Unification of Publish/Subscribe Systems and Stream Databases

Joseph Sventek, Alexandros Koliousis

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
{joseph.sventek,alexandros.koliousis}@glasgow.ac.uk

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 either stream database management systems or publish/subscribe systems, are adept at handling many of these applications. However, 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, streams, 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 active 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 (henceforth, the *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. An imperative programming language – viz., 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 organised as follows. Firstly, we describe related work to place our system in context (§2). This is followed by a discussion of the Cache architecture (§3), the automaton programming language (§4), and the automaton execution model (§5). We then proceed with an evaluation of the system's performance from a number of perspectives, including a performance comparison against Cayuga for several, relevant stock analysis queries (§6). The paper concludes with a discussion of the impact that such a unified system has on future complex event processing (§7), and future work (§8).

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 [1]. 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.

The continuous semantics of queries were first defined in Tapestry, a database system for mail and bulletin board messages [2]. 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. Fig. 1 shows a variant of the basic continuous query execution model, as proposed in Tapestry. This model inspired the first generation of interactions with the Cache, where continuous queries over network flow streams were used to produce real-time visualisations of home networking traffic [4]. Fig. 1 also shows

```
1  # Let T be a table with attribute x.
   Set τ = −∞
   Set S = ∅
   FOREVER DO
5     S = {select x from T [since τ]}
      # Each tuple j in S has a timestamp t_j.
      τ = arg max_{j∈S} t_j
      Return results to user
      Sleep for some period of time, t sec.
10 ENDLOOP
```

```
11 subscribe e to T; # A new tuple e of T.
   subscribe p to Timer;
   window S;
   int period, τ;
   initialization {
16    S = Window(sequence, SECS, t);
      period = t;
      τ = 0;
   }
   behavior {
      if (currentTopic() == 'T')
22       append(S, Sequence(e.x), e.tstamp);
      else { # The current topic is Timer.
         τ += 1;
         if (τ % period == 0) {
26          send(S); # Return results.
            S = Window(sequence, SECS, t);
         }
      }
30 }
```

Fig. 1. The continuous query execution model [2] (left); and its equivalent Glasgow automaton (right). Our [since τ] extension to select (line 5) guarantees to return all tuples that have been inserted into table T in the last t seconds. On the other hand, the automaton reacts upon every insertion of a tuple e to table T, populating a time-based sliding window S of duration t (lines 16, 22). Every t seconds, the automaton sends this window to its registering process (line 26).

an equivalent implementation of the continuous query model in GAPL, as an introductory example to our automata and the unified nature of the Cache. The nature of this unification, as well as our language features, will become apparent in the subsequent sections.

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 Glasgow automata.

First, an automaton can not only update local state, but also append tuples to other streams, locally (via a publish command) or remotely (via a send command). Second, in contrast to other event query languages, state need not necessarily 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 state automata, a computational model used in the event query 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; or, the skip till next match operator of SASE maintains intermediate state in arrays in order to express Kleene closures, an operator that has recently received considerable attention in complex event detection [12]. For non-deterministic finite automata, the complexity of a pattern lies in determining what comes "next" in event processing [13]. 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 query that uses the local state maintained by another.

The Cache was first used as a stream database of network flows and related policies that govern a home network. Network monitoring and management scenarios have been featured heavily in stream database research. The Tribeca query language [14], for example, supported demultiplexing and multiplexing of packet streams. The demultiplexing of streams, while similar to the group by SQL operator, enables processing of sub-streams beyond mere aggregate statistics using pipes to transform streams at several stages. The importance of multiplexing (merging) network streams has also been stressed in Gigascope, a high-performance network monitoring tool [15]. Gigascope has a two-level query architecture to process packets on high-speed links. High performance is achieved by pushing low-level queries (e.g., protocol filtering or aggregation) closer to the physical network interface. High-level queries then perform more complex tasks.

A number 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 [16], and its applications to network monitoring include finding the heaviest bandwidth consumers (heavy-hitters), or finding the heaviest connection initiators (super-spreaders) [17]. These algorithms are expressible, as we will demonstrate, in GAPL. Finally, 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. Thus, 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 temporally correlated flows are used for classification or intrusion detection and are usually expressed as sequences of events [18].

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 section describes a topic-based publish/subscribe cache that facilitates such real-time processing.

A working system consists of our centralised Cache and a varying number of applications that use it; the applications and the Cache interact through a custom RPC mechanism. There are three distinct roles that applications can assume with respect to this system:

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

The Cache supports the usual SQL commands for creating tables and inserting tuples into tables. The Cache supports 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 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.

