Runtime Optimization of Continuous Queries*

Balakumar Kendai and Sharma Chakravarthy
IT Laboratory and Department of Computer Science & Engineering
The University of Texas at Arlington, Arlington, TX 76019.
sharma@cse.uta.edu

Abstract

In data stream processing systems, Quality of Service (or QoS) requirements, as specified by users, are extremely important. Unlike in a database management system (DBMS), a query in a data stream management system (DSMS) cannot be optimized once and executed. It has been shown that different scheduling strategies are useful in trading tuple latency requirements with memory and throughput requirements. In addition, DSMSs may experience significant fluctuations in input rates. In order to meet the QoS requirements of data stream processing, a runtime optimizer equipped with several scheduling and load shedding strategies is critical. This entails monitoring of QoS measures at run-time to dynamically modify the processing of the queries at runtime to meet the QoS requirements.

This paper addresses runtime optimization issues for MavStream, a data stream management system (DSMS). The runtime optimizer presented in this paper matches the output (latency, memory, and throughput) of a continuous query (CQ) with its QoS requirements. Alternative scheduling strategies are chosen as needed based on the runtime feedback. A decision table is used to choose a scheduling strategy based on the priorities of QoS requirements and their violation. The decision table approach allows us to add new scheduling strategies as well as compute the strategy to be used in an extensible manner. Additionally, load shedders are activated and deactivated by the runtime optimizer to meet QoS requirements beyond adjusting scheduling strategies..

1 INTRODUCTION

Sensors and hand held devices generate ubiquitous data. Furthermore, the size of this data is unbounded and can be considered as a relation with infinite tuples (not stored on a secondary device as in traditional database management systems or DBMSs). This data created by sensors are called data streams [9, 5, 13]. Examples of applications [7] that have streaming input are network monitoring, stock tickers and variable tolling in highways. Most stream based applications require the results to be produced within a specific amount of time (termed tuple latency). This and other (such as memory usage and throughput) requirements, known as Quality of Service (QoS), necessitate the need for the data to be processed on the fly as they arrive. The large amount of time required for secondary storage access and lack of QoS support in DBMSs rule out the possibility using a traditional DBMS. Though main memory databases process data without storing them on a secondary storage, they assume the data to be readily available, which is not the case with data streams. These characteristics of data streams have entailed the development of specialized applications for handling stream data and are termed Data Stream Management Systems (DSMS) [12, 2, 14].

Stream data can arrive at a very high rate and furthermore, it can fluctuate quite over periods of time. This necessitates buffering [15] as there may not be enough capacity in a DSMS to process all incoming data without buffering. Absence of buffering can lead to loss of tuples. The amount of memory available in a system is crucial for stream applications as most of the data required for processing is stored in main memory. Since main memory is always limited in a system, there is always a possibility of memory overflow. Research in this field has proposed techniques such as storing excess tuples into secondary storage [2] and scheduling strategies [18, 11, 16, 10, 4] aimed at reducing the amount of tuples that reside in the memory. Mechanisms have also been developed to reduce the memory requirement of join and aggregate operators which operate on windows by using histograms, timeout, slack, and discarding tuples to reduce window sizes [3, 19].

The utility of results produced by a DSMS often depends on the delay with which it is produced. These constraints are generally specified as QoS requirements

*The work done in this paper is currently supported by NSF IIS - 0534611, NSF IIS - 0326505 and NSF EIA - 0216500.

International Conference on Management of Data
COMAD 2008, Mumbai, India, December 17–19, 2008

for a query. The three QoS measures that are mainly used in a DSMS are: tuple latency, memory utilization and throughput. Tuple latency is defined as the difference between the arrival and departure time of a tuple in the system. Memory utilization is the total memory usage of the tuples that reside in the queues of operators. Throughput is defined as the rate at which output is produced by the system. Various scheduling strategies have been proposed that improve the performance of a particular QoS measure given the unpredictable nature of data.

The unpredictable nature of stream data and QoS requirements of stream-based applications has nurtured research which has proposed various scheduling and approximation techniques. Specialized scheduling strategies [18, 11, 16, 10, 4] have been proposed for a DSMS. A continuous query issued to a DSMS can have multiple QoS constraints associated with it. Any particular strategy may perform exceedingly well for *one* QoS measure but it may not be equally good for other measures that can be part of the QoS requirements of a query. Since the data rate of streams can fluctuate drastically, it is not necessary to choose the best strategy for a particular measure. For example, when the memory utilization is within the QoS limits and arrival rate is low, it would be better to run the query in a strategy that tries to minimize tuple latency or any other strategy that performs better in QoS measures which are not met by the current strategy. As the choice of scheduling strategy and the arrival rates of the stream affect the performance of QoS measures, it is necessary to choose the best scheduling strategy at different periods during the lifetime of a query.

The paper presents the design, implementation, and experimentation of a runtime optimizer (RO) that monitors the output and uses scheduling and load shedding strategies to satisfy QoS requirements. The RO supports changing of scheduling strategies at runtime. The runtime optimizer also tries to minimize the number of switches between strategies based on heuristics. The two mechanisms of runtime optimization and load shedding, while not dependent on each other, work well in conjunction to enhance the overall performance of the system. A decision table is used to choose a scheduling strategy based on the priorities of QoS requirements and their violation. The decision table approach allows us to add new scheduling strategies as well as compute the strategy to be used in an extensible manner.

