



Department of Computer Science and Engineering
University of Texas at Arlington
Arlington, TX 76019

Towards an Integrated Model for Event and Stream Processing

Qingchun Jiang, Raman Adaikkalavan, and
Sharma Chakravarthy

Technical Report CSE-2004-10
2004

Towards an Integrated Model for Event and Stream Processing

Qingchun Jiang, Raman Adaikkalavan, and Sharma Chakravarthy
Information Technology Laboratory
Department of Computer Science and Engineering
The University of Texas at Arlington
{jiang, adaikkal, sharma}@cse.uta.edu

Abstract

Event processing in the form of ECA rules has been researched extensively from the situation monitoring viewpoint to detect changes in a timely manner and to take appropriate actions. Several event specification languages and processing models have been developed, analyzed, and implemented. More recently, data stream processing has been receiving a lot of attention to deal with applications that generate large amounts of data in real-time at varying input rates and to compute functions over multiple streams that satisfy quality of service (QoS) requirements. A few systems based on the data stream processing model have been proposed to deal with change detection and situation monitoring. However, current data stream processing models lack the notion of composite event specification and computation, and they cannot be readily combined with event detection and rule specification, which are necessary and important for many applications.

In this paper, we analyze the similarities and differences between the event and data stream processing models. Although research seems to address these two as separate topics, there are a number of similarities and differences between the two models. We argue that for many of the applications considered for stream processing, event and rule processing are needed and are not currently supported. On the contrary, event processing systems concentrate on complex event and rule processing in a DBMS environment and do not consider complex stream processing. By synthesizing these two and combining their strengths, we argue that the integrated model will be better than the sum of its parts. We then propose our integrated model and its functionality to combine the capabilities of both models for applications that not only need to monitor changes through continuous queries (CQs), but also to express and process complex events generated by CQs. We introduce the notion of a **semantic window**, which significantly extends the current window concept for CQs, and **stream modifiers** in order to extend current stream computation model for complicated change detection. We further discuss the extension of event specification to include CQs. Finally, we discuss the implemen-

tation of our integrated model using the extended data stream processing system with a local event detector.

1 INTRODUCTION

Event processing [19, 50, 20, 28, 25, 39, 11, 15, 33, 40, 51, 22, 21] and lately data stream processing [7, 1, 17, 41, 36, 44] have evolved independently based on situation monitoring application needs. Several event specification languages [29, 30, 25, 26, 16, 49, 2, 3] for specifying composite events have been proposed and triggers have been successfully incorporated into relational databases. Different computation models [27, 40, 24, 25, 22, 11, 15, 21] for processing events, such as Petri nets [24, 25], extended automata [27, 40, 30], and event graphs [11, 15, 22] – have been proposed and implemented. Various event consumption modes [11, 24, 25, 15, 16] (or parameter contexts) have been explored. Similarly, data stream processing has received a lot of attention lately, and a number of issues – from architecture [1, 41, 36, 17, 47] to Quality-Of-Service (QoS) [53, 6, 18, 37, 10, 12] – have been explored. Although both of these topics seem different on the face of it, we argue that there are more similarities than differences between them. Not surprisingly, the computation model used for data stream processing is not very dissimilar from some of the event processing models (e.g., event graph), but used with a different emphasis.

As many of the stream applications are based on sensor data, they invariably give rise to events on which some actions need to be taken. In other words, many stream applications seem to not only need computations on streams, but also these computations generate interesting events (e.g., car accident detection and notification, network congestion control, network fault management) and several such events may have to be composed, detected and monitored for taking appropriate actions. Currently, to the best of our knowledge, none of the work addresses the specification and computation of the above two threads of work. Our premise for this paper is that although each one is useful in its own right, their combined expressiveness and computation are critical for many applications of stream processing. Hence there is a need for synthesizing the two into a more expressive and more powerful model that combines the strengths of each one.

We use the following running example to explain the current limitations of each model, and the need for the integrated model. Consider the following car accident detection scenario that is slightly different, but more effective, from the linear road benchmark [5].

Example 1 (Car ADN) *In a car accident detection and notification system, each expressway in an urban area is modeled as a linear road, and is further divided into equal-length segments (e.g., 5 miles). Each registered vehicle on an express way is equipped with a sensor and reports its location periodically (say, every 30 seconds). Based on this location stream data, we can detect a car accident in a near-real time manner. If a car reports the same location*

(or with speed zero mph) for four consecutive times, FOLLOWED BY at least one car in the same segment with a decrease in its speed by 30% during its four consecutive reports, then it is considered as a potential accident. Once an accident is detected, the following life saving actions may have to be taken immediately: i) notify the nearest police/ambulance control room about the car accident, ii) notify all the cars in 5 upstream segments about the accident, and iii) notify the toll station so that all cars that are blocked in the upstream for up to 20 minutes by the accident will not be tolled.

Every car in the express way is assumed to report its location every 30 seconds forming the primary input data for the above example. The format of car location data stream (i.e., *CarLocStr*) is given below:

```
CarLocStr(
    timestamp,    /* time stamp of this record */
    car_id,       /* unique car identifier      */
    speed,        /* speed of the car           */
    exp_way,      /* expressway: 0..10         */
    lane,         /* lane: 0, 1, 2, 3         */
    dir,          /* direction: 0(east), 1(west) */
    x-pos);      /* coordinates in express way */
```

CarSegStr is the car segment stream (or the input *CarLocStr* stream), but with the location of the car replaced by the segment corresponding to the location. Query shown below produces the *CarSegStr* from the *CarLocStr* stream.

```
SELECT timestamp, car_id, speed, exp_way, lane, dir,
       (x-pos/5 miles) as seg FROM CarLocStr;
```

Detecting an accident in the above CAR ADN example has three requirements, and they are (1) IMMOBILITY: checking whether a car is at the same location for four consecutive time units i.e., over a 2 minutes window, in our example, as the car reports its location every 30 seconds. (2) SPEED REDUCTION: finding whether there is at least one car that has reduced its speed by 30% or more during four consecutive time units. and (3) SAME SEGMENT: determining whether the car that has reduced its speed (i.e., car identified in (2)) is in the same segment and it follows the car that is immobile (i.e., car identified in (1)). Immobility of a car can be computed using CQs that are supported by the current data stream processing systems as shown below:

```
SELECT car_id, AVG(speed) as avg_speed
FROM CarLocStr [2 minutes sliding window]
GROUP BY car_id
HAVING avg_speed = 0;
```

With the current event and stream processing models, using a declarative

language¹, it is difficult or impossible to efficiently compute the speed reduction. Whether the cars that are found in requirements (1) and (2) are from the same segment can be readily determined in an event processing model using a **sequence** operator [25, 11, 2]. Notifications or life saving actions have to be taken once the cars are identified, which is not supported by current stream processing model. As the cars that are identified in requirement (3) can be separated by more than 4 time units, it requires an efficient, meaningful and less redundant manner for notifications. In other words, number of times the accident is reported should be kept to a minimum. The above can be done efficiently using the current event processing models, but not the current stream processing model. Although JOIN operator can be used to compute it, the number of reports is not minimized.

As shown, all the above requirements strongly call for an integrated model. Furthermore, the second and third requirements pose challenges for synthesizing an integrated model. We will later illustrate how the above can be specified elegantly and be computed efficiently using the integrated model proposed in this paper. Some of the earlier work on sequence processing [51], temporal aggregation [46], and trigger processing in Ariel [33] address some computational and performance aspects that may be relevant to the extensions proposed in this paper. However, they were not in the context of streams and they addressed processing from event logs in the relational context without the notion of a window. In addition, performance work did not address either the real-time aspects or the QoS constraints.

In this paper we first summarize the characteristics of both threads of work to set the stage for understanding the differences and similarities. We propose an integrated model that combines the two using a uniform computation model. We introduce the notion of a semantic window, which significantly extends the current window concept for CQs, and stream modifiers in order to extend current stream computation model for complex change detection. We further discuss the extension of event specification to include CQs. Finally, we discuss our integrated model and its implementation. The extensions proposed in this paper are critical for integrating the two computation models in a seamless manner. The ability to use CQs as events and the extended specification of CQs and events are also needed for the integration of the two.

The rest of the paper is organized as follows. Section 2 explains the data stream processing model. Event processing model is explained in section 3. Detailed comparisons of both the models are presented in section 4. Integrated model is presented in section 5. Prototype system combining stream and event processing is discussed in section 6. Section 7 has related work. Conclusions and future directions are discussed in section 8.

¹Models that are based on procedures may compute this, but they are more difficult to use than those models that are based on declarative languages (i.e., SQL). In this paper we consider the latter one.

