Anatomy of a Data Stream Management System*

Qingchun Jiang and Sharma Chakravarthy

Department of Computer Science & Engineering
The University of Texas at Arlington
Arlington, TX 76019
{jiang, sharma}@cse.uta.edu

Abstract. In this paper, we identify issues and present solutions developed – both theoretical and experimental – during the course of developing a data stream management system (DSMS) for applications in a sensor environment. Specifically, we summarize our solutions for CQ processing, trigger mechanisms, and Quality of Service (QoS) management in a stream data processing system. Specifically, we first present our query processing model and then discuss our scheduling strategies to support different requirements of stream applications. We further discuss our QoS framework in such a DSMS and present our load shedding techniques, queueing model, and analysis techniques in order to effectively deliver QoS requirements for those applications. Finally, we discuss our integration model to enhance the active capability of a DSMS.

1 Introduction

The main issues encountered in the design of a data stream processing system (DSMS) used in sensor and other applications environments include:

1. Both the number of data streams and the amount of data are large. Various computer programs (agents) in the system generate data at regular or irregular intervals. numerous hardware sensors distributed at different places emit sensor readings.
2. Stream-based applications have different requirements for the underlying stream data processing system. The monitoring component requires query processing system to detect user-defined events in a timely manner. Although the applications in this category can accept approximate results (since the impact of one missing observation or wrong observation is limited on the final pattern/event recognition), they have a near real-time requirement to detect any defined events because they have to trigger a series of actions based on the detected event in a timely manner. The second category of applications such as decision making components require accurate results and has a real-time processing requirement. The consequence of one missing event can be

* This work was supported, in part, by NSF grants ITR 0121297, IIS-0326505, EIA-0216500, and IIS 0534611

significant. The remaining applications have more tolerant requirements for the accuracy of query results or the on-line processing time.
3. The types of queries to be handled include: continuous queries (CQs) – which run in the system for ever theoretically with one submission, and one time ad-hoc queries which are analogous to queries used in a traditional database management system.

In summary, the main issues we have to address are:

1. bursty inputs, which introduce consequently bursty and unpredictable load in the system;
2. efficient processing of a large number of queries and the capability to support large number of triggers;
3. quality of service (QoS) requirements, which include the tolerance for the accuracy of query results, and tuple latency;
4. scalability, which allows the system to scale in order to handle a smart-environment (e.g., MavHome [5]).

In the rest of the paper, we present our initial work on the design of a data stream processing system to address the above issues.

1.1 Related Work

A number of projects that focus on stream data processing share similar objectives. The NIAGARA system [4] proposes an architecture for CQs with group optimization techniques. The Fjord architecture [8] supports both a continuous data stream and a traditional static data set by connecting the push-based operators with the pull-based operators via queues. The STREAM project [1] is trying to build a general data processing architecture that can support the functionalities of both database management system (DBMS) and data stream management system (DSMS). The Aurora system [3] presents an architecture to process data streams with some QoS requirements by decoupling a CQ into a few pre-defined operators. The differences between these projects and our work is that our system not only has the capability to support CQs over stream data with QoS constraints, but also provides full support for active capability through a set of rich event-condition-action (ECA) rules. Specifically, 1*)* we present a family of scheduling strategies to meet different requirements of stream applications, ranging from memory-optimized path capacity scheduling strategy to latency-optimized segment scheduling strategy. 2*)* We also present a set of mechanisms and algorithms to effectively deliver QoS requirements of those applications. These algorithms are different from those developed in Aurora and STREAM in that: the algorithms do not depend on specific system parameters such as CPU cycles used in Aurora and our load shedding mechanism used collaboratively with our scheduling strategies provides an effective mechanism to resource allocation and management. 3*)* We present an integrated stream and even processing model to enhance the capability of data stream processing systems to include complex event processing..

2 System Architecture

Figure 1 shows the system architecture of a stream processing system for sensor applications. The system consists of six components: `data source manager`, `query processing engine`, `catalog manager`, `scheduler`, `QoS manager`, and `ECA manager`. The `data source manager` accepts continuous data streams and puts input tuples into corresponding input queues of query plans. It also monitors various input characteristics of a stream and data stream characteristics (i.e., sortedness). Those characteristics provide useful information for query optimization, query scheduling, and QoS management. The `query processing engine` is in charge of generating query plans and of optimizing query plans dynamically. It supports both CQs and one-time ad-hoc queries. The `catalog manager` stores and manages the meta data in the system, including stream meta data, detailed query plans, and resource information. The `scheduler` determines which query or operator to execute at any time slot due to the continuous nature of the queries in the system. A few scheduling strategies are proposed and discussed in the next section. Due to the fact that most of stream-based applications have different QoS requirements, the `QoS manager` employs various QoS delivery mechanisms (i.e., load shedding, admission control, and so on) to guarantee the QoS requirements of various queries. In a sensor environment, to capture and understand the physical environment, monitoring changes through CQs is necessary and important. However, it is equally important to react to those changes immediately. The `ECA manager` in our system is used to detect complex events and to respond to those events using predefined actions. We discuss the details of various components of our system in following sections.