The rest of the paper is organized as follows. Section 2 discusses related work. 3 describes the functional architecture of MavStream. The design and the details of the runtime optimizer is presented in 4. Experimental analysis of the runtime optimizer is discussed in 5. 6 contains conclusions.

2 RELATED WORK

Stream [5] is a prototype implementation of a complete data stream management system being developed at Stanford. The system generates a query execution plan on registration of a query that is run continuously. Operators make use of synopsis (an internal data structure at an operator) to store intermediate results. Stream has a central scheduler that has the responsibility for scheduling operators. The scheduler dynamically determines time quantum of execution for each operator. Period of execution may be based on time, or on the number of tuples consumed or produced. Stream uses the chain scheduling strategy [4] with the goal of minimizing the total internal queue size. Load shedding techniques [6] proposed in Stream focuses only on aggregation queries over sliding windows. Aurora [2, 7] is a data flow system that uses the primitive box and arrow representation. Tuples flow from source to destination through the operational boxes. It can support continuous queries, ad-hoc queries and views at the same time. The QoS evaluator continually monitors system performance and activates the load shedder, which sheds load until the performance of the system matches user-specified values. Quality of Service is associated with the output. It is specified in terms of a two-dimensional graph that specifies the output in terms of several performance-related and quality-related services. Aurora handles high load situations by dropping load using a drop operator. Load shedding [20] is treated as an optimization problem and consists of determining when and where to shed load and how much to shed. The QoS requirements are specified as value utility graphs and loss tolerance graphs. The load shedding algorithm consists of load evaluation step where in system load is determined using load coefficients. Borealis [1] is a distributed stream processing engine that inherits the core stream processing functionality from Aurora.

Telegraph [12] consists of an extensible set of composable dataflow modules or operators that produce and consume records in a manner analogous to the operators used in traditional database query engines, or the modules used in composable network routers. The modules can be composed into multi-step dataflows, exchanging records via an API called Fjords [17]. The key advantage of Fjords is that they allow query plans to use a mixture of push and pull connections between modules, thereby being able to execute query plans over any combination of streaming and static data sources. Cougar [8] is specifically targeted to meet the requirements of sensor-based applications. Cougar focuses on a distributed approach toward query processing and determines the data that needs to be extracted from the sensors depending upon the workload. Cougar is based on the Cornell Predator object rela-

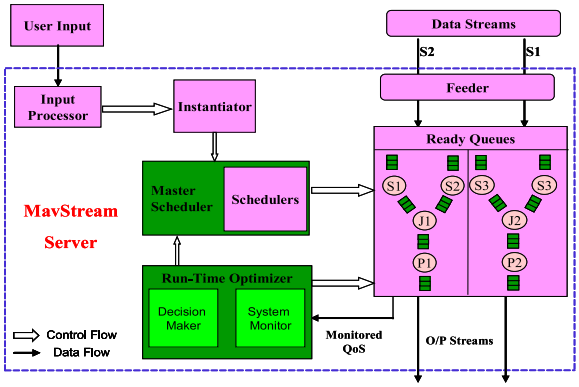

Figure 1: **MavStream Architecture**

tional database system. NiagaraCQ [14] is a system that mainly focuses on supporting continuous query processing over multiple, distributed XML files. NiagaraCQ uses a novel incremental group optimization strategy with dynamic re-grouping. It takes advantage of the fact that many web-based queries share similar structures. Grouping similar structures can save on the computation cost, memory used and the number of I/Os. When a new query arrives, the existing groups are considered as possible optimization choices instead of re-grouping all the queries in the system.

3 MavStream ARCHITECTURE

MavStream is a DSMS for processing continuous queries over data streams. MavStream is modeled as a client-server architecture in which client accepts input from the user and forwards it to the server. The various components of MavStream are shown in Fig. 1. The MavStream server upon receiving a query passes creates a query plan object. A query plan object is a tree of objects that contain information about all the operators of a query. The input processor uses the instantiator module to instantiate all the operators, paths, and segments [11]. The instantiated objects are put in to the appropriate ready queue based on the chosen scheduling strategy. The operators are scheduled using a scheduling strategy and output of the query is given back to the client. Below, the working of the scheduler is elaborated.

3.1 Scheduler

The scheduler is one of the critical components in MavStream. Scheduling is done at the operator level and not at the tuple level. It is not desirable to schedule at a tuple level as the number of tuples entering the system is very large. On the other hand, scheduling at the query level loses flexibility, as the granularity offered by the scheduler may not be acceptable. MavStream schedules operators based on their state and priority. The scheduler maintains a ready queue, which decides the order in which operators are scheduled. This queue is initially populated by the server. Operators must be in a ready state in order to get scheduled. Operator goes through a number of states while it is being scheduled. The following are the scheduling policies [11] supported by MavStream:

1. Round-Robin: All the operators are assigned the same priority (time quantum). Scheduling order is from leaves to parent nodes at the next level and is taken in the order stored in the ready queue. This policy is not likely to dynamically adapt to QoS requirements as all operators have the same priority.

2. Weighted round-robin: Here different time quanta are assigned to different operators based on their requirements. Operators are scheduled in a round robin manner, but some operators may get more time quantum over others. For example, operators at leaf nodes can be given more quantum of time as they are close to data sources. Similarly, Join operator, which is more complex and time consuming, is given higher priority than Select.

