# MavEStream: Synergistic Integration of Stream and Event Processing *

Qingchun Jiang
ANTs Software Inc.
Burlingame, CA 94401
dustin.jiang@ants.com

Raman Adaikkalavan
CIS Department
IU South Bend
raman@cs.iusb.edu

Sharma Chakravarthy
CSE Department
UT Arlington
sharma@cse.uta.edu

## Abstract

*Although research seems to address event and stream data processing as two separate topics, there are a number of similarities between them. For many advanced stream applications, both event and rule processing are needed and are not currently well-supported. Extant event processing systems concentrate primarily on complex events and rules and stream processing systems concentrate on stream operators, scheduling, and quality of service issues. Synergistic integration of these models will be better than the sum of its parts.*

*We propose an integrated model to combine the capabilities of both models for applications that need both of them. Specifically, we introduced a number of enhancements, including* stream modifiers*,* semantic windows*,* event generators*, and* enhanced event and rule specifications*, to couple two models seamlessly. We prototype our integrated system using the stream processing system (MavStream) with the event processing system (Snoop and Sentinel) and discuss the design and implementation issues of our prototype.*

## 1 INTRODUCTION

Event processing [8, 12, 17, 18, 21, 27, 33] and lately stream data processing [1, 6, 14, 30, 34, 36] have evolved independently based on situation monitoring application needs. Several event specification languages [2, 13, 21, 22, 25] for specifying composite events have been proposed and triggers have been successfully incorporated into relational databases. Different computation models [8,12,18–21,23,33] for processing events, such as Petri nets [20,21], extended automata [23,33], and event graphs [8, 12, 19] – have been proposed and implemented. Various event consumption modes [8, 12, 13, 20, 21] (or parameter contexts) have been explored. Similarly, stream data processing has received a lot of attention lately, and a number of issues – from architecture [1, 14, 30, 34, 37] to scheduling [5, 9, 31] to Quality-Of-Service (QoS) management [4, 15, 39] – have been explored. Although both of these topics seem to be different on the face of it, based on the applications, such as network fault management systems [29], that we have analyzed, they augment/complement each other in terms of computational needs of real-world applications. As it turns out, the computation model used for stream data processing (data flow model) is not very dissimilar from some of the event processing models (e.g., event graph), but developed with a different emphasis.

Current event processing systems assume that primitive events are domain specific and are assumed to be detected by the underlying system such as a database, operating system (OS), or an application. Treating the output of complex computations in the form of a stream query processing as events have not been addressed. Furthermore, combining these two computational models - one window based and the other non-window based (but using event consumption modes or parameter contexts) requires enhancements to *both models*. This paper addresses synergistic integration of these two strands of research and development into a more expressive and powerful model of computation.

In this paper we first analyze both threads of work with respect to a number of characteristics in Section 2. We propose a framework in Section 3 that combines the two using a uniform computation model and address design issues for their coupling. In section 4, we integrate the current window types in stream processing and parameter contexts in event processing into our semantic window, which not only significantly extends the current window concept for CQs, but also overcomes the limitations of parameter contexts currently used in event processing, in Section 4. The concept of semantic window first introduced in [29] is being developed further to integrate the two models. Finally, we discuss the design and implementation of the prototype system in Section 5. Section 6 has related work. Conclusions and future directions are discussed in Section 7.

## 2 Analysis of Event and Stream Processing Models

Processing of events using event detection graphs and a data flow architecture is similar to the processing of data streams. In this section, we analyze the characteristics of event and data stream processing models and analyze their

*This work was supported, in part, by NSF grants IIS-0534611, MRI-0421282, ITR-0121297, EIA-0216500, and ANTs Software Inc.

similarities and differences. This will form the basis of our integrated model and for identifying the extensions needed for each model.

**Inputs and Outputs:** Inputs to an event processing model are a sequence of events (or an event history) usually ordered by their time of occurrence. Most event sequences considered by event processing models are generated by a DBMS, system clock, etc. The input rate of an event sequence is not assumed to be very high or even bursty [1]. The outputs of event operators also form event sequences, which are ordered by their occurrence timestamps.

Inputs to the data stream applications are data streams. Input tuples in a data stream can be ordered by an attribute and not necessarily by a timestamp (e.g., sequence_id of a TCP packet in a TCP packet stream) as in the case of event sequences. In addition to streams, a data stream processing model can also include processing of static relations, which are not typically supported in an event processing model (although conditions and actions can access stored relations). Furthermore, the input characteristics of data streams are assumed to be highly unpredictable and dynamic (e.g., bursty). The output of a data stream processing model is a data stream as well. Thus, conceptually, both the models have similar inputs and outputs. However, The data sources in data stream processing model are highly bursty external raw data streams, which usually are not end-user interested primitive event streams, though primitive events can be computed from the raw data streams. While the data sources in event processing model are mostly internal ones and mainly assumed primitive event streams with low input rates.