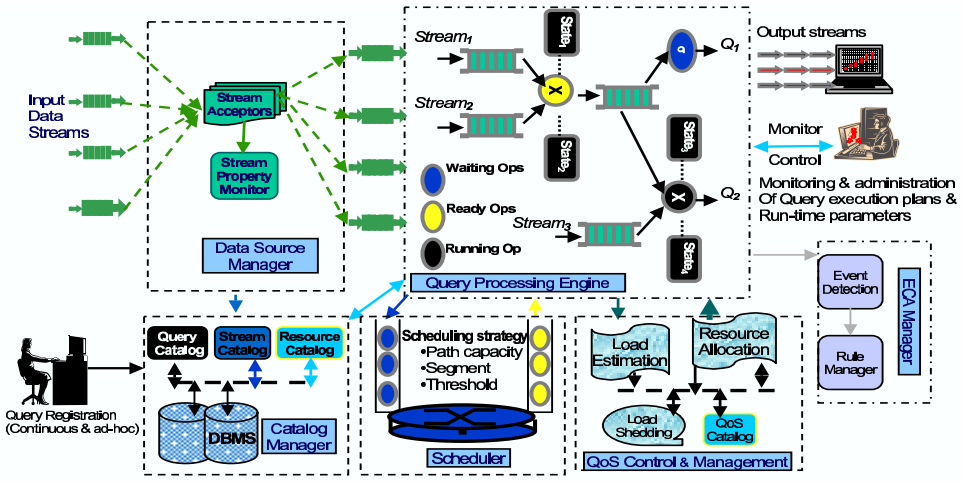

Fig. 1. Architecture of Continuous Query Processing System

3 Query Processing Model

3.1 Continuous Query Processing Model

The main computation required in a general query processing model based on data streams is to incrementally compute queries against arriving tuples and meet predefined QoS requirements of each query. This implies that a CQ processing model is some what different from a traditional query processing model in several aspects: 1*)* It requires queries to stay in the system as long as its input data streams are not terminated. Those queries are called `CQs` as opposed to one-time queries in traditional DBMSs. However, a CQ may only provide approximate query results as its QoS specification requires. 2*)* Because of the long life of CQs and the bursty input characteristics of data streams, a DSMS has to employ different scheduling strategies to allocate system resources such as memory, CPU cycles in order to satisfy the requirements of different applications. 3*)* The predefined QoS requirements from applications require that CQ processing systems compute final results correctly and also meet certain QoS requirements (i.e., tuple latency, precision, and so on). The final computed query results that do not meet predefined QoS requirements may not meaningful . As a result, a CQ processing system typically has three basic components: query processing engine, query/operator scheduler, and QoS control and management.

Our query processing model consists of a set of operator trees corresponding to query plans and the whole CQ processing system forms a data-flow diagram. A node in such a data-flow diagram corresponds to an operator of a query plan. Therefore, the basic job of our query processing system is to push each tuple from incoming data streams through a set of data transformation operations, and ultimately, the output streams are presented to applications that are represented as the root node in this data-flow diagram. We further define an `operator path` as a path starting from the first operator that a data stream enters and ending with the root node. In the rest of the paper, we present our solutions for each component in the DSMS.

4 Query Scheduling

In the literature, a few scheduling strategies have been proposed. The Chain scheduling strategy [2] is a near optimal scheduling strategy in terms of total internal queue size. However, we believe that the tuple latency and the memory requirement are equally important for a stream processing system. And the tuple latency is especially important for a query processing system where its applications have to respond in a near real-time manner. This near real-time response requirements motivate us to propose a family of scheduling strategies. Specifically, we first propose the `path capacity strategy` (PCS), which is an optimal scheduling strategy in terms of overall tuple latency. However, it has much bigger memory requirement than the Chain scheduling strategy. To overcome the large memory requirement of PCS, we further propose the `segment strategy` and its variant – the `simplified segment strategy`. These two strategies achieve a much better tuple latency than Chain strategy with a little larger memory requirement.