3. Path capacity scheduling: This strategy schedules the operator path which has the maximum processing capacity as long as there are tuples present in the base buffer of the operator path or there exists another operator path which has greater processing capacity than the presently scheduled operator path. This strategy is good for attaining the best tuple latency.

4. Segment scheduling: Schedule the segment which has the maximum memory release capacity as long as there are tuples present in the base buffer of the segment or there exists another segment which has greater memory release capacity than the presently scheduled segment. This strategy is good for attaining the lower memory utilization.

5. Simplified segment scheduling: The segments are constructed differently from the above. Instead of partitioning an operator path into many segments, we partition it into only two segments. This strategy takes slightly more memory than the segment strategy giving improvement in tuple latency.

The execution of all schedulers is controlled by the master scheduler. The master scheduler allocates time quantum to each scheduler. At any instance of time only one scheduler is allowed to run by the master scheduler. Different queries can be scheduled using different strategies. Also, scheduling can be changed at runtime to meet the output requirements.

3.2 Feeder

For experimental purpose, a feeder has been developed to feed tuples (given out by the stream sources) to

the buffers of leaf operators. If many streams from the sources are combined and given as one stream to the query processing system then the user should specify the split condition (using a split operator supported by MavStream) on the stream. Each stream is fed using a separate thread. Feeder thread reads the tuples from the secondary storage feeds the tuples to buffers associated with leaf operators. Presently there are no real streams used directly from the sensors. Hence we use flat files which contain synthetically generated data. The mean rate of feeder is changed over time and pauses to the feeding has also been introduced to simulate bursty nature of streams. The characteristics of feeding can be specified by a configuration file.

4 DESIGN OF RUNTIME OPTIMIZER

The primary goal of the runtime optimizer is to monitor QoS measures to make sure that user specified QoS values are met to the best extent possible. Based on the monitoring, we choose the best (or optimal) scheduling strategy for a query. In MavStream, the runtime optimizer acts like the decision making component of a closed loop feedback control mechanism, where *expected* QoS values of the reference output and *measured* QoS values representing the actual output are used. Runtime optimizer consists of a System Monitor, which monitors the values of QoS measures for a query, and a Decision Maker, which chooses the best scheduling strategy for a query and controls the load shedders. In this section, we discuss the issues encountered and assumptions made while designing the runtime optimizer.

4.1 Inputs to Runtime Optimizer

The goal is to match the user requirements with the resources available on the system and to make optimal use of available resources. Application requirements, typically, determine QoS requirements and as the resources available to a system vary, user-specified QoS requirements will have to be diligently mapped to available resources on the system. This calls for the prioritization of QoS measures (first from application viewpoint followed by available system resources viewpoint) so that the right decision is made by the runtime optimizer as per the QoS requirements of a query. Rather than specifying priority for queries explicitly, the priority of a query in MavStream is inferred from the QoS specifications.

4.2 Specification of QoS Measures

QoS is specified in Aurora using delay based, drop based and value based graphs [9]. The graphs denote the percentage utility of results for different values of delay in results or percentage of tuples delivered or the values produced. The QoS graphs are approximated as

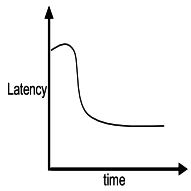

Figure 2: QoS Graph

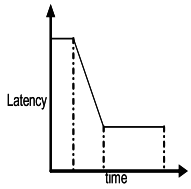

Figure 3: QoS Graph: Piecewise Approximation

piecewise linear function as it helps us to model complex functions as shown in Fig.2. In MavStream each QoS measure of interest is specified using a two dimensional graph. The QoS graphs are approximated as piecewise linear functions and contain the absolute values of the QoS measures as shown in Fig.3. The x values of a QoS graph represent relative time from the start of a query. The y values specify the expected value of QoS parameter for a time point. For tuple latency, memory utilization, throughput y values specified are time, size and tuples/sec, respectively, in their appropriate units. As QoS measures are approximated as piecewise linear functions, for each interval in a piecewise function only the start and end *(x,y)* values need to be specified. The usage of two dimensional graphs and piecewise approximation provides the flexibility to specify exact required values or relaxed values for all QoS measures.

The expected QoS values for any time period inside an interval of the piecewise function can be computed using the slope and boundary values of that interval. For a continuous query, it is infeasible to provide the QoS values for the entire lifetime of query. Hence we assume that the user provides few intervals that can be of any length. The number of intervals specified is presumed to be at-least one. The expected QoS values for time periods between two intervals is extrapolated from the border values of the preceding and succeeding intervals. For time periods that lie outside all the intervals provided, QoS value of the end time of the last interval specified or the beginning time of the first interval specified is extrapolated to obtain the expected value. This allows the runtime optimizer to have an expected value for comparison at any point in the lifetime of a query.

4.3 Priority of QoS Measures

Each QoS measure is presumed to have priority associated with it. The priority of a QoS measure determines what action should be taken when it is violated. Also each level of priority carries a weight that will be used while selecting a scheduling strategy. As the runtime optimizer chooses scheduling strategies for a query, we have categorized the QoS measures to fall into one of the three classes of priority:

1. Must Satisfy: This is a critical QoS measure for the query/application. Internally, this priority class has the highest weight associated with it. If QoS measures in this level are violated for any query, runtime optimizer tries to find a better scheduling strategy and if no better scheduling strategy is available, it activates load shedders for that query.

