H.: Revisiting the case for a minimalist approach for network flow monitoring. In: Proceedings of the 10th annual conference on Internet measurement. IMC '10, New York, NY, USA, ACM (2010) 328–341
19. Kandula, S., Chandra, R., Katabi, D.: What's going on?: learning communication rules in edge networks. In: Proceedings of the ACM SIGCOMM 2008 conference on Data communication. SIGCOMM '08, New York, NY, USA, ACM (2008) 87–98
20. Brenna, L., Gehrke, J., Hong, M., Johansen, D.: Distributed event stream processing with non-deterministic finite automata. In: Proceedings of the Third ACM International Conference on Distributed Event-Based Systems. DEBS '09, New York, NY, USA, ACM (2009) 3:1–3:12
21. Abadi, D.J., Ahmad, Y., Balazinska, M., Cetintemel, U., Cherniack, M., Hwang, J.H., Lindner, W., Maskey, A.S., Rasin, A., Ryvkina, E., Tatbul, N., Xing, Y.,

Zdonik, S.: The Design of the Borealis Stream Processing Engine. In: Second Biennial Conference on Innovative Data Systems Research (CIDR 2005), Asilomar, CA (January 2005)
22. Johnson, T., Muthukrishnan, S., Shkapenyuk, V., Spatscheck, O.: A heartbeat mechanism and its application in gigascope. In: Proceedings of the 31st international conference on Very large data bases. VLDB '05, VLDB Endowment (2005) 1079–1088
23. Koliousis, A., Sventek, J.: DEBS Grand Challenge: Homework automata. In: Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems (To Appear). (2012)
24. Zeitler, E., Risch, T.: Massive scale-out of expensive continuous queries. PVLDB **4**(11) (2011) 1181–1188
25. Arasu, A., Cherniack, M., Galvez, E., Maier, D., Maskey, A.S., Ryvkina, E., Stonebraker, M., Tibbetts, R.: Linear road: a stream data management benchmark. In: Proceedings of the Thirtieth international conference on Very large data bases - Volume 30. VLDB '04, VLDB Endowment (2004) 480–491