Current stream models do not have appropriate operators to convert stream outputs into more meaningful primitive events, which usually are application specific. We introduced *stream modifiers* [29], along with our *event generators*, which will be discussed in Section 3.3, to convert output of CQs into primitive events, as inputs of our event processing model.

**Consumption Modes Vs. Windows:** Event consumption modes or contexts [13, 21] were introduced primarily to reduce the number of event instances that should be kept for the purpose of detecting composite events. The number of events to be kept depended solely on the context and the semantics of the operator. Event contexts facilitate in keeping a small portion of an event sequence based solely on the value of timestamp and the context. In contrast, the notion of a window in stream processing is defined on each stream and does not depend upon the operator semantics. Also, the window need not be defined only in terms of either time or physical number of tuples, although that is typical in most of the applications. Hence, the window generated by a context is not the same as the window concept in streams.

Our introduction of the concept of a *semantic window* is meant to integrate current window types in stream processing and parameter contexts in event processing and to enhance current stream window types and parameter contexts, which is being explored further to generalize the notion of consumption modes or contexts in semantic windows.

**Event Operators Vs. CQ Operators:** Event operators are quite different from the operators supported by current stream processing models. Event operators are mainly used to express and define the computation on events (or event objects) and to reduce the number of output events through consumption modes, and they solely use the timestamp of an event for detecting composite events. *Thus, event operators do not perform any computation over the attributes of events.* On the other hand, current stream processing operators are mostly modified relational operators which focus on how to express and define the computation on the tuple attributes. Additionally, stream processing operators have input queues to deal with highly bursty inputs and synopsis to perform window-based computation. Therefore both types of operators are necessary in out integrated model.

**Computation Model:** Computations in event processing models are decomposed into three main components, which correspond to each component of an Event-Condition-Action rule : 1) timestamp correlations performed at the event level carried out by the event operators, 2) computations performed at the attribute/parameter level, which are carried out by the condition checks, and 3) computations for processing rules (triggering actions). In some event processing systems, a part of the computations that are carried out for primitive events at the attribute level may be moved to the event detection component (notion of masks proposed in [25]) for improving performance. A different mask notation for composite events is introduced in this paper. Computations in stream processing models are not clearly partitioned into different components as in the case of event processing. However, considering the functionalities, computations in stream processing models can be viewed as two components: *operator computation* and *window computation*. The former involves complicated condition checking and attribute level manipulations, and the latter is required to maintain a snapshot of tuples or status information (synopsis) for blocking operator computations.

Thus, computations performed in stream processing and event processing serve different purposes and are different. Although some event operators in some consumption modes can be modeled as join (SEQUENCE mainly) with some difficulty, other operators (e.g., NOT, Periodic, aperiodic) and other consumption modes (e.g., continuous, chronicle) cannot be modeled. Moreover, the semantics and their role are different too.

**Best-Effort Vs. QoS:** The notion of QoS is not present in the event processing literature. Although, there is some work on real-time events and event showers [7], event processing models do not support any specific QoS requirements. Typically, in the event processing model, whenever an event occurs it

[1] Publish&subscribe systems such as Le Subscribe and XML message filters [16] are considered as stream processing rather than event processing since their processing is more close to stream processing model.

is detected or propagated to form a composite event. Thus, events are detected based on the best-effort method. On the other hand, QoS support in a data stream processing model is necessary and critical to the success of data stream management systems (DSMSs). A large body of work addresses various QoS requirements such as tuple latency, memory usage, and throughput.

**Optimization and Scheduling:** Event expressions can be represented as event graphs for detection. Common event sub-expressions are grouped in order to reduce memory usage, overall response time, and computation effort. In general, event processing does not deal with runtime optimizations. On the other hand, efficient approaches for processing CQs are important to a DSMS. The concept of queues and windows in a DSMS introduce even more challenges and opportunities for query optimization. Optimizations in stream processing model include: 1) sharing of queues (inputs) among multiple operators, 2) sharing of windows (synopsis), 3) sharing of operator computations, and 4) sharing of common sub-expressions. The notion of scheduling is also absent from event processing systems. Typically a data-flow architecture (implicitly, a First-In-First-Out scheduling strategy is employed in event processing models) is assumed as indicated earlier and memory usage or event-latency has not been addressed in the literature. On the other hand, optimizing memory capacity, tuple latency, and the combination of the two have prompted many scheduling algorithms [5, 9, 31] in stream processing.

**Buffers and Load Shedding:** None of the event processing systems assume the presence of queues between event operators. Events were assumed to be processed as soon as they are detected (not necessarily occurred) and partial results are maintained in event nodes. Unlike stream processing, most of the event processing models assume that the incoming events are not bursty and hence usually do not provide buffer management or explicit load shedding strategies. Event consumption modes can be loosely interpreted as load shedding, used from a semantics viewpoint rather than QoS viewpoint. On the other hand, load shedding is extremely important in a stream processing environment. Even with the choice of the best scheduling strategy, it is imperative to have load shedding strategies as the input rates can vary dramatically. Several load shedding strategies, placement of load shedders, and the amount of tuples to be shed (possibly limiting the error in query results) have been proposed [4, 15, 39].