2 Data Stream Processing Model

In stream processing model, data items arrive as a continuous stream, which are ordered by their arriving time stamp or by other attributes (e.g., sequence number in an IP header). The data in this model can be accessed only sequentially and the data items are typically accessed only once. The operations in the model are read-only operations. However, computations on stream processing do not preclude the use of data from a conventional DBMS. For example, a join operator can use stream data as one input and a stored relation as another input. Therefore, update operations on conventional stored relations are not precluded.

The main computation required in the stream processing model is to incrementally compute queries against arriving tuples and to meet predefined QoS requirements of each query. This requires: 1) a query to be active in the system as long as its input data streams are not terminated, 2) certain QoS requirements to be met for those long-lived CQs, and 3) relational computations over conventional DBMSs to be carefully adapted (e.g., use of windows) to the stream data processing model due to the unbounded nature of input.

Due to the unique requirements of streams, data stream processing has received widespread attention, and there exists a large body of work, such as Aurora [1], Fjord [41], MavStream [36], NIAGARA [17], PLACE [44], STREAM [47] and so forth, to name a few. A number of solutions have been proposed for various problems on data stream processing. A queue is associated with each operator in order to handle bursty input(s), which permits a CQ to be modeled as a network of queueing system [35]. The paper [35] also shows how to model a CQ by applying queueing theory and use the queueing model to predict various QoS metrics for processing a data stream. To deal with the unbounded input size of an input stream, a sliding window [4, 1] is used to capture a subset of an input stream, on which an operator can compute its results. Due to the long-life of stream-based operators, various scheduling algorithms have been proposed to minimize: memory requirements [10], tuple latency [12, 37], or their combination [37]. A few QoS delivery mechanisms [53, 6, 18] have been proposed to handle overload situations in data stream processing systems. Additionally, various join algorithms [18, 32, 38] and indexing techniques [31] have been proposed to compute CQs efficiently. Complex event processing has received little attention in the current model.

3 Event Processing Model

Event-Condition-Action (ECA) rules are used to process event sequences and to make the underlying system active for applications such as situation monitoring, access control, and change detection. They consist of three components and they are 1) Event: occurrence of interest such as data-manipulation-events, clock-events, and external-notification-events, 2) Condition: can be a simple or a complex query, and 3) Action: specifies the operations that are to be

performed when an event occurs and the corresponding condition evaluates to true. ECA rules can be defined either at application level or system level. A number of event processing systems using ECA rules have been proposed and implemented in the literature ACOOD [22], ADAM [20], Alert [50], Ariel [33, 34], COMPOSE [28], Hipac [19], ODE [27, 40], REACH [11], Rock & Roll [21], SAMOS [24, 25], Sentinel [15, 16], SEQ [51], UBILAB [39], and [45, 46]. A comprehensive introduction and description about most of these systems can be found in [54, 48].

Primitive or simple events are specific to a domain and are predefined. On the other hand, composite or complex events are composed of more than one primitive or composite event forming an event expression using event operators. Event operators [29, 30, 25, 26, 16, 49, 2, 3] are used to compose events and can be unary, binary, or ternary based on the number of operands. Primitive events are detected by the underlying system at the time of occurrence. For example, the time of occurrence of an event can be the time at which a method is invoked by an object. Composite events are detected based on the event operator semantics when all of its constituent events occur. The time of occurrence of a composite event depends on the event operator semantics and detection semantics [23]. Several approaches have been proposed for the detection of composite events in the literature and they are: event detection graphs (EDGs) [22, 11, 16], extended finite state automaton [27, 40], colored Petri nets [26, 24, 25], and event algebra [39]. EDGs have been shown to be based on operators rather than instances and hence are efficient as compared to other approaches based on the computation and storage requirements for detecting events.

In this paper, we draw upon EDGs for our integrated model as it corresponds to operator trees and has similarities with respect to query processing whereas the other representations do not share these characteristics with query processing. We also use the masking capability introduced in ODE [30] to filter events on arbitrary conditions. Unconstrained event operator semantics correspond to the unrestricted (or general) context. This means events, once they occur, cannot be discarded at all. For a sequence event operator [25, 11, 14], all event occurrences that occur after a particular event will get paired with that event as per the unrestricted context semantics. In the absence of any mechanism for restricting event usage (or consumption), events need to be detected and parameters for those composite events need to be computed using the unrestricted context definitions of the event operators. However, the number of events produced (with unrestricted context) can be large and not all event occurrences may be meaningful for an application. In addition, detection of these events has substantial computation and space overhead that may become a problem for situation monitoring applications. Event consumption modes (or contexts) are supported by several active database systems, such as *ACOOD*, *SAMOS*, *Sentinel*, and *REACH* for restricting the unnecessary events from being detected.

4 Analysis of Event Vs. Stream Processing

Processing of events using event detection graphs (analogous to a query tree) and a data flow architecture, is similar to the processing of data streams. In this section, we analyze the relationship between event processing and data stream processing models. This will form the basis of our integrated model that combines the strengths of both.

4.1 INPUTS AND OUTPUTS

Inputs (or data sources) to an event processing model are a stream of events (or event histories). Event streams are considered as a sequence of events that are ordered by their time of occurrence. Most of event streams considered by event processing models are generated by the underlying system such as the data manipulation operations (i.e., INSERT, UPDATE, and DELETE) in RDBMSs, system clock, etc. The input rate of an event stream is not assumed to be very high and highly bursty as they are generated by the underlying system. The outputs of event operators also form event streams, which are ordered by their occurrence timestamps (may be an interval for composite events). Once the events are detected, they are used to trigger a set of ECA rules. Once the ECA rules are triggered, necessary conditions are checked and predefined actions are taken.

Main inputs to a data stream processing model are data streams. Input tuples in a data stream can be ordered by any attribute and not always by the 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 highly unpredictable and dynamic (e.g., bursty). The main output of a data stream processing model, if one of its input is a data stream, is a data stream too. Thus, conceptually, both the models have similar inputs and outputs. However, the data sources in data stream processing model are mostly external sources with high input rates and highly bursty input model, where as the data sources in event processing model are mainly internal ones with relatively low input rates.

4.2 CONSUMPTION MODES Vs. WINDOWS

Event consumption modes or contexts were introduced primarily to determine how many events from the same event sequence should be kept for the purpose of detecting composite events. The number of events to be kept depended solely on the context of the operator and the semantics of the operator. For example, for the sequence operator, one could not drop any event from the past if no context was associated (i.e., use of general context). On the other hand, for the OR operator, that is not the case. Event contexts indirectly resulted in keeping

a small portion of (the head of) an event sequence based solely on the value of timestamp and the context. Also, the set of event instances retained at a binary operator depended upon the dynamics of both event streams. For instance, if the completing/terminating event did not occur, event instances for the other operator would accumulate for some contexts such as chronicle and continuous.

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. The objective of defining a window in stream processing was to convert blocking operators into a non-blocking computation and to produce output in a continuous manner. Hence, the window generated by a context is not the same as the window concept in streams.

4.3 EVENT OPERATORS Vs. CQ OPERATORS

Event operators are quite different from the operators supported by current stream processing model. Event operators are mainly used to express and define the computation on event (tuple) level and to reduce the number of output events through consumption modes or profiles, and they solely use the timestamp of an event for detecting composite events. For example, AND operator in the event processing model is used to express and compute the occurrence (appearance) of two events (tuples) from its two input event streams. Thus, event operators do not perform any computation over the attributes of these events (tuples). On the other hand, current stream processing operators are mostly modified relational operators ², which focus on how to express and define the computation at the attribute level, rather than the tuple level. Additionally, stream processing operators have input queues and internal windows in order to deal with highly bursty inputs and to convert blocking operators to non-blocking operators.

Both relational or event operators do not have input queues and internal windows, as relational operators perform computations over static relations and event operators perform computations over low input-rate event streams. This is also because of the underlying assumption that query or event processing systems have sufficient processing capacity and resources to handle their data sources.

4.4 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) computations performed at the tuple level, which are carried out by the event operators, 2) computations performed at the attribute level, which are carried out by the condition checks, and 3) computations for processing

²The blocking operators have been converted as non-blocking in data stream model for properly computing CQs.

rules (triggering actions). In some event processing systems, a smaller part of the computations that are carried out at the attribute level are moved to the event detection component (i.e., the mask proposed in [29, 30]), improving the performance. 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 for blocking operator computations.