For monitoring applications, the selection operator has been augmented with appropriate time and count window extensions to reflect the continuous nature of the events. Thus, apart from typical order by and group by operators, *ad hoc* select queries over cached streams can use time interval expressions that narrow the scope of results to a particular time period, e.g. select $*$ from table [since τ], where τ is the timestamp of the last retrieved tuple. Typically, monitoring applications submit such queries periodically.

The third role of applications, viz. reaction applications, is enabled by the unification of this stream database view of events with a publish/subscribe infrastructure, achieved as follows. Every table created in the Cache, whether ephemeral or persistent, corresponds to a publish/subscribe topic with the same name. Whenever a tuple is inserted into a table, that tuple is published as an event to its associated topic. Applications can register automata (i.e., complex event patterns) against the database. As new tuples are inserted into a table, all automata that have subscribed to that topic will receive events for processing. If, while processing an event, an automaton determines that it has detected a pattern of interest, it may send information about the complex event to the application that registered the automaton.

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

Table 1. Description of data types.

Type	*Basics*
int	64-bit integer
real	Double-precision floating point
tstamp	64-bit unsigned integer (*nsec* since the epoch)
bool	True or false
string	Variable-length UTF-8 character array
Type	*Aggregates*
sequence	Ordered set of heterogeneous basic data type instances
map	Map from an identifier to an instance of the bound type
window	Collection of bound type instances, 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)

Additionally, during normal processing of events, an automaton may publish (append, insert, or update) a new tuple into another table in the Cache, 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. A typical reaction application (e.g., a policy management engine) registers one or more automata with the Cache.

4 The automaton programming language

4.1 Language design principles

Support for complex event pattern matching, requiring both consumption of raw publish/subscribe events and access to static relations, dictated the following features of the Glasgow Automaton Programming Language:

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

Furthermore, one requires a small set of basic data types, a small set of aggregate data structures, and a small set of control constructs to store and filter events locally. The basic data types are described in Table 1. The language also defines a minimal set of aggregate types, e.g. a sequence, a map, or a window, and types required to manipulate these aggregate types, e.g. an iterator over a window.

Every aggregate type is instantiated with a constructor. Note that a sequence instance can contain heterogeneous basic type instances, while each map or window instance is bound to a particular type, basic or aggregate. In fact, the ability to instantiate windows of sequences or maps of sequences enables the creation of ephemeral or persistent tables, respectively, that are truly local within the context of an automaton thread. Finally, the language supports if then else and while constructs. It also supports a typical set of operators for arithmetic, conditional expressions, and assignment.

Overall, the C-like syntax for GAPL was chosen to facilitate the coding of commonly-used stream processing algorithms (cf. §6.4), while enabling high-performance filtering of events.

4.2 General form for an automaton

In its general form, an automaton program consists of subscriptions, associations, declarations, an initialization clause, and a behavior clause – in that order.

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 bound local variable refers to the last received event over that topic. Attribute values are assigned automatically to variables with names and types being determined by the corresponding table schema. These variables are accessed using the dot notation. For example, variable e.x holds the value of attribute x of event e. The Cache provides a built-in topic, Timer, which delivers a tuple every second consisting simply of a timestamp data type attribute.[2] All other topics must have been created earlier by create table calls made by applications (or during Cache initialization, from a configuration file). An automaton must always subscribe to at least one topic.

Associations are used to bind a local map variable to a persistent table in the Cache. The automaton can then access and modify tuples in the associated persistent table through calls to lookup() and insert() methods on that map variable, respectively. Subsequent declarations in an automaton enable the programmer to declare additional local variables needed for processing.

The initialization clause of an automaton is executed once, after successful compilation. It is usually used to initialise local variables, but therein a programmer can perform any actions supported by the language. The behavior clause, on the other hand, is executed each time an event is delivered to any of the subscribed topics.

4.3 Example hybrid automaton

This section describes an automaton that implements a hybrid application scenario, one in which the the processing of events depends upon access to global persistent policy state, taken from the current deployments of the Cache – i.e., as part of a home network router [4].

[2] This is an example of punctuation-carrying heartbeat functionality [19].

```
31 create table Flows (
     proto integer,
     saddr varchar(16), sport integer,
     daddr varchar(16), dport integer,
     npackets integer, nbytes integer)

32 create persistenttable Allowances (
     ipaddress varchar(16) primary key,
     nbytes integer)

33 create persistenttable Usage (
     ipaddress varchar(16) primary key,
     nbytes integer)
```

Fig. 2. Tables associated with the bandwidth usage consumption automaton.

```
34 subscribe f to Flows;
   associate a with Allowances;
   associate b with Usage;
   int n, limit;
   identifier ip;
   sequence s;
   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);
     }
54 }
```

Fig. 3. Bandwidth usage consumption automaton.