2. Best Effort: This class has medium weight. The runtime optimizer does not invoke load shedders for this priority class. This class of priority can be used for applications that do not tolerate errors in results (because of which load shedding is not used). The scheduling strategy with the highest score is chosen for the QoS measures that are violated.

3. Don't Care: This class has the lowest weight. The actions taken by the runtime optimizer are similar to the Best Effort class except that when more than one better strategy is available any one of the scheduling strategies which has a higher score than the current is chosen.

The algorithm used by the runtime optimizer is general and can be extended to handle any number of priority classes. The actual weights for each priority class can be specified and the values of weights normalized to the number of priority classes is used while selecting scheduling strategy for any query.

4.4 Runtime Optimizer

The runtime optimizer is responsible for monitoring QoS measures of a query, make decisions to alter the scheduling strategy of a query and invoke load shedders to drop tuples when it cannot meet the QoS requirements. The parameters involved in deciding the scheduling strategy of a continuous query are: extant of violation of QoS measures and the weights of priority class to which they belong. The performance of QoS measures of a query depends predominantly on the scheduling strategy chosen for that query and arrival rate of input streams. If the arrival rate of input stream exceeds the processing capacity, tuple latency and memory utilization are bound to increase. The processing capacity of any system is fixed and can be computed by monitoring query characteristics such as selectivity and system characteristics such as memory release capacity and operator processing capacity. The runtime optimizer therefore can carry out decisions based on the arrival rate of streams. As the input rates of streams are bursty, any change in the arrival rates of stream can potentially trigger a change in scheduling strategy of queries depending on that stream. Such an approach would also be ignorant of the actual QoS requirements of the query and may end up taking decisions to change scheduling strategy when it may not be necessary. Hence we utilize the feedback obtained by monitoring actual QoS measures and a static table (termed the decision table).

4.5 Design Alternative

The performance on QoS measures of a query depends predominantly on the scheduling strategy chosen for that query and arrival rate of input streams. If the arrival rate of input stream exceeds the processing capacity, tuple latency and memory utilization are bound to increase. The processing capacity of any system is fixed and can be computed by monitoring query characteristics such as selectivity and system characteristics such as memory release capacity and operator processing capacity. The runtime optimizer therefore can carry out decisions based on the arrival rate of streams. As the input rates of streams are bursty, any change in the arrival rates of stream can potentially trigger a change in scheduling strategy of queries depending on that stream. Such an approach would not take into account the **actual** QoS requirements of the query and may end up taking decisions to change scheduling strategy when it may not be necessary. Hence, we have used the feedback mechanism which measures the actual QoS values and takes appropriate action to rectify undesirable situations. This approach uses techniques for choosing strategies using the feedback obtained by monitoring actual QoS measures and a static table called decision table.

4.6 Decision Table

Research in DSMS has proposed many scheduling strategies each with its own characteristics to deal with different QoS measures. For example Chain scheduling [4] is an optimal strategy to minimize the memory requirement while Path Capacity Scheduling [16] is an optimal strategy for tuple latency. The decision table encompasses and represents rank information about the relative performance of strategies for various QoS measures. Each row in the table holds a relative rank of different strategies for a particular QoS measure. The information about the rank can be easily obtained by studying the performance characteristics of each strategy for the measure being considered. The runtime optimizer uses the static infor-

QoS	Round Robin	PCS	Segment	Simplified Segment
Tuple Latency	2	4	1	3
Memory Utilization	2	1	4	3
Throughput	2	4	1	3

Table 1: **Decision Table**

mation provided in the decision table along with some heuristics to choose a scheduling strategy for a query violating its QoS measures. An example decision table is shown in table 1. The motivation behind using ranks for scheduling strategies is that by knowing the relative performance of scheduling strategies for various QoS measures better results can be obtained by choosing scheduling strategies that favor measures that are violated. A static decision table provides a low runtime overhead for deciding the scheduling strategy of a query. For a continuous query the overhead of decision making process will be insignificant compared to gains achieved by changing the strategy. The alternative scoring policy for decision table considers using binary values in the table where a value of one represents a favorable strategy for the QoS measure and zero represents a non-favorable strategy. The runtime optimizer will then choose a strategy that is favorable for majority of QoS measures. This alternative can lead to multiple strategies getting same scores. Also, it does not provide any distinction between the performance of strategies for any QoS measure.

4.7 System Monitor

The system monitor continuously (over intervals determined by the runtime optimizer) monitors the output of a query for QoS measures of interest. The monitored QoS measures are compared against expected values obtained from the QoS input graph. The runtime optimizer keeps track of the QoS measures that are being violated and the percentage by which the measures fall short or exceed the expected values. Based on the QoS measures violated and their priority class a score is computed for each scheduling strategy available in the system using the ranks provided in the decision table. The ranks used in the decision table are normalized to the number of scheduling strategies when scores for strategies are computed. The weight of the priority class to which a QoS measures belongs is also considered when computing the score of a strategy for a particular QoS measures as shown in (1). The total score for a strategy is computed by summing up scores obtained from equation 1 for each of the QoS measures that are violated. If any of the scheduling strategies is determined to have a higher score for the violating measures than the current scheduling strategy, runtime optimizer chosen one among them and initiates action to change the scheduling strategy.

$$Strategy\ Score = Priority\ Class\ Weight \times \frac{Score\ in\ Decision\ Table}{Number\ of\ Strategies} \quad (1)$$

Since there is an overhead associated with changing a query from one strategy to another, the algorithm used by the runtime optimizer tries to strike a balance between the number of times a strategy is switched and the overhead incurred. If the strategy is re-computed often and changed, the overhead will be high. On the other hand, if the strategy is not changed for a long period of time, the overhead will be low but if a QoS measure is being violated, it will continue to violate it for a longer period. Also, when a strategy is changed, its effect becomes visible only after a period of time (as the operators have to be scheduled sufficient number of times to make a difference in the QoS measure value). This necessitates the runtime optimizer to consider the time it takes to effect changes to QoS measures as a result of switching. The algorithm also involves some of the policies to deal with how decisions will be taken when multiple measures are violated.