Path Capacity Strategy: Before we present our PCS, we define *Operator path processing capacity* $C^P_{P_i}$ as the number of tuples that can be consumed within one time unit by the operator path P_i. Therefore, the operator path processing capacity depends not only on the processing capacity of an individual operator, but also on the selectivity of these operators and the number of operators in the path. For an operator path P_i with k operators, its processing capacity can be derived from the processing capacities of the operators that are along its path, as follows:

$$C^P_{P_i} = \frac{1}{\frac{1}{C^P_{O_1}} + \frac{\sigma_1}{C^P_{O_2}} + \cdots + \frac{\prod_{j=1}^{k-1}\sigma_j}{C^P_{O_k}}} \tag{4.1}$$

where $O_l, 1 \leq l \leq k$ is the l_{th} operator along P_i starting from the leaf node. The denominator in (4.1) is the total service time for the path P_i to serve one tuple; and the general item $(\prod_{j=1}^{h}\sigma_j)/C^P_{O_k}$ is the service time at the $(h+1)^{th}$ operator to serve the output part of the tuple from the h^{th} operator along the path, where $1 \leq h \leq k-1$. Segment processing capacity has the same definition as operator path processing capacity except that the number of operator along a segment is different.

Path Capacity Strategy: *At any time instant, consider all the operator paths that have input tuples waiting in their queues in the system, schedule a single time unit for the operator path with largest processing capacity to serve until its input queue is empty or there exists an operator path which has a non-null input queue and a larger processing capacity than the currently scheduled one. If there are multiple such paths, select the one with the largest processing capacity. If there are multiple paths with the largest processing capacity, select one arbitrarily. The bottom-up operator scheduling strategy is used to schedule the operators of the chosen path.*

The PCS is a static priority scheduling strategy. The priority of an operator path is its processing capacity, which is completely determined by the number of operators along its path, and the selectivity and the processing capacity of each individual operator. Therefore, the priority of an operator path does not change over time until we revise selectivity of operators. The scheduling cost is minimized and can be negligible. Most importantly, the PCS has the following two optimal properties that are critical for a multiple CQ processing system.

Theorem 1 *The PCS is an optimal one in terms of the total tuple latency or the average tuple latency among all the scheduling strategies.*

Theorem 2 *Any other path-based scheduling strategy requires at least as much memory as that required by the PCS at any time instant in the system.*

The detailed proofs of Theorem 1 and 2 are presented in [7].

Segment Scheduling Strategy: Although the PCS has optimal memory requirement among all path-based scheduling strategies, it still buffers all unprocessed tuples at the beginning of an operator path. In a query processing

system with a shortage of main memory, a trade off exists between the tuple latency and the total internal queue size. The `segment scheduling strategy` employs a similar strategy as the Path Capacity strategy except that the segment scheduling strategy schedules the operator segment with the biggest processing capacity rather than the operator path with the biggest processing capacity. Therefore, the segment scheduling strategy can minimize the memory requirement of our query processing system. The operator segments are constructed by the following algorithm.

Segment Scheduling Strategy: *At any time instant, consider all the operator segments that have input tuples waiting in their input queues. Schedule a single time unit for the operator segment which has the maximal memory release capacity to serve until its input queue is empty or there exists another operator segment which has a non-null input queue and a larger memory release capacity than the currently scheduled one. If there are multiple such segments, select the one with the largest memory release capacity. If there are multiple segments with the largest memory release capacity, select one arbitrarily. A bottom-up operator scheduling strategy is used to schedule the operators of the chose segment.*

Simplified Segment Scheduling Strategy: Simplified segment strategy differs from segment scheduling strategy in that it employs a different segment construction algorithm. In a practical multiple query processing system, we observe that: (*a*) the number of segments constructed by the segment construction algorithm may not be significantly less than the number of operators presented in the query processing system and (*b*) the leaf nodes are the operators that have faster processing capacities and less selectivity in the system; all the other operators in a query plan have a much slower processing rate than the leaf nodes. Based on these facts, we partition an operator path into at most two segments, rather than a few segments. The first segment includes the leaf node and its consecutive operators such that $\forall i, C^M_{O_{i+1}}/C^M_{O_i} \geq \gamma$, where γ is a similarity factor. In the previous segment construction algorithm, it implicitly states $\gamma = 1$. In the simplified segment strategy, the value of γ used is less than 1 in order to decrease the number of segments. In our system, we have used $\gamma = 3/4$ in our experiments. The remaining operators along that operator path, if any, form the second segment.

The simplified segment strategy has the following advantages: (*i*) Its memory requirement is only slightly larger than the segment strategy; (*ii*) The tuple latency significantly decreases because the number of times a tuple is buffered along an operator path is at most two; (*iii*) The scheduling overhead significantly decreases as well due to the decrease in the number of segments; and (*iv*) It is less sensitive to the selectivity and service time of an operator due to the similarity factor, which makes it more applicable.