Thus, computations that are performed at the tuple level and the computations for rule processing in the event processing models are absent in the stream processing model. On the other hand, window computations in the stream processing model do not appear in the event processing model. Even though both of them operate at the attribute level, operator computation in the stream processing model is more powerful than the computation performed for condition checking in the event processing model. From the above it is clear that the computations in both the models have different emphasis and different purposes, and they are for different applications. Thus, integration of these two computation models is more powerful and useful and can support larger class of applications.

4.5 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 [8], event processing models do not support any specific QoS requirements. Typically, in the event processing model, whenever an event occurs it 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) for the following reasons: 1) the input of a data stream processing system is highly dynamic and unpredictable in contrast to its fixed computation resources. During overload periods, some queries cannot get sufficient resources to compute their results, which can cause unexpectedly long delays for the final output results. 2) many stream-based applications require near real-time responses from underlying stream processing system. A delayed response may not be useful, and may even cause serious problems. Different applications can tolerate different response times or inaccuracy in the final query results, and 3) queries with different QoS requirements must be treated differently with a goal to minimize the overall violation of predefined QoS specifications. A number of QoS delivery mechanisms have been explored and proposed [53, 6, 18].

4.6 OPTIMIZATION AND SCHEDULING

Event expressions are represented as event graphs for detection. There has been some work carried out in rewriting event expressions, so that event detection can be made efficient by constructing optimal event detection graphs. Common event sub-expressions are grouped in order to reduce the 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 [10, 12, 37] in stream processing.

4.7 BUFFER MANAGER 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. Most of the event processing models assume that the incoming events are not bursty and hence do not provide any kind of 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 dynamically. 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 [53, 6, 18].

4.8 HIGH-LEVEL RULE PROCESSING

ECA Rules describe how the underlying system should respond when an event occurs, making the system reactive. Rules are either used to extend the range of applications that can be supported by the underlying system or to change the way in which new applications are developed. Rules are considered important as they allow users to specify predefined actions that need to be taken when an event occurs and the corresponding conditions are satisfied. Existing event processing systems support dynamic enabling and 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 [54, 13] have been proposed to accurately define and efficiently execute various rules in the literature for event processing models. Those semantics include rule processing granularity, instance/set oriented execution, iterative/recursive execution, conflict resolution, sequential/concurrent execution, coupling modes, and termination. Stream processing systems do not support high-level rule specification and processing, which are critical to many real-world applications.

4.9 Summary

Similarities and differences between event and stream processing models have been discussed above. Both models employ a similar data-flow architecture as their computation model over streaming data. However, both models have their limitations for handling applications that require stream processing followed by event processing, and most of the functionalities provided by these two models are complementary to each other. The stream processing model focuses on providing a set of functionalities similar to those provided by DBMSs to process and manage data streams. As a result, a general framework and a set of comprehensive techniques such as the notion of a window, optimization techniques, scheduling strategies, load shedding, and others, have been proposed and are being developed. On the other hand, the event processing models focus on detecting composite events and rule processing under the assumption that event sequences are generated (mostly) within the underlying systems with relatively low input rate. Consequently, scheduling strategies, load shedding techniques, QoS support, and so on were not explored for that model. However, the three component computation model (i.e., event processing, condition checking, and rule processing) developed in the event processing are specialized for event detection and rule processing. In contrast, event computation at tuple level and rule processing are absent in the data stream processing model. Although CQs consisting of modified relational operators and aggregation operators can be used in situation monitoring, its expressiveness and computation capability of complicated events are limited and it is also not well-suited for detecting composite events and applying contexts to reduce the number of meaningful events compared with the techniques provided by the event processing models.

Clearly, it is desirable and natural to combine the strengths of both models into an integrated model with a general framework and a set of comprehensive techniques of stream processing model plus the event computation model (i.e., computation at tuple level, consumption modes, and so on) and sophisticated rule processing capabilities. This integrated model will be much stronger and can serve a larger class of applications than what are currently supported by both the models individually.

5 EStreams: An Integrated Model

The proposed integrated model, termed EStreams (for *Event* and *Stream* processing system) is shown in Figure 1 and it consists of three stages: 1) CQ processing stage used for computing CQs over data streams, 2) event processing stage that is used for detecting events with/without masks, and 3) rule processing stage that is used to check conditions, and to trigger predefined actions once events are detected.

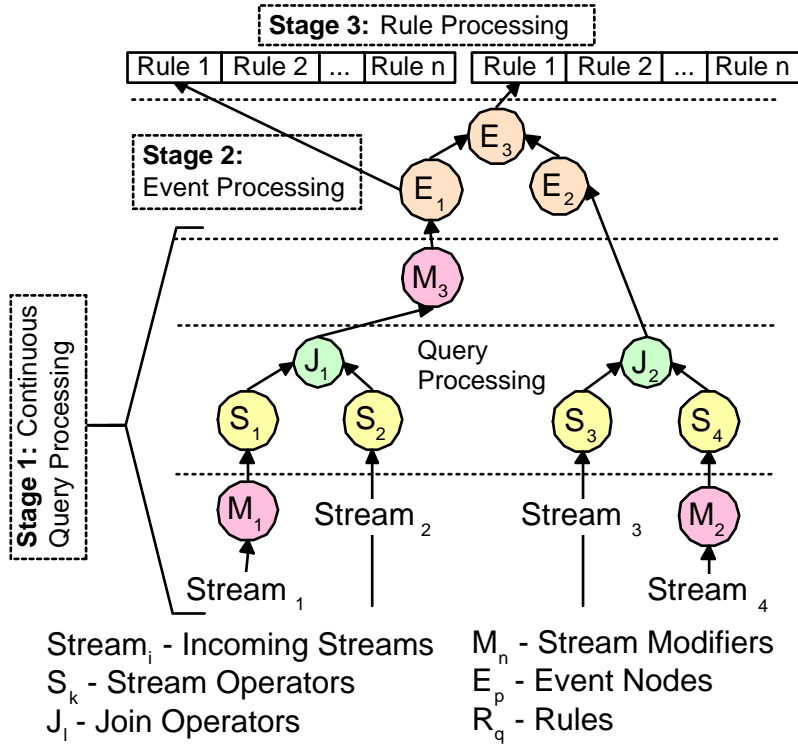


Figure 1: EStreams: Three Stage Integration Model

The seamless nature of our integrated model is due to the compatibility of the chosen event processing model (i.e., an event detection graph) with the structure used for stream processing. Based on our analysis, synthesizing both the processing models requires the following issues to be addressed: 1) handling highly bursty event streams (generated by the CQ processing stage) in event processing, 2) processing of events streams based on attributes and not solely on timestamp, 3) specification of events/event expressions, rules and CQs. We have enhanced both the models, as described below, to address the above mentioned issues: 1) output of CQs have to be fed as inputs to the primitive events in the event processing stage. We have named the continuous queries, so that in the event processing stage the outputs of CQs can be used for detecting events. 2) we have introduced stream modifiers that can detect complex changes between tu-

ples in a stream, 3) we have also introduced the semantic window to enhance the expressiveness and computation efficiency of CQs, and to allow creation of more meaningful windows; For the event processing model, 4) we have enhanced the event operators by introducing input queue(s) for each operator, which makes it possible to handle highly bursty outputs from CQ processing stage and take advantages of the techniques (i.e., scheduling strategies, load shedding) developed for stream processing model. 5) we have enhanced the event expressions in such a way that primitive events can process event streams based on event attributes, and not only on timestamp. 6) we have also enhanced the event consumption modes to support more meaningful windows. Finally, 7) we have extended SQL allowing user to specify events/event expressions, rules and CQs together.

5.1 Continuous Query (CQ) Processing Stage

This stage processes normal CQs where it takes streams as inputs, and outputs 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 events for defining situations that use multiple streams and composite events: 1) A large group of stream applications are interested in, not only the normal computations introduced by CQs with Select-Join-Project and aggregation operators, but also in the changes to one or more attributes of a monitored object over time. and 2) current windows based on size (i.e., in terms of time, number of tuples, values of attribute) are somewhat restrictive (refer to section 5.1.2).

To overcome the above shortcomings, we enhance CQs in the following aspects to support more complicated computations required by many stream applications: 1) Ability to name the CQs is needed so that they can be used to define primitive events, and can be used in defining other CQs, 2) We introduce a family of operators, which can be used to compute the changes of one or more attributes over a window. These operators are termed as **stream modifiers** in this paper, and 3) Finally, we extend the current **window** concepts to a **semantic window** in order to express and compute more meaningful and complicated windows in an accurate and efficient way. These enhancements not only greatly improve the ability of stream processing model to express more complicated computation requirements through semantic windows and named CQs, but also improve the ability to compute more accurate final results in a more efficient way through stream modifiers and semantic windows. However, none of the enhancements affect the operator semantics, scheduling algorithms, QoS delivery mechanisms, and other components proposed for stream data processing.