4.8 Choosing the Best Strategy

We illustrate the possible transitions and actions taken by the runtime optimizer. The strategy chosen uses the scores given in table 1. The numbers *1, 0.5, 0.01* denote the weights used for Must Satisfy, Best Effort and Don't Care priority classes respectively. A continuous query may have multiple QoS measures associated with it. At any instant of time either all measures are violated or satisfied or only some are violated or satisfied. As violated measures can fall into the same priority class or different priority classes the following two scenarios needs to be handled: (i)Violated measures belong to same priority class and (ii) Violated measures belong to different priority classes

Violated Measures Belong to the Same Priority Class: The actions taken when all measures are violated depends on the priority class of QoS measures. When only some of the measures are violated, the runtime optimizer tries to find a better strategy for the violating measures. As shown in table 2, when only memory utilization is violated Segment strategy gets the highest score (of 1) and is, hence, chosen by the runtime optimizer. The scores obtained for various strategies when multiple measures are violated are shown in table 3. Using the scores in table 3 PCS or SS can be chosen when all QoS measures are violated. If no better strategies are found after choosing this strategy, the decision maker starts activating load shedders if the measures belong to *must satisfy* class. If the measures belong to *best effort* or *don't care* priority class the runtime optimizer takes no further action and continues monitoring.

Violated Measures Belong to Different Prior-

Memory Utilization Violated	Scores
Round Robin	0.5
PCS	0 .25
Segment	1
Simplified Segment	0.66

Table 2: **Single QoS Measure Violated**

All Measures Violated	Scores
Round Robin	1*(2/4) + 1*(2/4) + 1*(2/4 = 1.5
PCS	1*(4/4) + 1*(1/4) + 1*(4/4) = 2 .25
Segment	1*(1/4) + 1*(4/4) + 1*(1/4) = 1.5
Simplified Segment	1*(3/4) + 1*(3/4) + 1*(3/4) = 2.25

Table 3: **Multiple QoS Measures Violated**

ity Classes: The runtime optimizer takes actions for lower priority measures only if higher priority measures are satisfied. Therefore the runtime optimizer considers QoS measures based on their priority and alterations to strategies for lower priority measures are done only when the higher priority measures are satisfied. Additionally if higher priority measures are not satisfied, we would like to first satisfy the critical intent of the user first and then satisfy the others as closely as possible.

$$Reduction\ Percentage = \frac{Expected\ Value - Observed\ Value}{Expected\ Value} \tag{2}$$

$$Reduced\ weight\ =\ Initial\ Weight\ - \\ (Reduction\ Percentage\ *\ Weight\ Range)$$

The priority-wise decision making scheme disallows switching for lower priority measures until higher priority measures are satisfied. But when strategies are chosen for lower priority measures, there arises a possibility that the selection made is poor for the higher priority measures. This can lead to higher priority measures getting violated. For example if tuple latency belongs to *must satisfy* class and memory utilization belongs to *best effort* class, runtime optimizer will choose Segment scheduling when tuple latency is satisfied and memory utilization is violated. This can lead to degradation of tuple latency. To prevent these situations, higher priority measures are also taken into account when decisions are taken for lower priority measures. The disadvantages of taking this approach is that best strategy for lower priority measures may never be chosen. Because of the nature of priorities, this approach is much better than choosing strategies that are unfavorable for higher priority QoS measures.

Table 4 depicts an example where QoS measures fall into different priority classes. As mentioned above memory utilization will be considered only after satisfying tuple latency. For example, if the expected value

Measures	Weights
Tuple Latency	1
Memory Utilization	0.5
Throughput	0.01

Table 4: **QoS Measures and Weights**

Latency satisfied and Memory violated	Scores
Round Robin	0.75*(2/4) + 0.5*(2/4) = 0.625
PCS	0.75*(4/4) + 0.5*(1/4) = 0.875
Segment	0.75*(1/4) + 0.5*(4/4) = 0.6875
Simplified Segment	0.75*(3/4) + 0.5*(3/4) = 0.9375

Table 5: **Measures Different Priority Classes**

for tuple latency is *2 seconds* and the observed value is *1 second* the reduction percentage for the weight will be *0.5* as per equation 3 shown above. The reduction percentage is multiplied with the range of reduction to obtain the reduced weight of *0.75*. This reduce weight is used to compute the scores for strategies as shown in table 5.