**Rule Processing:** Extant event processing systems support dynamic enabling/disabling of rules. On the other hand, rule execution semantics specifies how the set of rules should behave in the systems once they have been defined. A rich set of rule execution semantics [10, 40] have been proposed to accurately define and efficiently execute rules in the literature for event processing models. Those semantics include rule processing granularity, rule execution (instance/set, iterative/recursive, and sequential/concurrent), conflict resolution, coupling modes, and termination. Stream processing systems do not support rule specification and processing, which are critical to many real-world applications.

With our extensions to the event processing proposed in this paper, current rule processing techniques have to extended as well [28]. We do not explore this further in this paper due to lack of space.

**Summary:** Although two models share similar inputs and processing models, there are a number of differences among operator semantics, contexts, processing requirements, and emphases/purposes on inputs/outputs, computation, and final goals. Eventually, a number of extensions have to be made on the event processing side to accommodate high input rates and QoS requirements. Some extensions need to be made on the stream side to generate events (changes/values of interest) which are less in number as compared to stream output and are meaningful. In this paper, we are taking a small step (Please refer to [28] for more details) in the form of extending the window concept with a more powerful semantic window to integrate two computation models. Also, we have provided constructs for defining streams as events, event masks, and addressed architectural issues of coupling the two models. We have proposed stream modifiers and are currently working on applying semantic window in lieu of contexts, queues and scheduling for events.

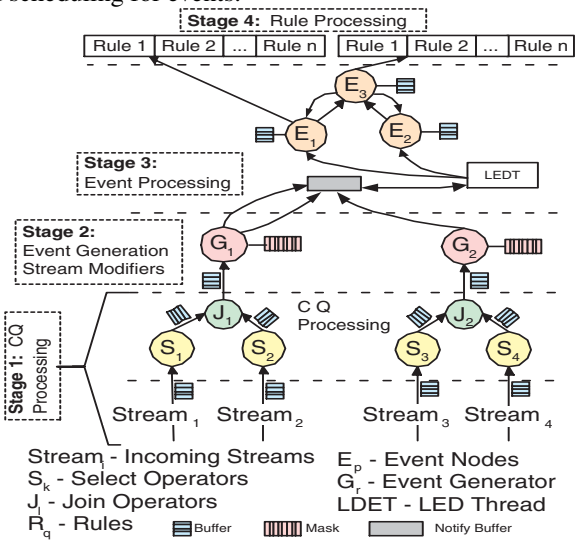


**Figure 1. MavEStream: Four Stage Integration Model**

## 3 MavEStream: An Integrated Model

The proposed integrated model, termed MavEStream is shown in Figure 1 and it consists of four stages: 1) CQ processing stage used for computing CQs over data streams, 2) coupling stream output with event processing system, 3) event processing stage that is used for detecting events, and 4) rule processing stage that is used to check conditions, and

to trigger predefined actions once events are detected. The seamless nature of our integrated model is due to a number of extensions discussed in this paper and the compatibility of the chosen event processing model [2] (i.e., an event detection graph) with the model used for stream processing.

### 3.1 Continuous Query Processing

This stage processes normal CQs where it takes streams as inputs and gives computed continuous streams. The scheduling algorithms and QoS delivery mechanisms (i.e., load shedding techniques) along with other techniques developed for stream processing model can be applied directly. In many cases, final results of stream computations need to be viewed as interested data points where primitive events can be computed for defining situations that use multiple streams and composite events. A CQ may give rise to multiple events based on the attribute values of the output stream. In Figure 1, operators $S_1$, $S_2$, and $J_1$ form a CQ. Similarly, operators $S_3$, $S_4$, and $J_2$ form a CQ.

```
CREATE CQ CQName AS (Normal CQ statements)
```

**Named Continuous Queries:** In order to express computations clearly, CQs are named. The name of a CQ is analogous to the name of a table in a DBMS and a named CQ has the same scope and usage as a table. The queue (buffer) associated with each operator in a CQ supports the output of a named CQ to be fed into the input queue of another named CQ. A named CQ is defined by using the CREATE CQ statement shown above. However, the FROM clause in a named CQ can use any previously defined CQs through their unique names. Events can be specified by using named CQs and in addition provide conditions on attributes to generate multiple event types.

### 3.2 Event Processing

Below we discuss two limitations of current event processing systems. Event detection graphs (or EDGs) in the current event processing systems do not have input queues/buffer for event operators as the input rate of an event stream is not assumed to be very high and highly bursty. Thus, in our integrated model, input queues/buffers are added to event operator nodes (shown in Figure 1) to handle the highly bursty input generated by the CQs from the CQ processing stage.