Households occupied by multiple, sometimes 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 three tables used to demonstrate this functionality are shown in Fig. 2. Table Flows is an ephemeral table populated with periodic aggregate statistics of home networking traffic (i.e., per flow number of bytes and number of packets accumulated every second). The other two tables are persistent. The Allowances table is populated with a monthly download byte-limit per monitored IP address using a network management utility; and the Usage table records accumulated bandwidth usage, reset to zero by a network management utility at an appropriate frequency.

Fig. 3 shows an automaton that tracks the bandwidth usage per monitored IP address, generating a notification to the registering application (i.e., a policy-based management system) when a limit has been exceeded. The automaton subscribes to Flows events; and associates maps a and b with tables Allowances and Usage, respectively. Upon receipt of each Flows event, it does the following:

- it generates an identifier ip from the flow's destination address;
- if no entry for ip is found in Allowances, it stops processing;
- otherwise, it looks up an allowance for this ip address;
- if there is an entry for ip in table Usage, it fetches the accumulated usage;
- otherwise, it sets usage n to zero;
- it increments n by the number of bytes in the Flows tuple;

– if ip's limit is exceeded, it sends an event to the registering application; and
– it updates the usage for this IP address in the Cache.

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 (i.e., a host, a port, and a service name). The source code is then 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 animate the automaton, and an identifier is returned instead; this identifier can be used by the registering application to manage the automaton at a later time (e.g., to unregister it).

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 a stack machine interpreter. The initialization sequence is executed once and the thread then enters a continuous loop, awaiting an event on one of its subscribed topics (a tuple insertion); the runtime system 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 call, 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 and appropriate conditional critical regions are used to guarantee safe execution amongst multiple automata threads.

The 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 optionally 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 require the creation of multiple temporary event streams for their operators to perform the requisite aggregations and disaggregations demanded by a 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 a directed acyclic graph of operators that represent the query. The imperative structure of GAPL, together with the ability to declare and manipulate an automaton's local state, enable combinations of multiple operators into a single automaton, thus reducing the scheduling stress on the Cache. The following example, documented fully in [20], demonstrates this effect.

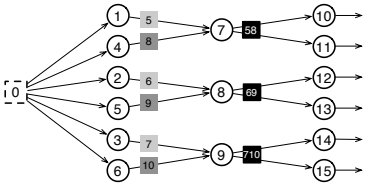

Fig. 4. The DEBS 2012 Grand Challenge query.

The DEBS 2012 Grand Challenge posed two complex queries with regards to monitoring manufacturing equipment. Here we discuss the first query, illustrated in Fig. 4. The query consists of fifteen operators (circles), generating nine intermediate streams (squares). Operators 1 and 4 compute a state transition by correlating consecutive events inserted into the initial stream 0; once a pattern is detected by either of them, events are published to operator 7. In turn, operator 7 looks for events in stream 5 followed by events in stream 8. Since this logic is sequential, GAPL allows us to combine these three operators together, in one automaton. Also, operators 10 and 11 have different functionality but they maintain the same state, a 24-hour sliding window over stream 58. If these operators were to be treated separately, then the same window would be maintained twice. Since their logic is independent, this duplication can also be avoided by merging them in one automaton. The other ten operators simply replicate the aforementioned functionality but on different attributes. In fact, our final solution merged all fifteen operators into a single automaton – i.e. one execution thread – avoiding the use of intermediate streams all together. By doing so, the throughput increased by 57% and the execution time decreased significantly, as opposed to having one automaton per operator [20].

5.2 Multi-query optimizations

Sharing derived events amongst Glasgow automata is currently a programmer's task and it is achieved by explicitly publishing these events to intermediate streams that are, in turn, accessible by other automata running in the context of the Cache; it is not an automated process. Managing these events programmatically, using basic or aggregate data type instances, is one of the key features that enable high-performance event filtering.