4.9 Impact of Strategy Switching

The runtime optimizer, after selecting a strategy for a query based on monitored measures, changes the scheduling strategy of that query by removing schedulable object (operators, paths, and segments in MavStream system) from ready queue of current scheduler and placing then into ready queue of selected scheduler; the new scheduler starts scheduling them. To avoid unnecessary switches and to ensure that best strategy is chosen we introduce the *lookahead factor*. *Lookahead factor* specifies the period of time ahead of the current time which the runtime optimizer uses to obtain the expected values. By comparing the measured values to expected values of a future time runtime optimizer ensures that it is ready to meet QoS requirements that will be encountered. The usage of *lookahead factor* avoids unwanted switches and makes sure that the right strategy is chosen thereby curtailing effect of switching delays. *Lookahead factor* should be a value greater than the time required to schedule all operators at least once and less than the monitoring interval of the query.

4.10 Cycling Through Strategies

As a consequence of the bursty arrival rate of streams and conflicting requirements of QoS measures, there is a possibility that the runtime optimizer will cycle through the same sequence of strategies. This cycling through same sequence of strategies can occur for two or more strategies. For example the runtime optimizer might choose Path Capacity and Segment strategies alternatively for a query to satisfy latency and memory requirements if both measures are of same priority. This may lead to a lot of unnecessary switches

that needs to be prevented.

The runtime optimizer handles such situation by remembering the decisions taken. Whenever runtime optimizer decides to change strategy it keeps track of the QoS measures that are violated and satisfied. Before making a change in scheduling strategy runtime optimizer verifies whether it is changing to a strategy that was utilized before the current strategy. If it is, runtime optimizer compares the measures that were violated previously to measures that are satisfied currently and vice versa. If they turn out to be the same, runtime optimizer assumes it as an indication of a cycle. To avoid cycling, the runtime optimizer tries to find a new strategy by taking into consideration previously violated and current violating measures. This method, though prevents cycling, can sometimes prevent a genuine change to an older strategy. But is much better than cycling through strategies introducing unacceptable overhead.

4.11 Overhead

The addition of a runtime optimizer results in some overhead in the form of monitoring, decision making, and switching strategies. The monitoring overhead includes the computation of QoS measures. The overhead for decision maker consists of computing expected measures from the QoS graph, comparing expected values to monitored values and the decision making process itself. Since the number of QoS measures and priority classes are fixed, the time taken for making decisions can be considered as small and constant. The number of times a query changes its scheduling strategy will be, in the worst case, the number of times the query output is monitored. Hence, the overhead is directly proportional to the frequency of monitoring which can be reduced by (i)Minimizing the frequency of monitoring and (ii) Minimizing the number of strategy switches.

4.12 Decision Maker

The algorithm for the Decision Maker is shown in Algorithm 1. For each query, the details about the current strategy, previous strategy and QoS graphs are tracked by the runtime optimizer. The system monitor provides the monitored values. The decision maker considers each priority class and finds violating measures for the current priority class using the *lookahead factor*. The decision maker chooses the best strategy for violating measures taking into consideration the reduced weights of satisfied measures. Decision Maker also conducts some checks to ensure that a query does not cycle through strategies. The sequence of actions for the Decision Maker is shown in Fig. 4

4.13 System Monitor

System Monitor monitors QoS measures of queries and arrival rates of input streams. The System Moni-

Algorithm 1: Runtime Optimizer Algorithm

INPUT: current time, monitored QoS values
OUTPUT: next time to monitor

```
1  HigherPriorityClassSatisfied ← true;
2  foreach PriorityClass do
3      if HigherPriorityClassSatisfied == true
       then
4      foreach QoSmeasure in
       currentPriorityClass do
5          Compare monitored and expected values
           using lookaheadfactor;
6          if violated then add to
           violatingmeasureslist;
7          else
8              Compute reduced weights using
               current expected values;
9              add to satisfiedmeasureslist;
10         end
11     end
12     if violatingmeasureslist! = NULL then
13     HigherPriorityClassSatisfied ← false;
14     newstrategy ←get strategy using
       violatingmeasureslist ,
       satisfiedmeasureslist;
15     if newstrategy == previousstrategy then
16     result ←check with previous strategy details
       for preventing looping;
17     if result == false then
18     newstrategy ← get strategy using
       previousviolatedmeasureslist
       ,violatingmeasureslist;
19     if newstrategy! = currentstrategy then
20     previousstrategy ← currentstrategy;
21     currentstrategy ← newstrategy;
22     Record current strategy details and switch to
       new strategy;
23 end
24 return get next time to monitor ;
```

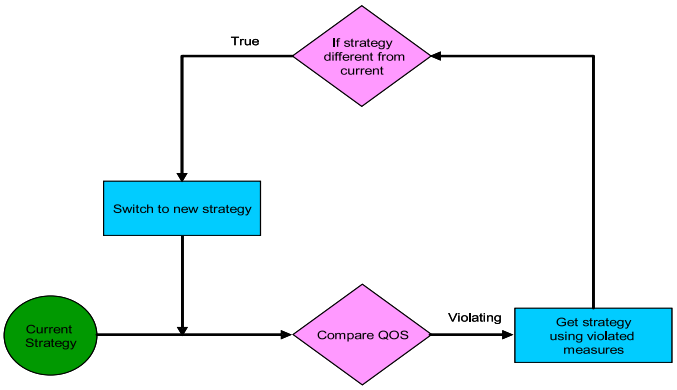

Figure 4: Runtime Optimizer Actions

tor cycles through all the queries that need to be monitored and determines if it is time to monitor a particular query. If System Monitor determines that a query needs to be monitored it obtains the QoS values and provides it to the Decision Maker. The Decision Maker compares QoS values and takes the appropriate actions and returns the time the query needs to be monitored next. The System Monitor updates the next time to schedule for a query and also keeps track of the earliest time for monitoring a query among all that needs to be monitored. This is used to determine the amount of time the monitor should sleep. The algorithm for runtime optimizer is described in Algorithm 2.