5.1.1 Named Continuous Queries

Many computations over streaming data are difficult to express and to be computed as a single CQ. In order to express computations more clearly, CQs are

named. The name of a CQ is analogous to the name of a table in a DBMS and it has the same scope and usage as that of 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 following CREATE CQ statement.

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

However, the FROM clause in a named CQ can use any previously defined CQs through their unique names. The meta information of a named CQ is maintained in a CQ_dictionary in the system. The meta information includes the *query name*, its *input sources*, all *output attributes* ordered by their order in final output tuples, and its *output destination(s)*. If the output destination is to an application, it can be in the form of a named pipe, socket, or an output queue/buffer. A default output destination is defined in the system simply as a sink if there is no destination associated with this CQ. A CQ with a sink as its destination can be disabled in the system until a meaningful destination is associated to it. For example, you can define a named CQ and register it with the system, and then register another CQ that refers to this CQ. Once a named CQ is registered to the system, if it refers to any other named CQ, the system will automatically associate it to the destination of its referred CQs. A named CQ can output its final results to multiple destinations.

5.1.2 Semantic Window

In current stream processing models, all blocking relational operators are supported through the concept of a window, which defines a historical snapshot of a finite portion of a stream at any time point. This window 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. Size-based (Time-based, Tuple-based) or Attribute-based windows are the common window types that are defined and supported by current stream processing models. A Time-based window can be simply expressed by *[Range N time units]* and a Tuple-based window can be expressed by *[Row N tuples]*, where N specifies the length of the window. Attribute-based is introduced in [1] and can be expressed by **Size s**, **Advance i** along with a user-defined or pre-defined procedure-based function/operator, where **s** is the size of the window in terms of values of an attribute and **i** is an integer or predicate that specifies how to advance the window when it slides. In addition, a partitioned sliding window [4] is defined as *[PartitionBy A_1, \dots, A_k RowsN]*. It logically partitions input stream into different sub-streams based on the equality of attributes A_1, \dots, A_k and the computations (i.e., aggregation) are performed over each sub-stream. The outputs from all sub-streams are merged into one single output stream.

Although the basic window types³ are useful for many applications, they are still limited by their inability to express more meaningful windows (i.e.,

³From now, all the above mentioned 4 types of windows are referred as basic windows.

based on semantic information) required by applications. The functionality of a stream operator do 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**. Expressing a meaningful and accurate window under different application domains is complicated, and it requires a general-purpose format to express the window concept and a more efficient way to compute the defined window than what is currently available ⁴. 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**, introduced in this paper, can express more complicated and meaningful windows required by different applications than the basic windows and the computations can be carried out through well-developed and highly optimized query processing engines, which is more efficient.

The main function of a semantic window is to determine which tuples in the current window should be deleted ⁵ after it adds a new tuple ⁶ into the current window. Before defining a semantic window, we have to identify the scope of data which it can access to perform computations. Obviously, all tuples in the current window and the new tuple that needs to be added to the current window are fully accessible by that window. Therefore, it is natural to express a semantic window based on the semantic information provided by current window and the new tuple. Each semantic window specifications uses a \mathcal{CW} reference and a \mathcal{NT} reference. Through the \mathcal{CW} , a semantic window can access any tuple in the current window. Similarly, all the attributes of the new tuple that needs to be added to the current window can be accessed through \mathcal{NT} .

Instead of introducing new operations for a semantic window in current SQL, we claim that a semantic window can be expressed using current SQL statements (i.e., SELECT-FROM-WHERE statement) over \mathcal{CW} and \mathcal{NT} with little effort needed for its implementation. We also can take advantage of the well-developed SQL query processing engine and its optimization techniques provided in stream processing model to efficiently compute semantic windows with little effort to modify current data stream systems.

A semantic window defines a finite portion of historical tuples seen so far from a data stream through the condition \mathcal{SWC} (*semantic window condition*), over \mathcal{CW} and \mathcal{NT} . Each \mathcal{NT} is appended to \mathcal{CW} by continuously evaluating the \mathcal{SWC} through Algorithm 1.

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

⁴The computation required to maintain a window is not discussed in detail in the literature, not mentioning the optimization of window computation.

⁵Since the oldest tuple in a window is the least useful for most applications, in this paper, we concentrate on windows that delete the oldest tuple (like basic windows). Deletion of other tuples (i.e., every n^{th} tuple) can also be achieved when procedures are used, rather than SQL, for the semantic window.

⁶Similarly, instead of a single tuple, a batch of new tuples can be merged into the window for performance reasons.

Algorithm 1: Add New Tuple Algorithm

```
INPUT: ( $CW$ ,  $\mathcal{NT}$ );
OUTPUT: Modified  $CW$ ;

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

where \mathcal{NT} is considered as a one tuple relation.

```
Stream [      CHECK  logical_Expression
             SELECT  a_1, a_2, ..., a_n
             FROM     $CW$ ,  $\mathcal{NT}$ 
             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 CW and \mathcal{NT} can appear in FROM clause. a_1, a_2, \dots, a_n are the attribute names (or alias after applying aggregate functions) from either CW or \mathcal{NT} . The CHECK clause is a logical expression which consists of the attribute names used in the SELECT statement, relational and logical operators, and parentheses. The CHECK clause is evaluated and a Boolean value is returned once the SELECT clause is evaluated. 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.

Based on the definition of SWC and named CQs we provide few examples to demonstrate the use of semantic window to support some of the basic windows as well as other more meaningful ones.

Example 2 (Similar with \square Row 50,000 \square) *The average speed of each car over a 50,000-tuple sliding window on stream CarLocStr can be computed using the following CQ with a regular Row window and a semantic window respectively. (Note: the SWC is evaluated after we append the \mathcal{NT} to CW).*

```

CREATE CQ AvgSpeedTuple AS
SELECT car_id, AVG (speed)
FROM CarLocStr [Row 50,000]
GROUP BY car_id

CREATE CQ AvgSpeedTuple AS
SELECT car_id, AVG(speed)
FROM CarLocStr [ CHECK CWCount <= 50,000
SELECT COUNT(*) AS CWCount FROM CW ]
GROUP BY car_id

```

It is worth noting that windows based on the number of tuples, irrespective of the kind of definition (i.e., Range or semantic) used, are not commutative [4]. From the above example we had clearly shown that semantic windows can be used to express and compute the common Tuple-based windows. Similarly we can represent other types of windows (we are not discussing those due to lack of space). However, a semantic window can be used to express and compute more meaningful windows, as discussed below.

Example 3 (Disjoint Window) *Compute the moving average speed of the car with car_id 100 within each segment.*

```

CREATE CQ AvgSpeedOfCar100_Wrong AS
SELECT AVG(speed)
FROM CarSegStr [ CHECK CWSeg = NTSeg
SELECT CW.seg AS CWSeg, NT.seg AS NTSeg
FROM CW, NT
WHERE (CW.timestamp =
(SELECT MIN(CW.timestamp) FROM CW)) ]
WHERE CW.car_id = 100

```

The above CQ AvgSpeedOfCar100_Wrong does not compute the average speed of the car with car_id 100 since the window is computed before the outer WHERE clause is evaluated. The window will change whenever a report from a car from different segment is received. Instead of specifying all the computations in one CQ, we can take advantage of the named CQs proposed in this paper and use the following two CQs to accurately and efficiently compute the Example 3. Similarly, we can also compute Example 3 in one CQ by using the semantic window with GROUP BY, which will be discussed later. However, it is not efficient since a sub-window should be maintained for each car.

```

CREATE CQ FilterForCar100 AS
SELECT *
FROM CarSegStr
WHERE CW.car_id = 100

```

```

CREATE  CQ AvgSpeedOfCar100 AS
SELECT  AVG(speed)
FROM    FilterForCar100 [ CHECK CWSeg = NTSeg
                        SELECT CW.seg AS CWSeg, NT.seg AS NTSeg
                        FROM CW,NT
                        WHERE (CW.timestamp =
                               (SELECT MIN(CW.timestamp) FROM CW)) ]

```

Example 4 Consider a network traffic stream $NTStr$ consisting of $timestamp$, pkg_id , pkg_size , $source_IP$, $dest_IP$, $source_port$, and $dest_port$. We want to compute the number of flows determined by the unique combination of ($source_IP$, $dest_IP$) over a window in which the total number of bytes transferred does not exceed 100 MB and the number of time units spanning the window does not exceed 10 seconds. (Note: Below “+” indicates concatenation)

```

CREATE  CQ NumberOfFlows AS
SELECT  COUNT(distinct source_IP + dest_IP)
FROM    NTStr [ CHECK ((CW_S <= 100M) AND
                      (NT_TS - CW_TS < 10Sec))
              SELECT SUM (CW.pkg_size) AS CW_S,
                      MIN(CW.timestamp) AS CW_TS,
                      MIN(NT.timestamp) AS NT_TS
              FROM CW, NT
              GROUP BY NT.timestamp ]

```

As NT is a one tuple relation, the GROUP BY clause in SWC in the above example outputs only one tuple. Otherwise, GROUP BY will partition the window into multiple sub-windows, which is discussed below. Before we discuss the GROUP BY clause used in SWC , we will give an example to show that GROUP BY is necessary and cannot be replaced by moving it to the CQ.