A limitation, however, of user-defined optimisations is that multiple automata registrations are not open to multi-query optimisations possible in other query engines. Cayuga, for example, merges equivalent automaton states into a directed acyclic graph using YFilters [21, 7], whose edges represent static or dynamic predicates that determine state transitions. This trade-off, where on one hand scaling to thousands of queries requires multi-query optimisations, while on the other hand high performance and expressiveness requires user-defined optimisations, is an open research question; we explore it by investigating the use of GAPL as an intermediate language between the two approaches.

```
55 subscribe t to Timer;
   int i;
   int limit;
   tstamp start;
   int dt;
60 # built-in specific declarations, e.g. sequence s; int x;
   initialization {
     limit = 100000;
63   # built-in specific initialization, e.g. s = Sequence('A', 1);
     print('Start of <built-in> test');
   }
   behavior {
     i = 0;
     start = tstampNow();
     while (i < limit) {
70     # invoke built-in, e.g. x = seqElement(s, 1);
       i += 1;
     }
     dt = tstampDiff(tstampNow(), start);
     print(String('<built-in>: ', float(dt)/100000000.0));
75 }
```

Fig. 5. Built-in cost template automaton.

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 Windows 7, respectively. The Cache is implemented as a multi-threaded process. Its main thread handles RPC requests (e.g., tuple insertions, automaton registrations) from other processes serially; new threads are created upon each successful automaton compilation. The remainder of this section documents the performance of automata in this environment.

6.1 Cost of Built-in functions

Automata are interpreted programs, thus it is important to characterise the costs of invoking built-in functions in the language. The automaton template in Fig. 5 was used to measure the execution costs to invoke a representative set of the built-in functions supported by the language. The built-in specific declarations (line 60), initialization (line 63), and invocation (line 70) were incorporated into the template as appropriate. The print function was used to display the results on standard output; the number printed is the number of microseconds required for each invocation of the built-in under test. Each automaton was executed for two minutes on an unloaded machine.

Fig. 6 shows the minimum, 25-th, 50-th, 75-th percentiles, and maximum of execution times recorded for each built-in.[3] Several things are apparent from this data:

[3] The overhead of the while loop was subtracted from the values produced by the automaton.

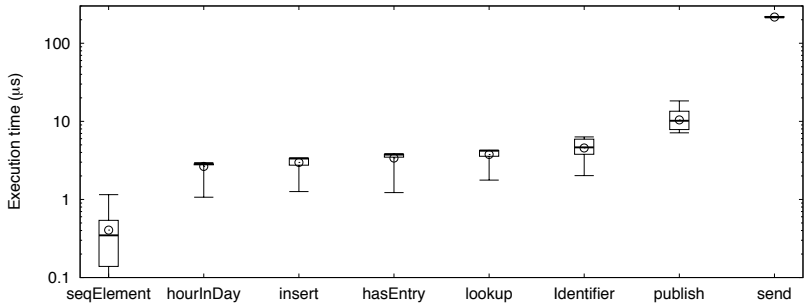

Fig. 6. The execution cost of built-in functions.

- the average cost of basic built-in functions (e.g. insert, lookup) is $\sim 3\mu s$;
- identifier generation, which requires access to the heap and copying of strings, is about twice as expensive as a basic built-in;
- publishing an event to another topic is about three times the basic built-in cost; and
- sending an event to an external process takes $\sim 200\mu s$.

6.2 Performance at Scale

As the number of simultaneously subscribed automata increases, one expects the scheduling delay for each automaton to increase. Thus, 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 (cf. §4.3). 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. 7. For each automaton, a different value is assigned to id (line 85); the subsequent log generated by the automaton is analysed for mean and standard deviation of the average delay observed across all automata, as well as for minimum and maximum delays observed (lines 91-93). The independent parameters for the experimental runs are the number of automata simultaneously subscribed and the cycle period of tuple insertion into Flows, Δt.

Fig. 8 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 one to eight. Note that, in the deployments to date, the typical number of flow tuples inserted are approximately 100 events/second; $\Delta t = 8ms$ corresponds to an insertion rate of 125 tuples/second. It is also important to note that it is quite uncommon in our experience to have several automata subscribed to high frequency topics like Flows.

```
76 subscribe f to Flows;
   real min, max, mean, dt;
   int count, nsecs;
   string id;
   initialization {
        min = 1000.;
        max = 0.;
        mean = 0.;
        count = 0;
85      id = 'A';
   }
   behavior {
        count += 1;
        nsecs = tstampDiff(tstampNow(),
             f.tstamp);
        dt = float(nsecs)/1000000.;
91      mean = mean + (dt - mean) /
             float(count);
        if (dt > max) max = dt;
93      if (dt < min) min = dt;
        if (count >= 1000) {
           print(String(id, ':', mean,
                ',', min, ',', max));
           min = 1000.;
           max = 0.;
           mean = 0.;
           count = 0;
        }
101 }
```

Fig. 7. Performance at scale template automaton.