Algorithm 2: System Monitor Algorithm

```
1  while true do
2      if no queries to monitor then
3      wait;
4      else
5          foreach query to be monitored do
6              if currenttime == timetomonitor
               then get currentQoSvalues;
7              provide currentQoSvalues to Decision
               Maker;
8              get next timetomonitor;
9              if next timetomonitor <
               minnexttimetomonitor then
               minnexttimetomonitor =
               nexttimetomonitor;
10         end
11         wait
           minnexttimetomonitor − currenttime;
12     end
13 end
```

4.14 Master Scheduler

The runtime optimizer recommends a different scheduling strategy (than the current one) for a query based on the measured QoS values. Changes to a scheduling strategy of a query requires that the system should have multiple active schedulers waiting to accept new queries. At any time instant a particular scheduling strategy may or may not have objects in its ready queue for scheduling. Therefore, the runtime optimizer must notify the appropriate scheduler when it decides to place a query into a new scheduling strategy. Since potentially all the schedulers can be active, each of them competes for CPU cycles. As the scheduling mechanism of the operating system is unaware of the availability of operators in ready queues, we need a mechanism native to MavStream to control the various schedulers. Without a mechanism for controlling schedulers it is possible that one of the schedulers always has some operators ready to be scheduled.

The Master Scheduler is similar to the two level scheduling proposed in Aurora. However, in MavStream, the first level selects the scheduler and second level schedules the operators. The model followed by Master Scheduler is the Master/Slave model similar to the way schedulers control the various operators. Schedulers in MavStream are modeled as independent threads. Master scheduler runs as a separate thread allocating time for each scheduler to execute. The master scheduler follows a fair scheduling scheme by using a weighted round robin fashion through all the schedulers. The time allocated for each scheduler is determined by the number of operators in the ready queue of each scheduler. Since each operators gets a fixed quantum of time for execution, the master scheduler allocates enough time for all the operators in the ready queue to execute. The master scheduler does not get involved in the way operators get scheduled by each scheduler. Further each scheduler notifies the master scheduler if it finishes processing before the allocated time quantum. In case the scheduler has not completed processing the operators it is preempted by the master after the operator or construct being scheduled currently completes execution. The algorithm for master scheduler is shown in Algorithm 3.

5 EXPERIMENTAL EVALUATION

Effect of Runtime Optimizer on a Single QoS Measure: The query used in this experiment consisted of eight operators including two *Hash Join* operators and three input streams. The window used for the *Hash Join* was a tuple based window of five hundred tuples. The QoS measure considered was tuple latency and a single interval was specified with start and end values of 1 second. By giving a single interval with the same values for the start and end of an interval, the given value is used through out the lifetime of a query. The priority was set to Best Effort class. The three scheduling strategies considered were PCS, Segment and SS. The mean input rates for the

Algorithm 3: Master Scheduler Algorithm

```
1  while true do
2      process strategy change requests
3      if all schedulerqueues empty then
4      wait;  else
5          foreach scheduler do
6              if readyqueue != empty then  resume
               scheduler; wait operator timequantum
               * number of operators in readyqueue
7              ; suspend scheduler;
8          end
9      end
10 end
```

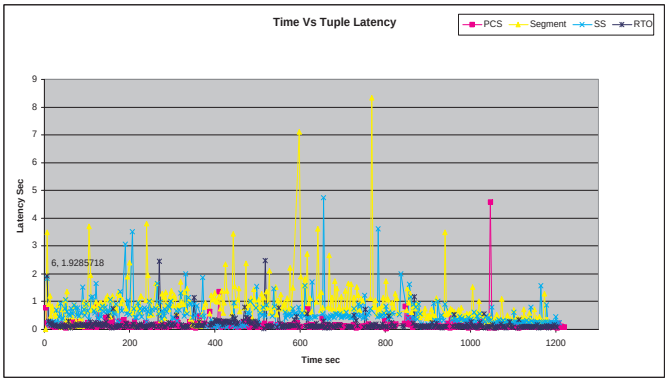

Figure 5: **Latency: Single QoS Measure**

poisson distribution was set to 2000, 1800 and 2200 tuples/sec. Each input stream consisted of two million tuples. Experiments were run by fixing each strategy without the runtime optimizer and an experiment was also run using the runtime optimizer with a starting strategy different from the one expected to see whether the decision maker will take the right decision. These experiments are plotted on the same graph to compare their effect of the QoS measure. Among the three strategies, as PCS provides the best performance for tuple latency, the runtime optimizer should choose PCS as the scheduling strategy and the latency of the query should be nearly equal to that of PCS. The initial strategy of the query was chosen to be Segment, as it provides the worst tuple latency. The monitoring interval for runtime optimizer and all other strategies were fixed to three seconds for comparison. When a single QoS measure is provided, the runtime optimizer chooses the best strategy for the QoS measure. The runtime optimizer cannot perform better than the best scheduling strategy for a QoS measure when load shedding is not allowed. As shown in Fig.5 the tuple latency is high as the initial strategy given is Segment. The Segment strategy schedules the operators that lie in the segment with the highest memory release capacity first, hence it does not produce any output initially. When output is produced the runtime optimizer determines that tuple latency is higher than the expected value and therefore changes the scheduling strategy of query to PCS. The runtime optimizer does not make any further changes as it does not find any better strategy to improve tuple latency. Fig.6 shows the latency of the query for a smaller period of time over the lifetime of query. From Fig.6 it can be observed that the strategy chosen by runtime optimizer (PCS) provides tuple latency equivalent to the best strategy (which is PCS). This experiment shows that the runtime optimizer is able to provide optimal performance for tuple latency in-spite of starting with an adverse strategy and monitoring overhead.