Example 5 Compute the moving average speed of each car on its current segment. Two CQs are provided, the first one gives what is required, and the second one does different computation.

```

CREATE  CQ CarMovingAvgSpeed AS
SELECT  AVG (speed), car_id
FROM    CarSegStr [ CHECK CWSeg = NTSeg
                   SELECT CW.seg AS CWSeg, NT.seg AS NTSeg
                   FROM CW, NT
                   WHERE (CW.timestamp =
                          (SELECT MIN(CW.timestamp) FROM CW))
                   GROUP BY CW.car_id ]

```

In the above CQ, we partition the input stream into multiple sub-streams based on car_id , apply the window computation (i.e., a window formed by the tuples from the same segment for a particular car) to form a correct window,

and then compute the moving average speed of the car based on the tuples in its window. Finally, the outputs from all sub-streams are merged into an output stream. As shown below, if the GROUP BY is moved down to the CQ, the query just forms one semantic window from the input stream, applies the GROUP BY and finally output the results.

```

CREATE  CQ CarMovingAvgSpeed AS
SELECT  AVG (speed), car_id
FROM    CarSegStr [ CHECK CWSeg = NTSeg
                   SELECT CW.seg AS CWSeg, NT.seg AS NTSeg
                   FROM CW, NT
                   WHERE (CW.timestamp =
                          (SELECT MIN(CW.timestamp) FROM CW)) ]
GROUP BY car_id

```

Clearly, the window formed by the latter case (i.e., GROUP BY statement is part of the CQ) is based on the input tuple without considering the car_id. If two consecutive tuples are from two different cars that are on the different segments, the query always outputs the current speed of each car as the window is a one tuple window. But, if the two consecutive tuples are from two different cars that are on the same segment, then the query computes the average speed of these two cars, rather than the average speed of a particular car. Therefore, the GROUP BY in *SWC* is necessary.

The semantic window expressed using a GROUP BY is similar to the partitioned window type introduced in [4]. 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 *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 *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 the 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 (after adding the new tuple). If the computation is applied to the other sub-windows, duplicated results will be generated.

Example 6 *Considering the network traffic stream NTStr in Example 4, we want to monitor the number of sessions of each host connected to the SIGMOD web server, which has its IP addresses as 199.222.69.250 and 199.222.69.251, over a sliding window of last 10,000 packets of each host or 5 minutes (300 seconds).*

```

CREATE CQ SIGMODTraffic AS
SELECT *
  FROM NTStr
 WHERE dest_IP = 199.222.69.250 OR dest_IP = 199.222.69.251

CREATE CQ NumberOfSessions AS
SELECT source_IP, count(distinct source_port) as sessions
  FROM SIGMODTraffic [ CHECK ((NPack <= 10,000) OR
                             (( NT_TS - CW_TS) <= 300Sec))
                       SELECT count(*) AS NPack,
                             min(CW.timestamp) AS CW_TS,
                             min(NT.timestamp) AS NT_TS
                       FROM CW, NT
                       GROUP BY CW.source_IP ]

```

The semantic window concept introduced above greatly enhances the expressiveness power of CQs as it allows accurate and meaningful ways of expressing a window for stream-based applications, and it also provides an efficient way of computing semantic windows through highly optimized SQL engines. Further, the adaptation of SQL for expressing it makes it even more useful as it avoids introducing new language constructs and defining its semantics and can be easily integrated into current stream processing model.

The computation introduced by maintaining a semantic window is necessary for window-based operators and its computation overhead is less than those introduced by basic window types. This is due to the fact that 1) accurate definition of a window reduces the computation overhead required to compute the operator as the number of tuples is decreased in its window, 2) well-developed optimization techniques and the well-optimized SQL engine can be used to process SQL-based semantic window efficiently. More important, 3) more meaningful and accurate window definition and computation can be supported only by our semantic window and not by current window definitions and computations. Many critical stream-based applications need accurate results that can only be obtained through accurate definition of windows, which cannot be achieved through basic window types.

Therefore, the semantic window introduced in this paper is necessary, critical, and efficient for data stream processing model and our integration model. The detailed computation overhead and optimizations of semantic windows are an interesting problem and will be part of our future work.

5.1.3 Stream Modifiers

The `stream modifiers` are introduced for CQs in order to extend the computation of current stream processing to capture the changes of interest in an input data stream. Before we introduce the detailed semantics of a stream modifier, we define the state in an input stream as follows:

Definition 1 (State) A state in a data stream is defined as a tuple in that stream. An n -state in a stream is denoted by $s = \langle v_1, v_2, \dots, v_n \rangle$, where v_i is the value corresponding to attribute A_i defined in the stream schema.

Definition 2 (Stream Modifier) A stream modifier is defined as a function to compute the changes (i.e., relative change of an attribute) between two consecutive states of its input stream. A stream modifier is denoted by $M(\langle s_1, s_2, \dots, s_i \rangle [, P < pseudo \rangle], [O|N \langle v_1, v_2, \dots, v_j \rangle])$, where M is called the modifier function that computes a particular kind of change. The i -tuple $\langle s_1, s_2, \dots, s_i \rangle$ is the parameter required by the modifier function M . The following $P < pseudo \rangle$ 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 could be defined using the above definitions. Currently, we have implemented the following three commonly used stream modifiers in our system. In the following definitions, x^i and x^{i+1} are the values of attribute x from state i and $i + 1$ respectively, and anything in between “[]” is optional.

ADiff() is used to detect absolute changes over two consecutive states. It returns absolute change of the values of attribute s_1 , and the values of a subset of attributes given in $O|N \langle \rangle$ profile. It is formally defined for case O as follows:

$$\begin{aligned} ADiff(\langle s_1 \rangle [, O \langle v_1, v_2, \dots, v_j \rangle]) \\ = \langle \frac{s_1^{i+1} - s_1^i}{s_1^i} [, v_1^i, v_2^i, \dots, v_j^i] \rangle \end{aligned}$$

RDiff() is used to detect the relative changes over two consecutive states. It returns relative change of the values of attribute s_1, s_2 , and the values of a subset of attributes given in $O|N \langle \rangle$ profile. It is formally defined for case N as follows:

$$\begin{aligned} RDiff(\langle s_1 \rangle [, P < pseudo \rangle [, N \langle v_1, v_2, \dots, v_j \rangle]]) \\ = \langle \frac{s_1^{i+1} - s_1^i + pseudo}{s_1^i + pseudo} [, v_1^{i+1}, v_2^{i+1}, \dots, v_j^{i+1}] \rangle \end{aligned}$$

ASlope() is used to compute the slope ratio of two attributes over two consecutive states. It returns the slope ratio of the values of attributes s_1, s_2 , and the values of a subset of attributes given in $O|N \langle \rangle$ profile. It is formally defined for case O as follows:

$$ASlope(< s_1, s_2 > [, P < pseudo >][, O < v_1, \dots, v_j >]) \\ = < \frac{s_1^{i+1} - s_1^i + pseudo}{s_2^{i+1} - s_2^i + pseudo} [, v_1^i, v_2^i, \dots, v_j^i] >$$

A stream modifier can only be used in a SELECT clause and is shown in the example in section 5.1.4. Similarly, any aggregation operator can be used inside a stream modifier. The output of an aggregation operator is considered a normal attribute in a stream modifier. For example we can use $ADiff(< AVG(speed) >, N < car_id, location, timestamp >)$ in an SQL statement.

Since we already have a **window** concept in current CQs, we can further extend the computation of a stream modifier within a window. When a window is specified, a stream modifier can be used to compute the changes between the oldest tuple and the latest tuple, instead of the two consecutive tuples.