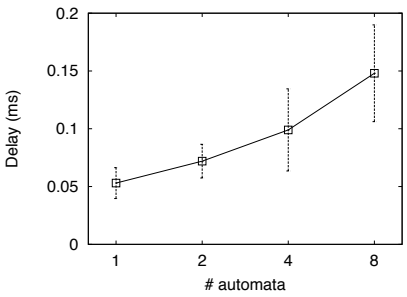

Fig. 8. Delay vs. # automata, $\Delta t = 8ms$.

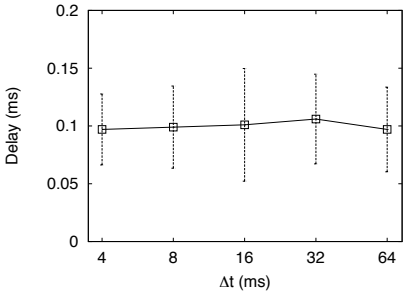

Fig. 9. Delay vs. event inter-arrival rate with 4 automata running.

Fig. 9 shows the measured delay parameters for four automata as Δt scales from $4ms$ to $64ms$ (insertion rates of 250 events/second to 16 events/second). The average and variance of the delay remain essentially constant across this range of packet insertion rates.

Thus, 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 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.

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 RPC requests and responses. To measure this, we executed the automaton of Fig. 10 to measure one-way and two-way stress performance, with a single application performing insert calls into a Test table as rapidly as possible. Note that to measure two-way stress performance, simply uncomment line 117 in the automaton.

```
102 subscribe t to Timer;
    subscribe e to Test;
    int count;
    initialization {
       count = 0;
       print('Start of stress test');
    }
    behavior {
       if (currentTopic() == 'Timer') {
          if (count > 0)
             print(String(count,
                   'events/sec'));
          count = 0;
       } else {
          count += 1;
          # Uncomment for 2-way stress test:
117       # send(s);
       }
119 }
```

Fig. 10. Performance at stress template automaton.

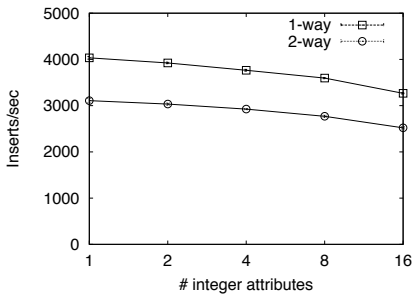

Fig. 11. Integer stress test.

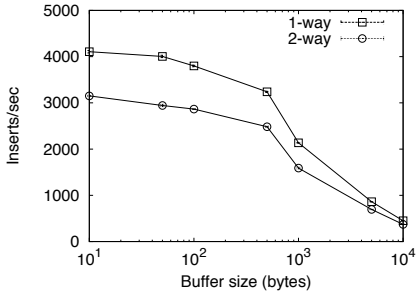

Fig. 12. Character string stress test.

The performance as the number of integer fields in the Test table schema varies from 1 to 16 is shown in Fig. 11. Fig. 12 shows the performance as the number of characters in a schema consisting of a single varchar field varies from 1 to 10,000. The RPC system performs fragmentation/reassembly at 1024-byte boundaries, so the drop with buffer size is to be expected.

6.4 Finding frequent items

This section evaluates the implementation of the "frequent" algorithm, a one-pass algorithm for finding the top-k items in a data stream [16], as a Glasgow automaton (Fig. 13). The algorithm stores $k-1$ out of n items, according to their popularity; after processing n events, the approximate result set will contain at least those items that have occurred n/k times. The input data to the automaton are 264,745 out-going HTTP requests (appended to an ephemeral table Urls) to 5,572 unique hosts, as logged by a router running in a small office environment at the University of Glasgow. Fig. 14 shows the Zipfian frequency distribution of the data set, where hosts are ranked by their popularity, a well-known characteristic of Web traffic.

An alternative approach is to introduce the algorithm as a built-in procedure in the language, a *de facto* approach in traditional query languages. Indeed, there

```
120 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);
          }
       }
150 }
```

Fig. 13. The frequent algorithm [16].

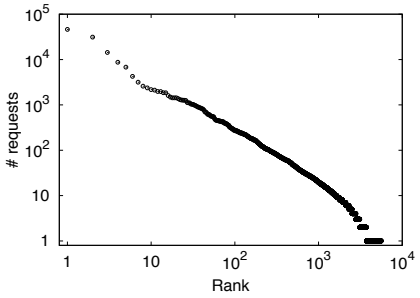

Fig. 14. Number of requests per Web page ordered by popularity

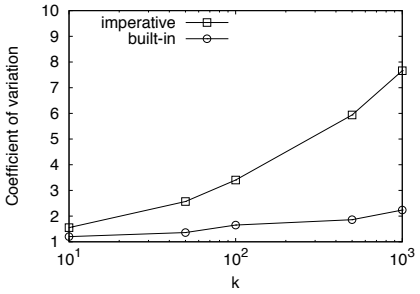

Fig. 15. Imperative *vs.* built-in execution time of the frequent algorithm.

```
151 subscribe e to Urls;
    map T;
    initialization { T = Map(int); }
154 behavior { frequent(T, Identifier(u.host), k); }
```

Fig. 16. The frequent algorithm as a built-in function.

might be situations where, despite the efficient code generated from the GAPL compiler, an interpreted solution may be insufficiently performant.