5 QoS Control and Management

Quality of Service is one of the main factors that distinguishes a DSMS from a traditional DBMS. Each application can specify its own QoS metrics such as

latency and accuracy-tolerance. The key problem is to deliver those QoS requirements. Currently, the mechanisms that we have been exploring are: (*i*) multi-query optimization, which is an important mechanism to deliver QoS requirement. For example, in this approach, QoS can be satisfied through sharing a common sub expression to decrease the system load or dynamically adapting a query plan to trade off the resource requirement and the QoS requirement. (*ii*) memory allocation: due to the fact that approximate results are applicable for most of stream-based applications, we are able to prevent from violating the maximal approximation error an operator can introduce by reserving the minimal buffer space for that operator. (*iii*) scheduling strategy, which is one mechanism to control QoS management. It assigns a higher priority to the operators that have to meet a better QoS requirement and a lower priority to those which have a lower QoS requirement; (*iv*) load shedding, which is an explicit mechanism at tuple level to deliver QoS requirements. It dynamically inserts a drop operator, which drops unprocessed or partially processed tuples in a random manner or based on some semantics, in an existing query plan during heavy load periods and removes the drop operator when system load decreases to a certain level. (*v*) admission control, which works at the query level, is the final choice to satisfy the QoS requirements of those active queries in the system. In the worst case, the system cannot guarantee the QoS requirement even it sheds all the load that it can shed without violating the maximal tolerant approximation errors. When this situation happens, the only solution is to choose some victim queries in the system and to disable them during a high load period.

For the last two mechanisms, a fundamental problem is to efficiently estimate the system load, based on which the system has to determine when to activate load shedding, how much load to shed, and whether to accept a new query into the system or not. In the following section, we present our solution for these problems.

5.1 Estimation of System Load

For each query, assume a maximal tolerant latency (MTL) requirement. Without loss of generality, m active queries in the system can be further decomposed into k operator paths $\mathcal{P} = \{p_1, p_1, \cdots, p_k\}$. To prevent a query from violating its MTL , we have to guarantee that the output results from each of its operator paths do not violate its MTL, which implies that each operator path has a MTL requirement, which is the MTL of its query. For any operator path p_i, the query processing system has to process all the tuples that arrived during the last l_i time units if its MTL is l_i time units no matter what scheduling strategy it employs. It may schedule some operators along the path multiple times in the last l_i time units, or schedule some operators more often than others. But the age of the oldest unprocessed or partially processed tuple left in the queues along that operator path must be less than its MTL. Therefore, without considering the cost of scheduling, the minimal computation time $\mathcal{T}_i$ required by operator

path p_i within l_i time units is

$$\mathcal{T}_i = \frac{\int_{t-l_i}^{t} v_k(t)d_t}{C_i}; \qquad 1 \leq i \leq k \tag{5.1}$$

where $v_k(t)$ is the input rate of its input stream, and C_k is its processing capacity. The equation (5.1) gives the minimal absolute computation time units the operator path p_i required within its MTL. Furthermore, the percentage of computation time units ϕ_i it requires is,

$$\phi_i = \mathcal{T}_i / l_i \tag{5.2}$$

Without considering the sharing segments among the operator paths, the total percentage of computation time units Φ for a query processing system with k operator paths is:

$$\Phi = \sum_{i=1}^{k} \phi_i = \sum_{i=1}^{k} \frac{\int_{t-l_i}^{t} v_k(t)d_t}{C_i l_i} \tag{5.3}$$

Due to the fact that the MTL of a query is no more than a few seconds, we can expect that the input rate during its MTL can be captured by its mean input rate. Then the equation (5.3) can be approximated as:

$$\Phi \approx \sum_{i=1}^{k} \frac{\bar{v}_k l_i}{C_i l_i} = \sum_{i=1}^{k} \frac{\bar{v}_k}{C_i} \tag{5.4}$$

where $\bar{v}_k$ is the mean input rate of the input stream of the operator path p_k within a period of time of its MTL.

When there are some sharing segments among the operator paths, we have to deduct the over-counted parts. Assume that there are g sharing segments in the system, and that each of them is shared f_i times, where $f_i \geq 2$ and $1 \leq i \leq g$, then

$$\Phi \approx \sum_{i=1}^{k} \frac{\bar{v}_k}{C_i} - \sum_{i=1}^{g} (f_i - 1)\frac{\bar{v}_i}{C_i^S} \tag{5.5}$$

where C_i^S is the processing capacity of the sharing segment.