5.1.4 CQ for CAR ADN Example

The following **IMMOBILE** and **DECREASE** queries can be used to find all cars that stay at the same location and the cars whose speed has decreased by 30% within the last 2 minutes using the extensions described so far.

```
CREATE CQ DECREASE AS
SELECT RDiff(<speed> as C_speed, p<0.01>,
            N<car_id, location, timestamp>)
FROM CarLocStr [ CHECK CWtime - NTtime <= 2
                  SELECT MIN(CW.timestamp) AS CWtime,
                           MIN(NT.timestamp) AS NTtime
                  FROM CW, NT
                  GROUP BY CW.car_id ]
WHERE C_speed <= -30%

CREATE CQ IMMOBILE AS
SELECT RDiff(<speed> as C_speed, p <0.01>,
            N<car_id, location, timestamp>)
FROM CarLocStr [ CHECK CWtime - NTtime <= 2
                  SELECT MIN(CW.timestamp) AS CWtime,
                           MIN(NT.timestamp) AS NTtime
                  FROM CW, NT
                  GROUP BY CW.car_id ]
WHERE C_speed = 0.0
```

5.2 Event Processing

In this section we will discuss the enhanced event expression computations, event specification using the extended SQL, and event node inputs and outputs.

Enhanced Event Operators and Event Expressions: 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 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 of either class level 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. Example discussed below reveals the limitations of the current event operators that operate based on timestamp.

Let us take the CAR ADN example which has three requirements. Let us assume that event *Eimmobile* represents IMMOBILITY (requirement 1) and event *Edecrease* represents SPEED REDUCTION (requirement 2). An accident is represented as event *Eaccident* and is detected when an event *Eimmobile* happens before (i.e., followed by) event *Edecrease*. In addition, both the cars that are detected should be in the same segment for the event *Eaccident* to happen. Let us assume that stream *CarSegStr* (refer to section 1) sends inputs to the named continuous queries CQ1 and CQ2. CQ1 checks the car for immobility and CQ2 checks for speed reduction. CQ1 inputs to the event node *Eimmobile* and CQ2 inputs to event *Edecrease* with the following formats

CQ1: (timestamp, car_id, speed, exp_way, lane, dir, segment_id)

CQ2: (timestamp, car_id, speed, exp_way, lane, dir, segment_id, decrease_in_speed)

Let us assume that *Eimmobile* occurs at 10.00 a.m. with tuple (10.00 a.m, 1, 0 mph, EW123, 3, NW, 104), and event *Edecrease* occurs at 10.03 a.m. and 10.04 a.m. with tuples (10.03 a.m, 2, 40mph, EW123, 1, NW, 109, 45%), (10.04 a.m, 5, 20mph, EW123, 4, NW, 104, 40%).

Two events are said to be in sequence if the first event precedes the second event either in time (or a monotonically increasing number). Thus, tuples with *car_id* 1 (i.e., *Eimmobile*) and *car_id* 2 (i.e., *Edecrease*) triggers the event *Eaccident*. Similarly tuples with *car_id* 1 and *car_id* 3 triggers the event *Eaccident*. From the above it is evident that in current event processing systems (with or without event masks), the important condition that both the cars should be from the same segment is checked only after the event *Eaccident* is detected. This introduces a high overhead on the event computation as there can be many unnecessary detection of event *Eaccident*. The above example can be modeled using instance level events, but all the instances of a class should be predefined (or known previously). This may be impossible in a system where the data streams' attribute values are dynamic. In addition, they require lot of event nodes, and introduce a high overhead for computation. Hence, event detection computation has to be enhanced to support efficient detection of events. In our integrated model this is achieved by enhancing event expression computation by pushing the event masks into the event operator node, so that attribute conditions are checked before the events are detected.

Inputs to Event Processing Stage: CQs output data streams in the form of tuples. These tuples are fed as input event streams to the event processing stage. We assume that each tuple in a data stream that enters the system is time stamped or has an ordering attribute (i.e., has a monotonically increasing sequence attribute) and can be used in the event processing stage. With enhanced event expression computations any attribute of an event (tuple) can be used in the event processing stage for 1) checking conditions, 2) masking the inputs to the event nodes, and 3) merging event streams.

Event Specification using Extended SQL: CQ processing is compatible with the event graph processing approach used by the event processing model. Thus, by suitably extending queries over event streams and processing them using a push model, users' can monitor diverse event stream combinations in a timely and meaningful manner. Users can specify events based on CQs using the CREATE EVENT statement shown below:

```
CREATE EVENT   $\mathcal{E}_{name}$ 
SELECT   $\mathcal{A}_1, \mathcal{A}_2, \dots, \mathcal{A}_n$ 
MASK     $\mathcal{AC}_1, \mathcal{AC}_2, \dots, \mathcal{AC}_n$ 
FROM     $\mathcal{E}_S \mid \mathcal{E}_X$ 
```

CREATE EVENT creates a named event \mathcal{E}_{name} , SELECT selects the attributes $\mathcal{A}_1, \mathcal{A}_2, \dots, \mathcal{A}_n$ from either the event stream \mathcal{E}_S or the event expression \mathcal{E}_X , MASK applies conditions on the attributes $\mathcal{AC}_1, \mathcal{AC}_2, \dots, \mathcal{AC}_n$ of those events that enter the event operator node in the event detection graph. \mathcal{E}_S is a named CQ or a CREATE CQ statement and \mathcal{E}_X is an event expression that combines more than one event using event operators, and event attributes. Below shown is the CREATE EVENT statement for a primitive event E_{prim} that selects all the cars that have segment id “<15” from the event stream $E1$ produced by $CarLocStr$. As shown, MASK applies attribute condition \mathcal{AC}_1 (i.e., $seg_id < 15$), and SELECT selects the car_id and seg_id FROM $E1$.

```
CREATE EVENT  prim
SELECT  stream.car_id, E1.seg_id
MASK    stream.seg_id < 15
FROM    (CREATE CQ E1 as ...);
```

Creation of Events for CAR ADN Example: Section 5.1.4 provides the CQs for the CAR ADN example. $E_{immobile}$ is the event created from the CQ *IMMOBILE* and $E_{decrease}$ is the event created from the CQ *DECREASE*. Event $E_{accident}$ represents an accident and is detected when an event $E_{immobile}$ happens before $E_{decrease}$, and both the seg_ids are same. Event expression \mathcal{E}_X for the accident is $E_{accident} = E_{immobile}$ SEQUENCE $E_{decrease}$ where SEQUENCE is an event operator. CREATE EVENT for the same is shown below:

```

CREATE EVENT  Eaccident
  SELECT     Eimmobile.car_id as EIcar_id, Edecrease.car_id as EDcar_id,
            Eimmobile.seg_id, Eimmobile.timestamp
  MASK      Eimmobile.seg_id = Edecrease.seg_id
  FROM      (CREATE EVENT Eimmobile
            SELECT IMMOBILE.car_id, IMMOBILE.seg_id
            FROM IMMOBILE)
  SEQUENCE
  (CREATE EVENT Edecrease
  SELECT DECREASE car_id, DECREASE.seg_id
  FROM DECREASE)

```

Event nodes are created in the EDGs based on the CREATE EVENT specifications. Output from the CQ is fed as inputs to the event nodes in the EDGs as event streams. In the integrated model, input to the event nodes can be created in many ways (i.e., for the FROM clause) and they are: (1) from the CQ, (2) from the underlying system, and (3) from an external source. Once *Eaccident* is detected, it is propagated to the rule processing stage.

5.3 Rule Processing

The rule system is responsible for triggering predefined actions. A rule is used to trigger predefined actions once its associated event is detected. In our integrated model rules can be specified and created using the CREATE RULE statement as shown below.

```

CREATE RULE   $\mathcal{R}_{name}$  [,  $\mathcal{CM}$ ,  $\mathcal{CT}$ ,  $\mathcal{P}$ ]
  ON         $\mathcal{E}_{name}$ 
  R_CONDITION  Begin; (Simple or Complex Condition); End;
  R_ACTION    Begin; (Simple or Complex Action); End;

```

As shown above, CREATE RULE creates the rule \mathcal{R}_{name} along with its properties such as coupling mode \mathcal{CM} (e.g., immediate, deferred), consumption mode or context \mathcal{CT} (e.g., recent, continuous) and priority \mathcal{P} (e.g., 1, 2 where 1 is the highest) a positive integer used to set rule priority. ON specifies the event \mathcal{E}_{name} associated with the rule and it can be replaced by the CREATE EVENT statement. In addition a rule also contains conditions associated with the rule and actions to be performed when conditions results are true. Conditions on attributes act as event mask. Other conditions that are pertinent to the rule, and those that are complex (i.e., any arbitrary condition such as average, standard deviations, PL/SQL code etc.,) are specified in the rule condition.