In a traditional event processing system, primitive events can be either class or instance level, but both of them are based on *timestamps*. Instance level events play an important role for events generated by stream processing, but with the dynamic nature of incoming streams it is difficult or *impossible* to determine the instance level events ahead of time [28]. Even if the values are predefined, they require large number of event nodes, which introduce high computation and memory overhead. Hence, event processing needs to be burdened less to support *efficient* detection. We have generalized event expression computation, so that attribute conditions are checked *before* the events are detected. This generalized expression allows both primitive and composite event nodes to detect events based on MASKS or attribute-based constraints. MASKS are pushed to the event generator node with primitive events (i.e., for leaf nodes in EDG) and are pushed into the event operator nodes (i.e., internal nodes in EDG) for other events. For instance, in Figure 1 MASKS corresponding to CQ with $J_1$ as the root node is pushed to event generator node $G_1$. Thus, when multiple events are defined on the same CQ but with different MASKS, all of them are pushed to the corresponding event generator node.

```
CREATE EVENT   E_name
       SELECT  A_1, A_2, ..., A_n
         MASK  Conditions
         FROM  E_S | E_X
```

Users can specify events based on `CQs` (for primitive events) or on `Events` using the CREATE EVENT statement shown above.

- CREATE EVENT creates a named event $\mathcal{E}_{name}$
- SELECT selects attributes $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_n$
- MASK applies conditions on the attributes
- $\mathcal{E}_S$ is a named `CQ` or a CREATE CQ statement
- $\mathcal{E}_X$ is an event expression that combines more than one event using event operators

### 3.3 Coupling Event and Stream Processing

The local event detector (LED) has a common *notify buffer* (or event processor buffer) into which all events that are raised are queued. A single queue is necessary as events are detected and raised by different components of the system (CQs in this case) and they need to be processed using their time of occurrence. We will discuss the design choices in section 5, but briefly, a new operator is added to every stream query at the root if an event is associated with that CQ. This operator can take any number of MASKS and for each MASK, a different event tuple/object is created and sent to the notify buffer. This operator is activated only when an ECA rule associated with that CQ is enabled. This operator is similar to the select operator except that when it generates an event, it invokes an API of LED to queue that event in the notify buffer. CQs output data streams in the form of tuples and *event generator* operator nodes are attached to the root node of the CQ. As shown in Figure 1, nodes $J_1$ and $J_2$ are attached to event generator nodes $G_1$ and $G_2$. In addition, nodes $G_1$ and $G_2$ are also associated with MASKS. Thus, stream tuples from $J_1$ and $J_2$ are converted to events by nodes $G_1$ and $G_2$.

We have also added *stream modifiers* to detect expressive changes (compute primitive events) between tuples in a stream and to further reduce the number of tuples before they are sent to the event generator. A stream modifier is defined as a function to compute the changes (i.e., relative change of an

[2] It will be difficult to integrate either the Petri net event processing model of SAMOS or the extended automata model of ODE with stream processing models

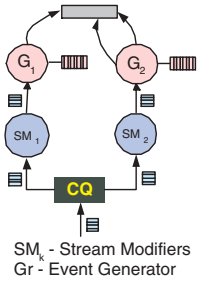


**Figure 2. Stream Modifiers**

attribute) between two consecutive tuples of its input stream. It is denoted by

$$M(< t_1, t_2, ..., t_i >[, P < pseudo >][, O|N < v_1, v_2, ..., v_j >])$$

where $M$ is the modifier function that computes a particular kind of change. The i-tuple $< t_1, t_2, \cdots, t_i >$ is the parameter required by the modifier function $M$. The following $P < pseudo >$ defines a pseudo value for the $M$ function in order to prevent underflow. The following j-tuple element is called the untouched attribute that needs to be output without any change. The $O|N$ part is called modifier profile, which determines whether the oldest values or the latest values of the j-tuple that needs to be output. If $O$ is specified, the oldest values are output or the latest values are output if $N$ is specified. Both untouched attributes and modifier profile are optional. A family of stream modifiers specific to applications can be defined using the above definitions. As shown in Figure 2, output stream of a CQ is given as input to stream modifiers $SM_1$ and $SM_2$. Once the change computations are performed, stream modifiers give their output as inputs to the event generator. Some of the stream modifiers [28] are; **ADiff()** to detect absolute changes over two consecutive states, **RDiff()** to detect relative changes over two consecutive states, and **ASlope()** to compute the slope ratio of two attributes over two consecutive states.

## 4 Semantic Window

In current stream processing models the concept of a window defines a historical snapshot of a finite portion of a stream at any time point. It defines the meaningful set of data used by an operator. Accurate window definition not only impacts the accuracy of final results, but also has a significant impact on system performance. A few window types are wide used in data stream processing. They are 1) *Time-based window*, windows are simply expressed in number of time units; 2) *Tuple-based window*, window are simply expressed in number of most latest tuples; 3) *attribute-based window*, windows are expressed based on the value of a specific attribute [1]; and 4) *partitioned sliding window*, in which inputs are logically partitioned into different sub-streams and computations are performed over each sub-stream, similar with GROUP BY in SELECT.