To demonstrate this, we also implemented the frequent algorithm as a built-in function in the language and evaluated the two approaches – namely, the automaton of Fig. 13 and the automaton of Fig. 16. From an algorithmic perspective, the execution time of these automata is dominated by $O(1)$ operations (e.g. inserting into T) and $O(k)$ operations (e.g., iterating over T). As k increases, the number of $O(1)$ operations increases, and the $O(k)$ operations become more expensive. Thus, it is expected that as k increases, the mean execution time (μ) will decrease and the standard deviation (σ) will increase. Fig. 15 shows the coefficient of variation (σ/μ) for both the imperative and the built-in implementation of the frequent algorithm.

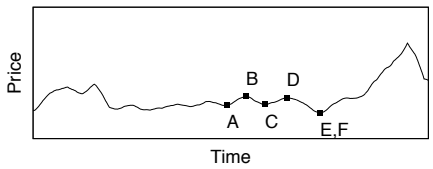

Fig. 17. An M-shaped pattern in the dataset illustrated.

6.5 Comparison with Cayuga

This section evaluates the performance of the Cache against the Cayuga query engine [7]. Most of the examples in the Cayuga distribution, as well as the data sets provided, are related to complex event processing for stock market investors. We evaluate three exemplar Cayuga queries against equivalent implementations in GAPL. The dataset used contains 112,635 anonymised stock events whose schema consists of a timestamp, an identifier as a company's name, and a price.

The first query is an example of a basic built-in operator that simply publishes incoming events to another stream. In the Cayuga Event Language (CEL), the query is:

```
155 SELECT * FROM Stock PUBLISH T
```

The equivalent Glasgow automaton is one that subscribes to stream Stock and publishes an event to another stream T with the same schema. I.e.,

```
156 subscribe s to Stock;
157 behavior { publish('T', s); }
```

The second query is an example of a Cayuga automaton with multiple states that detects a double-top formation in the price chart of any stock – this is a well-known pattern amongst trade analysts, also known as an M-shaped pattern. Fig. 17, for example, shows one of the M-shaped patterns found in the data set. The CEL query that detects such patterns is illustrated in Fig. 18. Certain details of the query have been omitted, while others have been simplified; we summarise its functionality in the following.[4]

Starting from the innermost expression (lines 174-176), Cayuga correlates two consecutive Stock events for the same company until the price rebounds. The engine then attaches a new automaton instance to the next query state, continuously iterating over an *a priori* unknown number of Stock events for the same company, as long as its price monotonically increases. This is achieved by the inner FOLD operator (line 177). Cayuga proceeds outwards in the query, transiting to the next Cayuga automaton state, looking for a monotonic decrease in that stock's price, and so on, spawning new automaton instances along the way and storing valleys and peaks of a given stock's price in attributes A to F of the resulting schema.

[4] The complete queries summarised in Figures 18 and 19 are available at `www.dcs.gla.ac.uk/~koliousa/middleware.html`.

```
158 SELECT Name, A, B, C, D, E, Price as F
    FROM
    FILTER {...} ( # Price increases.
    FILTER {...} ( # Price decreases.
    SELECT Name, A, B, C, D, Price as E
    FROM
    FILTER {...} ( # Price increases.
    SELECT Name, A, B, C, Price as D
    FROM
    FILTER {...} ( # Price decreases.
    SELECT Name, A, B, Price as C
    FROM
    FILTER {...} ( # Price increases.
    SELECT Name, A, Price as B
    FROM
    FILTER {...} ( # Price decreases.
174 SELECT Name, Price as A
    FROM Stocks
176 NEXT {$1.Name=$2.Name} Stock
177 ) FOLD{...} Stock # Price increasing.
    ) FOLD{...} Stock # Price decreasing.
    ) FOLD{...} Stock # Price increasing.
    ) FOLD{...} Stock # Price decreasing.
    ) NEXT {$1.Name = $2.Name} Stock
182 ) PUBLISH T
```