From (5.5), we can efficiently estimate the system load through monitoring the input rates of all input streams. If $\Phi > 1$, we know that the system is out of computational resources, we have to activate either load shedding mechanism or admission control mechanism to prevent the system from violating the predefined QoS requirements.

5.2 Load Shedding

The load shedding is implemented as insertion of a drop operator into a victim query plan. Two kinds of drop operators have been proposed: a random drop operator and a semantic drop operator. The random drop operator is implemented as a p `gate` function: for each tuple, it generates a random value $\acute{p}$. The tuple passes to the next operator if $\acute{p} \geq p$. Otherwise, it is discarded. The semantic

drop operator discards tuples based on a condition whose selectivity corresponds to $1 - p$, it requires some specific-application information from a query and its application domain. We choose the randomly drop operator as our load shedder in our system.

Without knowing the specific query/domain information, we can assume that all tuples in the same data streams have the same importance towards the accuracy of the final results of a query. Each query has an maximal relative error which it can tolerate. Based on this, we discuss how to find a best place for a shedder,and how to allocate the load that we have to shed among those shedders.

When to shed From section 5.1, we are able to predict the system load in the next few time units through monitoring current input rates of all input streams in the system. Therefore, the problem of when to shed can be solved by simply periodically predicting the system load every few time units. When total system load is going to exceed 100% of its capacity, then we know that the system is going to experience a overload period, which will violate some predefined QoS requirements. Therefore, we have to shed some load. The amount of load we have to shed equals the amount of system load that is beyond the system capacity.

Placement of load shedders A load shedder can be placed in any location along an operator path. However, each location has a different impact on the accuracy of the final results and on the amount of computation time units it releases. Placing a load shedder earliest in a query plan (i.e., before a leaf operator) is the most effective one in decreasing the amount of computation time units, but it is not the most effective one in terms of the effect on the accuracy. Therefore, the best potential location for a load shedder along an operator path is the place where the load shedder is capable of releasing maximal computational time units while introducing the minimal relative errors in the final query results by dropping one tuple there. However, the relative error introduced by dropping one tuple before an operator is unknown and difficult to estimate. For example, it is impossible to estimate the error in final results introduced by dropping a tuple from an input queue of a join operator without having any apriori knowledge of the input tuples and the applications that depend on this query. In the following section, we assume that each tuple from a data stream has an equal impaction on accuracy of final query results of a query.

For each operator path of a query plan, there exists a candidate load shedder. To find the most effective candidate place of a load shedder among an operator path, our solution is to compute all potential places of that load shedder among the path. The best place along that path is the place where the load shedder has maximal place weight. The place weight of a load shedder is defined as the percentage of computation time units α it can save to the relative error ϵ in its final results introduced by that load shedder through dropping one tuple for one time unit there. Let $v(d)$ be the maximal drop rate that a load shedder can introduce at a potential place, then

$$W = \frac{\alpha}{\epsilon};\ \text{where } \epsilon = \frac{v(d)}{v_{shedder}} \text{ and } \alpha = \frac{v(d)}{C^S} - \frac{v_{shedder}}{C^O_{shedder}} \tag{5.6}$$

where C^S is the processing capacity of the segment starting from the operator right after the load shedder until the root node (excluding the root node) along the operator path; $C^O_{shedder}$ is the processing capacity of the load shedder; $v_{shedder}$ is the input rate of the load shedder, and $v_{shedder} = v \prod_{i=x_1}^{x_n}(\sigma_i)$, x_1 to x_n are the operators before the load shedder starting from leaf operator, and σ_i is the selectivity of the operator x_i, and v is the input rate of the input stream of the operator path.

For a query plan with k operator paths, we compute the best candidate place for each load shedder among its k paths. However, we have to merge some load shedder into one shedder due to the presence of multi-way operators and the sharing segment(s) among multiple-queries. However, each shedder has its own limitation to maximal load it can shed, which is determined by both the maximal tolerant relative error in the final results of a query plan and its place along an operator path.

Allocation of shedding load among load shedders We find a list of candidate load shedders for each query plan in the system. The maximal relative error in final query results each load shedder can introduced is the maximal relative error limitation of the query plan to which it belongs. We also find the total load that we have to shed in order to avoid violating the pre-defined QoS requirements in the system. Now, the problem is to allocate the total load that we have to shed among all the candidate load shedders in the system with a goal of minimizing total relative errors introduced in the final query results.