Although the basic window types thereafter, the above mentioned 4 types of windows are referred as basic windows.) are useful for many applications, they are still limited by their ability to express more meaningful windows (i.e., based on semantic information) required by applications. The functionality of a stream operator does not change when it is applied to different applications. On the contrary, its window specification changes as the applications change in order to compute results correctly and efficiently. We call the functionality of an operator as its `global property` and the window property of an operator as its `local property`. Computing a meaningful and accurate window under different application domains is not trivial as it requires a general-purpose format to express the window concept than what is currently available [3]. The window itself can be based on a *computation*, which is used to determine a meaningful snapshot of a portion of an input stream for its respective operator. The **semantic window**, elaborated in this paper, can express more complicated and meaningful windows required by applications than the basic windows. In addition, semantic window computations can be carried out through well-developed and highly optimized SQL query processing engines.

The main function of a semantic window is to determine which tuples should be in the current window by performing deletion of existing tuples and addition of new tuples. Before defining a semantic window, we need to identify the scope of data that a semantic window can access to perform computations. Obviously, all tuples in the current window, new tuples, and static relations, if needed, are fully accessible by that window. All data accessible by a semantic window are referred as *semantic window input data* $(SWID)$.

**Definition 1 (Semantic Window)** *a semantic window is defined as a finite portion of historical tuples at any time point, which satisfy a* semantic window condition $(SWC)$*, by computing $\mathcal{SWC}$ over $\mathcal{SWID}$.*

The $\mathcal{SWC}$ can be any arbitrary condition over $\mathcal{SWID}$. However, to simplify the way to express a $\mathcal{SWC}$, we use the CHECK statement shown below:

```
Stream [      CHECK  logical_Expression
            SELECT   a1, a2, ..., an
              FROM   SWID
             WHERE   Conditions
          GROUP BY   Attributes
            HAVING   Conditions   ]
```

All the clauses used in the above statement (i.e., SELECT, FROM, WHERE, GROUP BY, and HAVING) have the same semantics and usage as in the standard SQL. However, only $\mathcal{SWID}$, include all tuples in the current window, new tuples, and static relations can appear in FROM clause. $a_1, a_2, \ldots, a_n$ are the attribute names (or alias after applying

[3]The computation required to maintain a window is not discussed in detail in the literature.

aggregate functions) from $\mathcal{SWID}$. The CHECK clause is a logical expression that consists of the attribute names used in the SELECT statement, relational and logical operators, and parentheses. The CHECK clause is the last clause to be evaluated in the SQL statement and returns a Boolean value. The CHECK clause requires only one row from the FROM clause after applying other clauses. If more than one row is returned, only the oldest one in current window is used to evaluate the CHECK clause.

Semantic window can be expressed using current SQL statements (i.e., SELECT-FROM-WHERE statement) over $\mathcal{SWID}$ with little effort needed for its implementation. We can also take advantage of the well-developed SQL query processing engine and its optimization techniques[4] provided in stream processing model to compute semantic windows with little effort to modify current data stream systems.

Efficient implementations of a semantic window are still one of our challenging research topics in our integrated model considering the highly-bursty input characteristics. Currently, we implemented semantic windows using Algorithm 1. In this algorithm, we add a new tuple [5] referred as $\mathcal{NT}$, to current window $\mathcal{CW}$, then delete oldest tuples, if necessary, to make the condition $\mathcal{SMC}$ to be true.

**Algorithm 1**: Add New Tuple Algorithm

**INPUT**: $(\mathcal{CW}, \mathcal{NT})$;
**OUTPUT**: Modified $\mathcal{CW}$;
**if** *$\mathcal{CW}$ is empty or not initialized* **then**
  // append the new tuple to current window
  Appended Window $\mathcal{AW} \leftarrow \mathcal{NT} + \mathcal{CW}$;
  **return** *$\mathcal{AW}$ as the new $\mathcal{CW}$;*
**else**
  Appended Window $\mathcal{AW} \leftarrow \mathcal{NT} + \mathcal{CW}$;
  **while** *((the condition $\mathcal{SWC}$ over the $\mathcal{AW}$ is not TRUE) AND ($\mathcal{AW}$ is NOT empty))* **do**
    Delete the oldest tuple in the $\mathcal{AW}$;
  **end**
  **return** *$\mathcal{AW}$ as the new $\mathcal{CW}$;*
**end**

Our current implementation of semantic window is not the most efficient, but a straightforward one that needs a minimal implementation effort. Currently, we are more interested in the whole integrated system, rather than individual components, though more efficient algorithms for individual components including semantic window are being explored.