Fig. 18. M-shaped pattern in Cayuga.

```
183 subscribe s to Stock;
    map m;
    identifier id;
    sequence stock;
    real a, ..., f, previous;  # Prices
    bool A, ..., F;            # States
    initialization { m = Map(sequence); }
    behavior {
       id = Identifier(s.id);
       if (! hasEntry(m, id)) {
          # Init states & prices for stock id
          stock = Sequence(s.price, false, ...);
          insert(m, id, stock);
       } else {
          stock = lookup(m, id);
          previous = seqElement(stock, 0);
          # A is the 1st element, B the 2nd, etc.
          # ...
          if (previous < s.price) {
             # Monitor increasing...
          } else if (previous > s.price) {
             # Monitor decreasing...
          }
       }
207 }
```

Fig. 19. M-shaped pattern in GAPL.

```
208 subscribe s to Stock;
    map stocks;
    identifier id;
    sequence tuple, updated;
    initialization { stocks = Map(sequence); }
    behavior {
       id = Identifier(s.id);
       if (! hasEntry(stocks, id)) {
          # New run; set count to 1.
          tuple = Sequence(s.price, s.time, 1,
                s.price, s.time);
          insert(stocks, id, tuple);
          publish('Folded', 'create', s.id,
                tuple);
       } else { # Existing run; update.
          tuple = lookup(stocks, id);
          if (s.price > seqElement(tuple, 3)) {
             # Price is increasing;
             # Incr. count (l. 228)
             updated = Sequence(
                seqElement(tuple, 0),
                seqElement(tuple, 1),
228             seqElement(tuple, 2) + 1,
                s.price, s.time);
             insert(stocks, id, updated);
             publish('Folded', 'change',
                s.id, updated);
          } else {
             remove(stocks, id);
             publish('Folded', 'remove',
                s.id, tuple);
          }
       }
237 }
```

```
238 subscribe f to Folded;
    map m;
    window w;
    identifier id;
    sequence s;
    iterator i;
    initialization { m = Map(window); }
    behavior {
       id = Identifier(f.id);
       if (f.command == 'create') {
          w = Window(sequence, ROWS, 1000);
          s = Sequence(f.price1, f.time1,
                f.count, f.price2, f.time2);
          append(w, s);
          insert(m, id, w);
       } else if (f.command == 'change') {
          w = lookup(m, id);
          s = Sequence(f.price1, f.time1,
                f.count, f.price2, f.time2);
          append(w, s);
       } else { # Removed from map stocks.
          w = lookup(m, id);
          i = Iterator(w);
          while (hasNext(i)) {
             s = next(i);
             publish('T', s);
          }
          remove(m, id);
       }
265 }
```

Fig. 20. Detecting sequences of increasing stock prices.

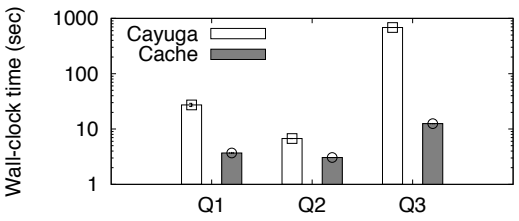

Fig. 21. Benchmarking against Cayuga.

Our implementation (simplified) is shown in Fig. 19. The automaton maintains booleans A to F, together with their associated price values (a to f), in a map of sequences; each entry represents a small state machine for a given stock. Once all states A to F for a stock are true, then the M-shaped pattern has been detected. Depending on the current stock price, the algorithm backtracks to previous states or proceeds to the next (ascending or descending price runs) accordingly. Note here that our solution is algorithmic, using the if then else constructs of GAPL.

The final query is an example use of the FOLD operator to perform aggregate computations across multiple iterations. The CEL query has the form:

```
SELECT *
FROM (
SELECT *, 1 AS counter FROM Stock
) FOLD {$1.Name = $2.Name, $.Price < $2.Price, $.counter + 1 AS counter} Stock
PUBLISH T
```

The desired behaviour is to detect continuous runs of increasing prices for each stock, and to display the sequence of events that constituted each run. This has been implemented using the two Glasgow automata illustrated in Fig. 20. The automaton on the left maintains a map entry (a sequence) for a given company (s.id) as long as its price is monotonically increasing. At the end of each run, i.e., when the stock's price decreases, the sequence contains the current lowest price, the start time, the length of the run, and current highest price, and the end time. In order to also display the sequence of events that constitute each run, we publish to another topic, Folded, and implement a custom state management system – with states create, change, and remove – in the automaton on the right.