Enhanced Event Consumption Modes: Analogous to the event operator semantics, current event consumption modes are also based on time and have similar drawbacks. In order to make the consumption modes more meaningful and consistent with stream processing, we introduce semantic windows and attribute based event consumption. For example, the recent mode from [14]

can be viewed as a single tuple window and is used by applications where the events are happening at a fast rate and multiple occurrences of the same type of event only refine the previous data value.

We will explain the limitation of the current consumption modes based on the recent consumption mode. Let us assume that event *Eimmobile* occurs at 10.00 a.m. with tuple (`car_id 1, Time 10.00 a.m, seg_id 1, Speed 0`). It is propagated to the SEQUENCE node and it waits in the composite event node for the event *Edecrease* to occur, in order to detect *Eaccident*. When the next instance of event *Eimmobile* occurs at 10.01 a.m. with the tuple (`car_id 4, Time 10.01 a.m, seg_id 3, Speed 0`), it replaces the previous instance with “`car_id 1`” even though both have different `car_ids`. This is because recent mode is based on a single tuple window, and the computation is based only on the timestamp where the instance with a latest timestamp replaces the previous instance. In order to detect the event occurrences without losing potential events, events can be replaced based on some attribute forming a set or partition. Thus, in the above example only event instance of “`car_id 1`” that occurs at a later point in time can replace “`car_id 1`” that occurred at 10.00 a.m., and not “`car_id 4`” that had occurred at 10.01 a.m.

Currently, there is no notion of windows and all the events are kept in the event node until a new instance occurs or until it is consumed. As the input rate is likely to be bursty in the integrated EStreams model (as it is fed by streams), there is a need for associating a window with each event node. Thus, by introducing windows along with the event consumption modes, the event processing system will be able to handle bursty event streams.

Event consumption modes are specified as an option along with the CREATE RULE statement. In our integrated model event consumption modes are enhanced to support attribute and window based computations. As mentioned previously, events can be unary, binary or ternary. Thus, specifying consumption modes in event expressions requires the attributes/windows for all the operators. Enhanced event consumption mode are specified as shown below, where *CT* represents the context, L, M and R represents the left, middle, and right events. Attributes (A), number of events (E), and time units (T) are optional. When an operator is binary only L and R are specified. Similarly A is specified when attribute based partition is required and E/T is specified when the events (tuples) have to be maintained in an event node at a point in time.

CT [L[A, E/T], M[A, E/T], R[A, E/T]]

Let us create a rule *Rsample* for an event *Esample*. This event selects the cars with id “>1000” from the named CQ *Estream*. In addition we want to partition the window based on the attribute `car_id`. This rule should have immediate as *CM*, recent as *CT* (including the attribute as a left event as *Esample* is a primitive event) and with highest priority \mathcal{P} (i.e., 1). There are no conditions/actions associated. CREATE RULE statement for this rule is shown below:

```

CREATE RULE  Rsample, IMMEDIATE, RECENT L[Esample.car_id], 1
ON          (CREATE EVENT Esample
             SELECT Estream.car_id, Estream.timestamp
             MASK Estream.car_id > 1000
             FROM (CREATE CQ as Estream ...))

```

Rule Creation for CAR ADN Example: When an event corresponding to an accident *Eaccident* is detected, various types of life saving actions are required to be performed. Creation of event *Eaccident* is shown in section 5.2. Rule corresponding to the CAR ADN example is shown below.

```

CREATE RULE  AccidentNotify, IMMEDIATE,
             RECENT L[Eimmobile.car_id], R[Edecrease.car_id], 1
ON          EVENT Eaccident
R_CONDITION Begin; (true); End;
R_ACTION   Begin;
           //INDICATE POLICE CONTROL ROOM
           PCR(Eaccident.seg_id, Eaccident.EIcar_id,
              Eaccident.EDcar_id, Eaccident.timestamp);
           //INDICATE AMBULANCE CONTROL ROOM
           ACR(Eaccident.seg_id, Eaccident.timestamp
              Eaccident.EIcar_id, Eaccident.EDcar_id);
           //INDICATE UPSTREAM CARS
           UpSSeg(Eaccident.seg_id, Eaccident.timestamp);
           //INDICATE TOLL STATION
           TollSt(Eaccident.seg_id, Eaccident.timestamp);
           End;

```

6 Prototype Implementation

Currently, we have a prototype implementation of the proposed EStream system which uses the event detection graph and most of the event operators in the literature and the stream processing system. It consists of a CQ engine, a resource manager, a stream manager, a query/operator scheduler, a load shedding manager, a QoS manager, an event detector, and a rule manager.

The operators instantiated in the current system include *project*, *select*, *join*, *group-by*, stream modifiers such as *ADiff*, *RDiff*, *ASlope*, and some aggregate operators, such as *max*, *min*, *average*, *count*, and special operators for streaming data processing, such as *duplicate*, *split*, and *drop*. The CQ engine compiles a named CQ consisting of the above operators into a query plan and registers it with the system. However, the query plans in the system now are static query plans, which are not optimized and cannot adapt to changes in input characteristics and system load. The window type of a stream operator is implemented as a semantic window, which can be specified in a CQ using the

semantic expression proposed in this paper. The ECA manager and rule processing component are the components that are currently being integrated into the system. Their basic role includes management of various ECA rules and provide an effective means to continuously detect events, evaluate conditions, and execute per-defined actions.

7 Related Work

Our paper is directly related to a set of papers [7, 1, 17, 41, 36, 44], 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. To the best of our knowledge, this paper is the first to support event and rule processing for stream processing model and to enhance the data stream computation model horizontally (i.e., semantic window) and vertically (i.e., stream modifier, event and rule processing). The paper [4] 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 [4, 56, 1] that try to enhance the expressiveness power of SQL in a data stream environment. Carlo Zaniolo and et al [56] 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 [4] 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, which reduces the number of tuples in the window, thereby reducing the computation requirements of window-based operators. In [1], the window concept is based on an attribute and is specified in the form of (size s , Advance i), where s is the size (in terms of values of an attribute) of the window and i is an integer or predicate that specifies how to advance the window when it slides. 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. The semantic window based on SQL can be static or dynamic. More important, the implementation of semantic window is straightforward by taking advantage of the existing SQL processor and its run-time overhead can be low because of the

well-developed SQL optimization techniques.

Our paper is further related to a set of papers [18, 32, 38, 52], which try to enhance the computation over streaming data. However, those papers mainly focus on relational operators, such as multi-way join algorithms, and approximated join algorithms. Those enhancements are not capable of efficiently computing the changes over streaming data, which can be done efficiently through a family of stream modifiers proposed in this paper. These stream modifiers allows a stream processing system to flexibly express and efficiently monitor complicated change patterns for a large group of stream applications. A number of sensor database projects, Cougar [9, 55], TinyDB [43, 42] 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 [19, 15, 14, 16, 40, 28, 50, 25, 20, 22, 39, 11]. However event expressions, operators and consumption modes are solely based on timestamps. As discussed, in this paper we have enhanced the event expressions and event consumption modes to support attribute based computations as they are more meaningful in stream processing. In addition, we have added input queues to event operators to support the highly bursty nature of input event streams created from CQs as traditional event processing assumes low input rate.

8 Conclusions and Future Work

In this paper, our focus is to provide an integrated model for advanced stream applications that require not only stream processing, but also complicated event and rule processing. We have analyzed the similarities and differences between the stream processing model and the event processing model developed separately. Based on the sophisticated requirements of advanced stream applications and our analysis, we have proposed EStream, an integrated model that combine their strengths through the techniques and enhancements proposed in this paper. Those techniques and enhancements include: 1) the ability to name CQs, 2) stream modifiers, 3) semantic windows, 4) extended event operators with input queues/buffers, 5) enhanced event expressions, 6) enhanced event consumption modes (or contexts), and 7) extended SQL to support combined specification of events and CQs . Through the above techniques and enhancements, our integrated model not only supports a larger class of applications, but also provides more accurate and efficient ways for processing CQs and event expressions. All the enhancements proposed in this paper do not affect any current stream processing techniques and can be easily integrated into any current data stream management systems. Finally, a prototype of the proposed integrated

model is underway.

In terms of future work, we are investigating optimization techniques and load shedding techniques based on semantic windows. In addition, we are also investigating to support generalized stream modifiers and QoS driven event and rule processing.