By considering the total percentage of computation time units Δ we have to release by load shedding as the total capacity of the ship, and the relative error ϵ_i introduced by a shedder as the weight of an item, and the load α_i saved by a shedder as the total value of the item, the problem of allocation of shedding load among load shedders is the well known `knapsack problem`. The 0-1 knapsack problem is a NP-hard problem, while the fractional knapsack problem is solvable by a greedy strategy in $O(n \lg n)$ time, where n is the total number of candidate shedders. In our case, fortunately, it is a fractional knapsack problem. Therefore, the allocation of shedding load problem can be solved as following: Since we aim to minimize the total relative errors introduced by load shedding, it is easy to see that an optimal solution will to choose a shedder with a biggest place weight in the system and to shed maximal load there. If the total load still exceeds system capacity, we in turn choose the shedder that has the second latest place weight, and shed maximal load it can shed there. We repeat this procedure until the total load in the system is less than its capacity.

5.3 Estimation of Tuple Latency

The effective estimations of both the tuple latency and the memory requirement of a query plan are fundamental problems in our system because they provide approximate but important information for us to determine: (a) whether the predefined QoS requirements of a query plan will be satisfied or not (in advance), which consequently determines whether the system has the capability to accept new queries or not; (b) which scheduling strategy is the best choice. Some strategies can achieve minimal memory requirement, while others can achieve minimal

tuple latency. When the estimated total memory requirement is a bottleneck in the system, a scheduling strategy with a goal of minimizing memory requirement is the best choice; otherwise, a strategy which can achieve minimal tuple latency is the best choice; (c) when to start the load shedding. When the estimated tuple latency exceeds the predefined QoS requirement, and the best suitable scheduling strategy cannot recover it back to a normal scenario, then load shedding is the best choice.

The main idea behind estimating the tuple latency and the memory requirement of a query plan is to model each operator in the system as a queueing system, in which the functionality of the operator is modeled as service facility of a queueing system, and its input and output buffers are modeled as the input and the output queue, respectively. As a result, the whole query processing system is modeled as a queueing network. Furthermore, an operator in the system works in two modes, vacation mode and busy mode, as follows: once the operator gains the processor, which is determined by a particular scheduling algorithm, if its input queue is empty, the operator goes to vacation immediately. Otherwise, the processor needs a setup time and then serves a certain number of tuples determined by a specific service discipline, such as gated-service or exhaustive-service, using a first-come-first-served (FCFS) order, and the service of a tuple is non-preemptive. After that, the processor becomes unavailable (to serve other operators or to handle other tasks) for a vacation period and then returns for further service. So each operator in the query plan is modeled as a queueing system with vacation period and setup period.

In this model of queueing network, the queueing systems can be categorized into three classes based on their inputs:

1. external input(s) queueing system. Intuitively, this class has only external input(s) from a continuous data stream. Whether it is in a vacation period or a serving period, the input tuple is inserted into its input queue immediately whenever it arrives.
2. internal input(s) queueing system. The arrival time of a tuple from an internal input is the departure time of the output process of another operator. Therefore, this class only has inputs during its vacation period, and there is no input during its setup period and service period.
3. external input and internal input queueing system. This class has both internal input and external input, and its inputs are a combination of the above two classes.

If we are able to determine the vacation period and the busy period of each operator in the system, we are able to compute the queue size in its input queue and the tuple waiting time at that operator (queue waiting time plus the processing time) based on a Poisson input model. The tuple latency of a query plan is the sum of the mean waiting times at all the operators along its operator path. For additional details, please refer to [6].

5.4 Stream Property Monitor

In order to estimate system load and tuple latency, we have to know the input rates of all input streams. Therefore, we design a stream property monitor to monitor not only the arrival characteristics, but also the data properties of a data stream. To monitor the input rate of an input stream efficiently, we sample a series of points over the time axis. If the input rates at the last n points have not changed too much (the change is measured by the variance of those input rates), we decrease the sample rate, otherwise, we increase the sample rate. This adaptive algorithm makes it possible to monitor hundreds of data streams efficiently and effectively. A data stream has various properties such as sortedness, clustering, and so on. Each of them has a different impact on query optimization. Due to the fact that some properties are well-defined on some domains while others are not defined or even not applicable, currently there is no a standard set of data properties to monitor. We monitor the sortedness for all data streams and clustering only when it is applicable. The sortedness of a data stream is monitored according to a timestamp attribute or a counter attribute. The clustering is defined as a group of consecutive tuples which have a common value on a specific attribute. As this definition is too rigorous for some applications, a relaxed definition – that of a group of tuples which share a common value on a specific attribute, and the distance[1] between two continuous tuples is no more than a threshold value k can be used. Detailed information about how those properties can benefit query processing and how to monitor those properties are presented in papers [9]. It is worth noting that in a particular domain, there may have more properties applicable, which can provide more useful information for query processing.