The semantic window expressed using a GROUP BY is similar to the partitioned window type introduced in [3]. However, the condition that maintains each logical sub-window is more meaningful and powerful than a simple condition on the number of rows. When a GROUP BY clause is used in the $\mathcal{SWC}$ definition, a semantic window is logically split into a number of sub-windows based on the attributes in the GROUP BY clause, and the $\mathcal{SWC}$ is applied to each sub-window. Each logical sub-window is labeled by the value(s) of GROUP BY attributes. The new tuple is added only to the logical sub-window whose label matches the corresponding values of GROUP BY attributes in the new tuple. Since a new tuple is only added to one logical sub-window, only that logical sub-window is evaluated (actually, evaluation results for all other sub-window are always TRUE because they are not changed). When a stream operator computes over its semantic window, the computation is only on the sub-windows that have been changed (i.e., add new tuples or delete old tuples).

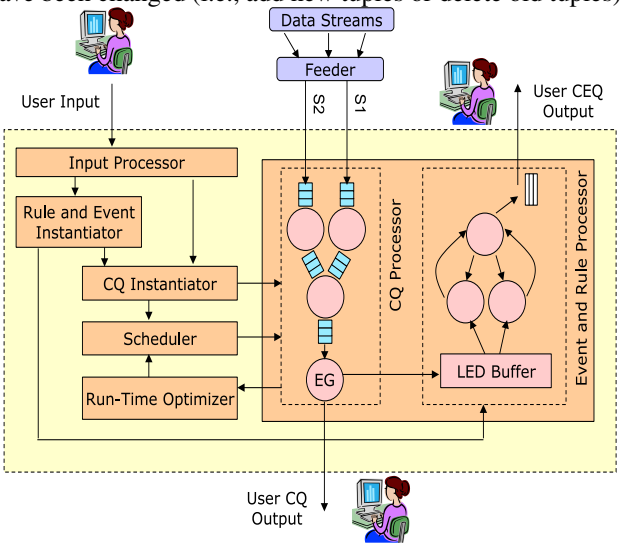


**Figure 3. MavEstream Architecture**

## 5 Prototype Implementation

MavEstream is implemented by integrating MavStream (a stream processing system) [26, 30] and Local Event Detector (LED, an event processing system) [2, 12]. Both the systems are homegrown and are implemented in Java. One decision was whether to have a single address space or different address spaces. We chose to have them in the same address space as inter-process communication would add significant overhead. MavEstream, as illustrated in Figure 3 consists of an Continuous Event Query (CEQ) input processor, a rule and event instantiator, a CQ instantiator, a query/operator scheduler, a CQ processor, event generator, a rule and event processor, a runtime optimizer and a load shedder.

The user submits the CEQ, which is parsed and validated by the input processor. The CQ and the event part are separated and given to the appropriate instantiator. Rule and event instantiator propagates the event generator part to the CQ instantiator. The CQ instantiator generates the query graph for the CQ, stream modifiers (if any) and also associates an event generator with each query. Semantic windows for CQ and event operators, if any, are implemented as discussed in Al-

[4] We do not elaborate on implementation of semantic window and its optimization as SQL-based optimizations are well-known and can be directly used/adapted.

[5] A tuple is not necessary an individual tuple, it could be a page of individual tuples.

gorithm 1 currently. The event detection graph is generated and the rules are defined on the event nodes by the rule and event instantiator. At runtime the event generator (as shown in Figure 1) is responsible for raising the events which are then processed by the event processor. For each event detected, the rules defined on it are triggered by checking the conditions and executing the actions if the corresponding conditions evaluate to true.

Event generation is accomplished by implementing an operator connected to the root of a CQ or a stream modifier. The benefit of such an implementation is that since the event generator is associated with every CQ, it is executed whenever CQ is scheduled and it does not continuously execute as a worker thread, thus saving system resources. Currently, we have implemented a family of stream modifiers with both modifier profiles and in both windowed and non-windowed versions. The event generator has references to primitive events associated with the query and can raise them when they are detected. It constructs an event object, which contains the information associated with an event such as the event name, time of event occurrence etc., and inserts it into a common event processor buffer from which the LED removes events in the order they arrive and raises the event by pushing it into a primitive event node. The tuple attributes are inserted in the event object as a list of parameters. Instead of a common buffer, an alternative approach is to have a producer/consumer buffer for each CQ from which the event processor would directly consume events. This would mean that the order of event occurrence and the order of event consumption may not be the same which is critical for event processing.

The decision to incorporate the MASK specified with primitive events with the CQ or the event processing side would effect the performance of the system. Filtering events as early as possible avoids buffering and processing them after its generation. Hence we decided to push the MASK to the level of the CQ and check the MASKS associated with a primitive events in the event generator. The implementation uses a *FESI ECMA* condition evaluator. Only the tuples which satisfy the MASK are sent to the event side and thus the load in the event processor is reduced. The MASKS on the composite events are checked at the composite event or operator node. This allows us to separate event types by specifying an arbitrary condition on the stream output. The issues discussed in Section 3.2 are handled using this approach.