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. In *Proceedings, East-European Conference on Advances in Databases and Information Systems*, Sep. 2003.
- [3] R. Adaikkalavan and S. Chakravarthy. Formalization and Detection of Events Over a Sliding Window in Active Databases Using Interval-Based Semantics. In *Proceedings, East-European Conference on Advances in Databases and Information Systems*, Sep. 2004.
- [4] A. Arasu, S. Babu, and J. Widom. The CQL Continuous Query Language: Semantic Foundations and Query Execution. *Stanford Technical Report*, Oct. 2003.
- [5] A. Arasu et al. Linear Road: A Stream Data Management Benchmark. In *Proceedings, International Conference on Very Large Data Bases*, Sep. 2004.
- [6] B. Babcock, M. Datar, and R. Motwani. Load Shedding for Aggregation Queries over Data Streams. In *Proceedings, International Conference on Data Engineering*, Mar. 2004.
- [7] S. Babu and J. Widom. Continuous Queries over Data Streams. In *ACM SIGMOD RECORD*, Sep. 2001.
- [8] 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.
- [9] P. Bonnet, J. E. Gerhke, and P. Seshadri. Towards Sensor Database Systems. In *Proceedings, International Conference on Mobile Data Management*, Jan. 2001.
- [10] B. Brian et al. Chain: Operator Scheduling for Memory Minimization in Stream Systems. In *Proceedings, International Conference on Management of Data (SIGMOD)*, 2003.

- [11] A. P. Buchmann et al. *Rules in an Open System: The REACH Rule System*. Rules in Database Systems, 1993.
- [12] D. Carney et al. Operator Scheduling in a Data Stream Manager. In *Proceedings, International Conference on Very Large Data Bases*, Sep. 2003.
- [13] 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.
- [14] S. Chakravarthy et al. Composite Events for Active Databases: Semantics, Contexts, and Detection. In *Proceedings, International Conference on Very Large Data Bases*, pages 606–617, 1994.
- [15] S. Chakravarthy et al. Design of Sentinel: An Object-Oriented DBMS with Event-Based Rules. *Information and Software Technology*, 36(9):559–568, 1994.
- [16] S. Chakravarthy and D. Mishra. Snoop: An Expressive Event Specification Language for Active Databases. *Data and Knowledge Engineering*, 14(10):1–26, Oct. 1994.
- [17] J. Chen et al. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *Proceedings, International Conference on Management of Data (SIGMOD)*, 2000.
- [18] A. Das, J. Gehrke, and M. Riedewald. Approximate Join Processing over Data Streams. In *Proceedings, International Conference on Management of Data (SIGMOD)*, 2003.
- [19] U. Dayal et al. The HiPAC Project: Combining Active Databases and Timing Constraints. *SIGMOD Record*, 17(1):51–70, Mar. 1988.
- [20] O. Diaz, N. Paton, and P. Gray. Rule Management in Object-Oriented Databases: A Unified Approach. In *Proceedings, International Conference on Very Large Data Bases*, Sep. 1991.
- [21] A. Dinn, M. H. Williams, and N. W. Paton. ROCK & ROLL: A Deductive Object-Oriented Database with Active and Spatial Extensions. In *Proceedings, International Conference on Data Engineering*, 1997.
- [22] H. Engstrom, M. Berndtsson, and B. Lings. Acood essentials. Technical report, University of Skovde, 1997.
- [23] A. Galton and J. Augusto. Two Approaches to Event Definition. In *Proceedings, International Conference on Database and Expert Systems Applications*, 2002.

- [24] S. Gatzui and K. R. Dittrich. SAMOS: An Active, Object-Oriented Database System. *IEEE Quarterly Bulletin on Data Engineering*, 15(1-4):23–26, Dec. 1992.
- [25] S. Gatzui and K. R. Dittrich. Events in an Object-Oriented Database System. In *Proceedings of Rules in Database Systems*, Sep. 1993.
- [26] S. Gatzui 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.
- [27] N. H. Gehani and H. V. Jagadish. Ode as an Active Database: Constraints and Triggers. In *Proceedings, International Conference on Very Large Data Bases*, pages 327–336, Sep. 1991.
- [28] 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.
- [29] N. H. Gehani, H. V. Jagadish, and O. Shmueli. Composite Event Specification in Active Databases: Model & Implementation. In *Proceedings, International Conference on Very Large Data Bases*, pages 327 – 338, 1992.
- [30] N. H. Gehani, H. V. Jagadish, and O. Shmueli. Event Specification in an Object-Oriented Database. In *Proceedings, International Conference on Management of Data (SIGMOD)*, pages 81–90, San Diego, CA, June 1992.
- [31] L. Golab, S. Garg, and M. T. Özsu. On Indexing Sliding Windows over Online Data Streams. In *Proceedings, International Conference on Extended Data Base Technology*, 2004.
- [32] L. Golab and M. T. Özsu. Processing Sliding Window Multi-Joins in Continuous Queries over Data Streams. In *Proceedings, International Conference on Very Large Data Bases*, Sep. 2003.
- [33] E. N. Hanson. The Design and Implementation of the Ariel Active Database Rule System. *IEEE Transactions on Knowledge and Data Engineering*, 8(1), 1996.
- [34] E. N. Hanson. Active Rules in Database Systems. pages 221–232. Springer, New York, 1999.
- [35] Q. Jiang and S. Chakravarthy. Queueing Analysis of Relational Operators for Continuous Data Streams. In *Proceedings, International Conference on Information and Knowledge Management*, Nov. 2003.
- [36] Q. Jiang and S. Chakravarthy. Data Stream Management System for MavHome. In *Proceedings, Annual ACM Symposium on Applied Computing*, Mar. 2004.

- [37] Q. Jiang and S. Chakravarthy. Scheduling Strategies for Processing Continuous Queries over Streams. In *Proceedings, British National Conference on Databases*, Jul. 2004.
- [38] J. Kang, J. F. Naughton, and S. Viglas. Evaluating Window Joins over Unbounded Streams. In *Proceedings, International Conference on Data Engineering*, Mar. 2003.
- [39] 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.
- [40] D. L. Lieuwen, N. H. Gehani, and R. Arlein. The Ode Active Database: Trigger Semantics and Implementation. In *Proceedings, International Conference on Data Engineering*, pages 412–420, Mar. 1996.
- [41] S. Madden and M. J. Franklin. Fjording the Stream: An Architecture for Queries over Streaming Sensor Data. In *Proceedings, International Conference on Data Engineering*, 2002.
- [42] S. R. Madden et al. TAG: a Tiny AGgregation Service for Ad-Hoc Sensor Networks. In *Proc. of OSDI*, Dec. 2002.
- [43] S. R. Madden et al. The Design of an Acquisitional Query Processor for Sensor Networks. In *Proceedings, International Conference on Management of Data (SIGMOD)*, 2003.
- [44] M. F. Mokbel et al. PLACE: A Query Processor for Handling Real-time Spatio-temporal Data Streams. In *Proceedings, International Conference on Very Large Data Bases*.
- [45] I. Motakis and C. Zaniolo. Formal Semantics for Composite Temporal Events in Active Database Rules. *Journal of System Integration*, 7(3-4):291–325, 1997.
- [46] I. Motakis and C. Zaniolo. Temporal Aggregation in Active Database Rules. In *Proceedings, International Conference on Management of Data (SIGMOD)*, pages 440–451, 1997.
- [47] R. Motwani et al. Query Processing, Resource Management, and Approximation in a Data Stream Management System. In *Proceedings, Conference on Innovative Data Systems Research*, Jan. 2003.
- [48] N. W. Paton. *Active Rules in Database Systems*. Springer, New York, 1999.
- [49] C. Roncancio. Toward Duration-Based, Constrained and Dynamic Event Types. In *Active, Real-Time, and Temporal Database Systems*, pages 176–193, 1997.

- [50] U. Schreier et al. Alert: An Architecture for Transforming a Passive DBMS into an Active DBMS. In *Proceedings, International Conference on Very Large Data Bases*, 1991.
- [51] P. Seshadri, M. Livny, and R. Ramakrishnan. The Design and Implementation of a Sequence Database System. In *Proceedings, International Conference on Very Large Data Bases*, pages 99–110, 1996.
- [52] U. Srivastava and J. Widom. Memory-Limited Execution of Windowed Stream Joins. In *Proceedings, International Conference on Very Large Data Bases*, Sep. 2004.
- [53] N. Tatbul et al. Load Shedding in a Data Stream Manager. In *Proceedings, International Conference on Very Large Data Bases*, Sep. 2003.
- [54] J. Widom and S. Ceri. *Active Database Systems: Triggers and Rules*. Morgan Kaufmann Publishers, Inc., 1996.
- [55] Y. Yao and J. E. Gehrke. Query Processing in Sensor Networks. In *Proceedings, Conference on Innovative Data Systems Research*, Jan. 2003.
- [56] C. Zaniolo et al. Stream Mill, Available [Online]: <http://wis.cs.ucla.edu/stream-mill/index.html>.