6 Event Processing and Rule processing

The capability of detecting composite events and supporting a large number of triggers is another important problem in order to respond to any abnormal patterns or user-defined events in a timely manner. Traditionally, the Event-Condition-Action (or ECA) model is used to detect composite events and process associated rules. However, *1*) the ECA model does not include does not include stream processing and current stream processing systems have little or no support for event detection and rule processing. *2*) both the ECA model and stream processing model employ a similar data-flow processing model for their computations. Therefore, it is only natural to synthesize these two models and combine their strengths into an integrated model.

6.1 An Integrated Model For Event Processing

Both ECA and stream processing models have their limitations for handling applications that require both stream and event processing. The ECA model has its strength in expressing and processing composite events. On the other

[1] The distance between two tuples can be defined as the number of tuples between them or the time difference of the two time stamps. It is application specific.

hand, the stream processing model is geared towards complex computation, but it lacks event processing (e.g., to detect a sequence of events) and expressive event specification capability. The proposed integrated model consists of three stages: (*1*) CQ processing stage used for computing continuous queries over data streams, (*2*) event processing stage that is used for detecting events (both primitive and composite), and (*3*) rule processing stage to check conditions and trigger pre-defined actions.

In our integrated model, the output of a CQ can be specified as a primitive event. A timestamp is added to the output of a CQ to be used for event detection. We have added stream modifiers to detect expressive changes between tuples in a steam before they are sent to event processing. The window concept has been generalized to create windows that are computation-based (as opposed to temporal or tuple-based). The simplicity of our integrated model is due the compatibility of the event detection model in Snoop and the widely-used stream processing model .

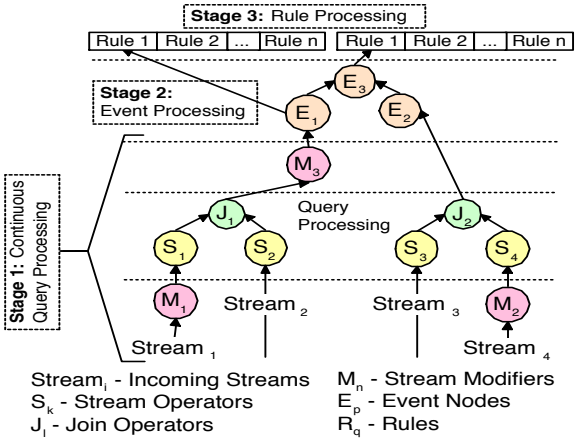

Fig. 2. Three Stage Integration Model

6.2 Continuous Query (CQ) Processing Stage

The CQ processing stage in our integration model processes normal CQs; it takes streams as inputs and outputs computed continuous streams. However, we enhance CQs to support more complicated computation requirements of many stream applications. None of the enhancements affect the operator semantics, scheduling algorithms, QoS delivery mechanisms, and any other components proposed for stream data processing. First, in order to express computations more clearly, CQs are named. The name of a continuous query 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.

Second, 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.

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 data stream. A stream modifier is denoted by

$$M(< s_1, s_2, \cdots, s_i > [, P < pseudo >][, O|N < v_1, v_2, \cdots, v_j] >)$$

where M is called the modifier function that computes a particular kind of change. The i-tuple $< s_1, s_2, \cdots, s_i >$ is the parameter required by the modifier function M. The following $P < pseduo >$ defines a pseudo value for the M function in order to prevent underflow. The following j-tuple element is called the untouched attributes that need to be output without any change. The $O|N$ part is called modifier profile, which determines the oldest values or the latest values of the j-tuple that need to be output. If O is specified, the oldest values are output and 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 definition of a stream modifier. Currently, we have implemented the following three commonly used stream modifiers in our system.

1. ADiff() is used to detect absolute changes over two consecutive states;
2. RDiff() is used to detect the relative changes over two consecutive states.
3. ASlope() is used to compute the slope ratio of two attributes over two consecutive sates;

6.3 Event Processing

Enhanced Events and Event Expressions: In a traditional event processing system, primitive events are associated with operators that modify the state of the system. can be of either class level or instance level (i.e., statically determined). In this model, the output of CQs (perhaps modified using event modifiers) are treated as events. Masks are used to further reduce the number of events generated by a CQ and to generate multiple events from the same CQ.

Inputs to Event Processing Stage: CQs output data streams in the form of tuples. These tuples are fed as the input event streams to the event processing stage. The event detection graphs (EDGs) of Snoop are used in the integrated model. 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. Any attribute of a tuple can be used in the event processing stage for condition checking, for masking the inputs to the event nodes, and for merging event streams.