Fig. 21 shows the results from comparing the execution time of the three CEL queries with their equivalent implementations in the automaton programming language. The Cayuga engine best compiles in Microsoft's Visual Studio, thus the experiments were run on a Windows platform. 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.

For the first query, the performance improvement against Cayuga is an order of magnitude. This strengthens the argument of the efficacy of the automaton execution model and the efficient unification of the publish/subscribe infrastructure with the data stream management code. For the second query, the automaton detects the pattern twice as fast as Cayuga. This is another example of the ability to implement multiple state machines under a single execution thread, which contributes to this performance enhancement. Finally, for the third query, our implementation is dramatically faster ($\times 50$) than Cayuga's equivalent.

7 The impact on complex event processing

We have proposed an algorithmic, imperative approach to complex event processing. The first impact of this work is Turing completeness, something that was proposed from a theoretical basis in [3]. Some event languages have a solid background in event calculi and result in one-line expressions that are compact implementations, but are not open to user optimizations – apart from physical query execution 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 in GAPL.

Our experiences thus far are that the imperative programming style of GAPL enables it to be used in many domains: home network management [4], industrial applications [20], and, given the present comparisons to Cayuga, stocks. In addition to its expressiveness, this imperative model has been shown more performant than a declarative event language (CEL). Thus, it may be viewed as an intermediate language between SQL-like queries (logical query plans) and their execution interpretation (physical query plans).

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 [22]. We continue our comparative endeavours with the Linear Road Benchmark [23].

References

1. Codd, E.F.: A relational model of data for large shared data banks. Commun. ACM **13** (1970) 377–387

2. Terry, D., Goldberg, D., Nichols, D., Oki, B.: Continuous queries over append-only databases. In: Proceedings of the ACM SIGMOD. (1992)
3. Law, Y.N., Wang, H., Zaniolo, C.: Query languages and data models for database sequences and data streams. In: Proceedings of the VLDB. (2004)
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: Proceedings of the IFIP/IEEE IM. (2011)
5. Arasu, A., Babu, S., Widom, J.: The CQL continuous query language: semantic foundations and query execution. The VLDB Journal **15** (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** (2003) 120–139
7. Demers, A., Gehrke, J., Hong, M., Riedewald, M., White, W.: Towards expressive publish/subscribe systems. In: Proceedings of the EDBT. (2006)
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 ACM SIGMOD. (2007)
9. Agrawal, J., Diao, Y., Gyllstrom, D., Immerman, N.: Efficient pattern matching over event streams. In: Proceedings of the ACM SIGMOD. (2008)
10. Wu, E., Diao, Y., Rizvi, S.: High-performance complex event processing over streams. In: Proceedings of the ACM SIGMOD. (2006)
11. Gyllstrom, D., Agrawal, J., Diao, Y., Immerman, N.: On supporting kleene closure over event streams. In: Proceedings of the IEEE ICDE. (2008)
12. 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) (2010) 150–161
13. White, W., Riedewald, M., Gehrke, J., Demers, A.: What is "next" in event processing? In: Proceedings of the ACM PODS. (2007)
14. Sullivan, M., Heybey, A.: Tribeca: a system for managing large databases of network traffic. In: Proceedings of the USENIX ATEC. (1998)
15. Cranor, C., Johnson, T., Spataschek, O., Shkapenyuk, V.: Gigascope: a stream database for network applications. In: Proceedings of the ACM SIGMOD. (2003)
16. Cormode, G., Hadjieleftheriou, M.: Finding the frequent items in streams of data. Commun. ACM **52** (2009) 97–105
17. Sekar, V., Reiter, M.K., Zhang, H.: Revisiting the case for a minimalist approach for network flow monitoring. In: Proceedings of ACM IMC. (2010)
18. Kandula, S., Chandra, R., Katabi, D.: What's going on?: learning communication rules in edge networks. In: Proceedings of the ACM SIGCOMM. (2008)
19. Johnson, T., Muthukrishnan, S., Shkapenyuk, V., Spatscheck, O.: A heartbeat mechanism and its application in Gigascope. In: Proceedings of the VLDB. (2005)
20. Koliousis, A., Sventek, J.: DEBS Grand Challenge: Glasgow automata illustrated. In: Proceedings of the ACM DEBS. (2012)
21. Diao, Y., Altinel, M., Franklin, M.J., Zhang, H., Fischer, P.: Path sharing and predicate evaluation for high-performance XML filtering. ACM Trans. Database Syst. **28**(4) (2003) 467–516
22. Zeitler, E., Risch, T.: Massive scale-out of expensive continuous queries. PVLDB **4**(11) (2011) 1181–1188
23. 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 VLDB. (2004)