## 6 Related Work

Our paper is directly related to a set of papers [1, 6, 14, 30, 34, 36], which mainly focus on the system architecture, CQ execution (i.e., scheduling and various non-blocking join algorithms), and QoS delivery mechanisms for stream processing. The main computations over stream data are limited to the computation of relational operators over high-speed streaming data, and the event and rule processing and the extensions to CQs to enhance their expressive power and computation efficiency are rarely discussed. The paper [3] proposed a CQ language for CQs, and provide formal expression for primarily sliding windows. The intent of this paper is not to provide a complete CQ language. Instead we propose a formal and meaningful extensions to express a much richer set of computations, which can be used to enhance current CQ languages without changing their syntax and the overall computation model.

Our paper is also closely related to a set of papers [1,3,42] that try to enhance the expressiveness power of SQL in a data stream environment. Carlo Zaniolo and et al [42] tried to enhance the expressive power of SQL over the combination of relation streams and XML streams by introducing new operators (e.g. continuous UDAs) and supporting sequences queries (e.g., to search for patterns). A. Arasu and J. Widom [3] proposed an enhanced SQL, termed CQL, to instantiate the abstract semantics and to map from streams to relations. However, the notion of semantic window proposed in this paper enhances the expressive power of SQL over streams and improve the computation efficiency through accurate definition of a window, In [1], the window concept is based on arbitrary attribute, not only size on time or the number of tuples, However, the window is still a static window (by specifying the fixed size of the window) and only one attribute can be used to define a window. The semantic window proposed in this paper can be used to define window based on meaningful information, rather than the size.

A number of sensor database projects, Cougar [41], TinyDB [35] have also tried to integrate the event processing with query processing under a sensor database environment. However, the event-driven queries proposed in TinyDB is used to activate queries based on events from underlying operating systems. Our focus in this paper is to process large number of high volume and highly dynamic event streams from CQ processing stage for the applications that needs complex event processing and CPU-intensive computation (i.e., CQs) for generating events. Finally, our paper is related to a large body of work on event detection and rule processing [8, 11–13, 17, 19, 21, 24, 32, 33, 38].

## 7 Conclusions and Future Work

In this paper, we argued for keeping the semantics of event and stream processing systems intact and integrating them *synergistically* to provide an end-to-end system for advanced applications. Our goal was to provide an integrated model for advanced stream applications that supports not only stream processing, but also complicated event and rule processing. We analyzed the similarities and differences between the stream processing and the event processing models and identified a number of enhancements needed. We elaborated on coupling the two systems, specifying ECA rules where the events are generated by CQs and elaborated on the generalization of the window concept in the form of a semantic window as a first step. By using masks and stream modifiers, it is

easier to specify expressive rules. We are currently working on efficient algorithms for individual components such as semantic window, event detection, event generator, and so on, and other aspects of the integration.

## References


[1] D. Abadi et al. Aurora: A New Model and Architecture for Data Stream Management. *VLDB Journal*, 12(2), Aug. 2003.

[2] R. Adaikkalavan and S. Chakravarthy. SnoopIB: Interval-Based Event Specification and Detection for Active Databases. *DKE*, 59(1):139–165, Oct. 2006.

[3] A. Arasu, S. Babu, and J. Widom. The CQL Continuous Query Language: Semantic Foundations and Query Execution. *Stanford Technical Report*, Oct. 2003.

[4] B. Babcock, M. Datar, and R. Motwani. Load Shedding for Aggregation Queries over Data Streams. In *Proc. of ICDE*, Mar. 2004.

[5] B. Babcok et al. Operator scheduling in data stream systems. *The VLDB J.*, 13:333–353, 2004.

[6] S. Babu and J. Widom. Continuous Queries over Data Streams. In *ACM SIGMOD RECORD*, Sep. 2001.

[7] R. Birgisson, J. Mellin, and S. F. Andler. Bounds on Test Effort for Event-Triggered Real-Time Systems. In *International Workshop on Real-Time Computing and Applications Symposium*, 1999.

[8] A. P. Buchmann et al. *Rules in an Open System: The REACH Rule System*. Rules in Database Systems, 1993.

[9] D. Carney et al. Operator Scheduling in a Data Stream Manager. In *Proc. of VLDB*, Sep. 2003.

[10] S. Chakravarthy et al. HiPAC: A Research Project in Active, Time-Constrained Database Management (Final Report). Technical Report XAIT-89-02, Xerox Advanced Information Technology, Cambridge, MA, Aug. 1989.

[11] S. Chakravarthy et al. Composite Events for Active Databases: Semantics, Contexts, and Detection. In *Proc. of VLDB*, pages 606–617, 1994.

[12] S. Chakravarthy et al. Design of Sentinel: An Object-Oriented DBMS with Event-Based Rules. *Information and Software Technology*, 36(9):559–568, 1994.

[13] S. Chakravarthy and D. Mishra. Snoop: An Expressive Event Specification Language for Active Databases. *Data and Knowledge Engineering*, 14(10):1–26, Oct. 1994.