Event Specification using Extended SQL: CQ processing model shares the DAG processing approach with ECA 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 syntax shown

below, where CREATE EVENT creates a named event, SELECT selects the attributes from the event_stream or event_expression, MASK applies some condition on the events that enter that node. Event_stream is either a CQ name or an event_expression that combines more than one event using event operators, and attributes from the tuples. In addition, event_stream can be replaced by the create CQ statement.

```
CREATE EVENT event_name
SELECT attribute as attribute_name, ...  MASK attribute_conditions
FROM (event_stream | event_expression)
```

Event nodes are created in the EDGs based on create event specifications. Output from the CQ is fed as inputs to the event nodes in the EDGs as event streams. In an EDG, leaf nodes represent primitive events and internal nodes represent composite events. In the integrated model, input to the event nodes can be created either by a CQ, from the underlying system or from an external source.

6.4 Rule Processing

The rule system is responsible for executing rules (consisting of conditions and actions). This section briefly describes how rules are defined, triggered, and executed.

Rule Definition: A rule is used to trigger predefined conditions and actions once its associated event is detected. Rule properties such as coupling mode, consumption mode and priority are provided, along with the event name and condition associated with the rule, and the action to be performed when conditions results are true. When a composite event uses this primitive event, only the event tuples that satisfy this condition are passed on to it. 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. Syntax for rule definition is shown below,

```
CREATE RULE rule_name [,coupling mode,consumption mode,priority]
                                              ON event_name
  RULE_CONDITION Begin; (Simple or Complex Condition); End;
  RULE_ACTION Begin; (Simple or Complex Action); End;
```

Where ON specifies the event associated with the rule, coupling mode can be IMMEDIATE, DEFERRED, and DETACHED, consumption modes can be Recent, Recent-Unique, Continuous, Cumulative, and Chronicle along with the window specification, and priority is a numeric value used to set rule priority.

7 Prototype Implementation

Currently, we have a prototype implementation of the proposed system in C++. It consists of a CQ engine, a resource manager, a stream manager, a query/operator scheduler, a load shedding manager, a QoS manager, and a rule manager.

The CQ engine compiles a named CQ into a query plan and registers it with the system. It is capable of processing various select-project-join queries with simple aggregate operators and stream modifiers presented in this paper. However, currently the query plans in the system are static query plans, which are not optimized and cannot adapt to changes in input characteristics and system load. 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 window type of a stream operator is implemented as a semantic window, which can be specified using proposed semantic expression proposed in this paper in a CQ.

The ECA manager and rule processing component brings the event and rule processing components implemented in our Sentinel system to our MavStream system. Its basic role includes management of various ECA rules and provide an effective means to continuously detect events, evaluate conditions, and execute per-defined actions once it detects corresponding events.

8 Conclusion and Future Work

In this paper, we have identified a number of issues related to CQ processing in a sensor environment, and presented our solutions for these issues. First, a family of scheduling algorithms are discussed. Then the load shedding approaches are discussed. Finally, the integrated model is presented that combines complex event processing with stream processing. The integrated model greatly enhances current stream data model horizontally and vertically to combine event processing and rule processing with stream processing.

References

1. S. Babu and J. Widom. Continuous queries over data streams. In *SIGMOD Record*, pages 109–120, Sept. 2001.
2. B. Brian, B. Shivnath, et al. Chain: Operator scheduling for memory minimization in stream systems. In *Proc. of SIGMOD*, June 2003.
3. D. Carney, U. Cetintemel, et al. Monitoring streams - a new class of data management applications. In *Proc. of VLDB*, Sept. 2002.
4. J. Chen, D. Dewitt, F. Tian, and Y. Wang. Niagaracq: A scalable continuous query system for internet databases. In *Proc. of SIGMOD*, June 2000.
5. D. Cook and et al. Mavhome: An agent-based smart home. In *Proceedings of the Conference on Pervasive Computing*, 2003.
6. Q. Jiang and S. Chakravarthy. Queueing analysis of relational operators for continuous data streams. In *Proc. of CIKM*, Nov. 2003.
7. Q. Jiang and S. Chakravarthy. Scheduling strategies for processing continuous queries over streams. In *Proc. of BNCOD*, July 2004.
8. S. Madden and M. J. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *Proc. of ICDE*, 2002.
9. U. S. S. Babu and J. Widom. Exploiting k-constraints to reduce memory overhead in continuous queries over data streams. *Technical Report, http://dbpubs.stanford.edu:8090/pub/2002-52*, 2002.