[14] J. Chen et al. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *Proc. of SIGMOD*, 2000.

[15] A. Das, J. Gehrke, and M. Riedewald. Approximate Join Processing over Data Streams. In *Proc. of SIGMOD*, 2003.

[16] Y. Diao and M. J. Franklin. High-Performance XML Filtering: An Overview of YFilter. *IEEE Data Engineering Bulletin*, March 2003.

[17] O. Diaz, N. Paton, and P. Gray. Rule Management in Object-Oriented Databases: A Unified Approach. In *Proc. of VLDB*, Sep. 1991.

[18] A. Dinn, M. H. Williams, and N. W. Paton. ROCK & ROLL: A Deductive Object-Oriented Database with Active and Spatial Extensions. In *Proc. of ICDE*, 1997.

[19] H. Engstrom, M. Berndtsson, and B. Lings. Acood essentials. Technical report, University of Skovde, 1997.

[20] S. Gatziu and K. R. Dittrich. SAMOS: An Active, Object-Oriented Database System. *IEEE Quarterly Bulletin on Data Engineering*, 15(1-4):23–26, Dec. 1992.

[21] S. Gatziu and K. R. Dittrich. Events in an Object-Oriented Database System. In *Proceedings of Rules in Database Systems*, Sep. 1993.

[22] S. Gatziu and K. R. Dittrich. Detecting Composite Events in Active Databases using Petri Nets. In *Proceedings of Workshop on Research Issues in Data Engineering*, Feb. 1994.

[23] N. H. Gehani and H. V. Jagadish. Ode as an Active Database: Constraints and Triggers. In *Proc. of VLDB*, pages 327–336, Sep. 1991.

[24] N. H. Gehani, H. V. Jagadish, and O. Shmueli. COMPOSE: A System For Composite Event Specification and Detection. Technical report, AT&T Bell Laboratories, Dec. 1992.

[25] N. H. Gehani, H. V. Jagadish, and O. Shmueli. Composite Event Specification in Active Databases: Model & Implementation. In *Proc. of VLDB*, pages 327 – 338, 1992.

[26] A. Gilani, S. Sonune, B. Kendai, and S. Chakravarthy. The anatomy of a stream processing system. In *BNCOD*, pages 232–239, 2006.

[27] E. N. Hanson. The Design and Implementation of the Ariel Active Database Rule System. *IEEE TKDE*, 8(1), 1996.

[28] Q. Jiang, R. Adaikkalavan, and S. Chakravarthy. Towards an Integrated Model for Event and Stream Processing. *TR CSE-2004-10, CSE Dept., Univ. of Texas at Arlington*, 2004.

[29] Q. Jiang, R. Adaikkalavan, and S. Chakravarthy. $NFM^i$: An Inter-domain Network Fault Management System. In *Proc. of ICDE*, Apr. 2005.

[30] Q. Jiang and S. Chakravarthy. Data Stream Management System for MavHome. In *Proc. of ACM SAC*, Mar. 2004.

[31] Q. Jiang and S. Chakravarthy. Scheduling Strategies for Processing Continuous Queries over Streams. In *Proc. of BNCOD*, Jul. 2004.

[32] A. Kotz-Dittrich. Adding Active Functionality to an Object-Oriented Database System - a Layered Approach. In *Proc. of the Conference on Database Systems in Office, Technique and Science*, Mar. 1993.

[33] D. L. Lieuwen, N. H. Gehani, and R. Arlein. The Ode Active Database: Trigger Semantics and Implementation. In *Proc. of ICDE*, pages 412–420, Mar. 1996.

[34] S. Madden and M. J. Franklin. Fjording the Stream: An Architecture for Queries over Streaming Sensor Data. In *Proc. of ICDE*, 2002.

[35] S. R. Madden et al. The Design of an Acquisitional Query Processor for Sensor Networks. In *Proc. of SIGMOD*, 2003.

[36] M. F. Mokbel et al. PLACE: A Query Processor for Handling Real-time Spatio-temporal Data Streams. In *Proc. of VLDB*, Sep. 2004.

[37] R. Motwani et al. Query Processing, Resource Management, and Approximation in a Data Stream Management System. In *Proc. of CIDR*, Jan. 2003.

[38] U. Schreier et al. Alert: An Architecture for Transforming a Passive DBMS into an Active DBMS. In *Proc. of VLDB*, 1991.

[39] N. Tatbul et al. Load Shedding in a Data Stream Manager. In *Proc. of VLDB*, Sep. 2003.

[40] J. Widom and S. Ceri. *Active Database Systems: Triggers and Rules*. Morgan Kaufmann Publishers, Inc., 1996.

[41] Y. Yao and J. E. Gehrke. Query Processing in Sensor Networks. In *Proc. of CIDR*, Jan. 2003.

[42] C. Zaniolo et al. Stream Mill, Available [Online]: http://wis.cs.ucla.edu/stream-mill/index.html.