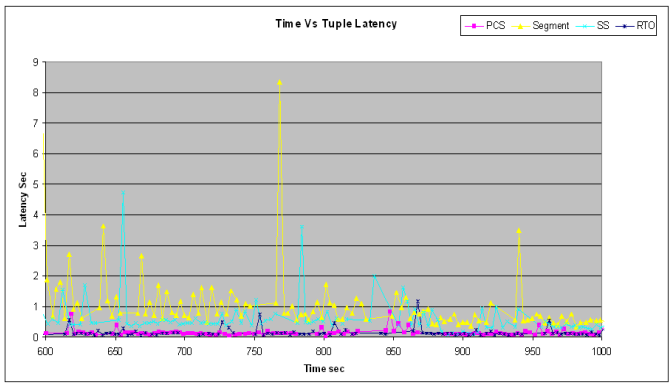

Figure 6: **Latency: Single QoS Measure**

Effect of Runtime Optimizer on Multiple QoS Measures: The query used in this experiment consisted of eight operators which includes two *Hash Join* operators and three input streams. The window used for the *Hash Join* was a tuple based window of five hundred tuples. The QoS measures considered were tuple latency and memory utilization. The three scheduling strategies considered were PCS, Segment and SS.

QoS Measures With Different Priority: For this experiment the QoS measures were given different priorities. Tuple latency belonged to the *Must Satisfy* class and a single interval was specified with a constant value of 500 ms. Memory utilization belonged to the *Don't Care* class and the expected values were 10MB for the first 500 seconds and 1MB for the remaining time. The mean input rates for the poisson distribution was set to 800, 950 and 500 tuples/sec. Each input stream consisted of two million tuples and the mean rates for the poisson distribution was doubled at different points in time to simulate bursty nature of input. As tuple latency has higher weight than memory utilization, runtime optimizer chooses PCS when it is violated as shown in Fig.7. Due to the low weight associated with the Don't Care priority class the specification of memory utilization does not make any difference in the scores computed using the decision table.

QoS Measures With Same Priority:

The QoS intervals specified for this experiment was

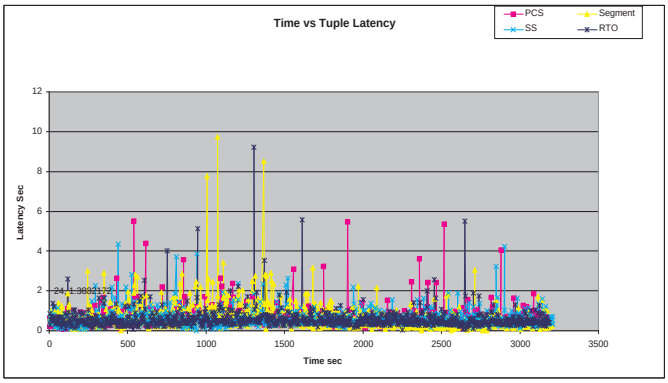

Figure 7: **Latency: Measures With Different Priority**

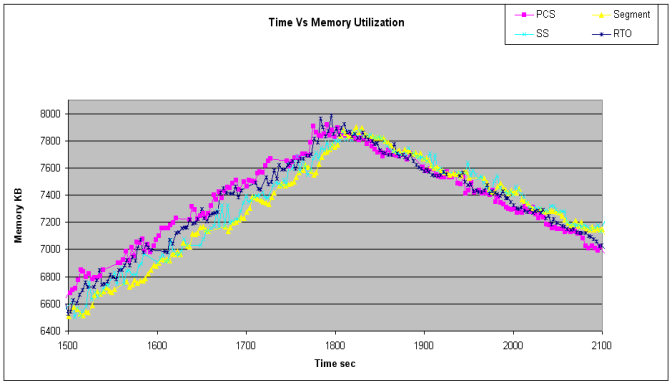

Figure 8: **Memory Utilization: Measures With Different Priority**

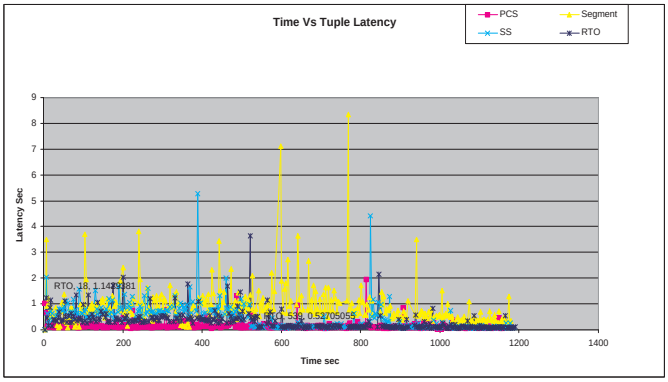

Figure 9: **Latency: Measures With Same Priority**

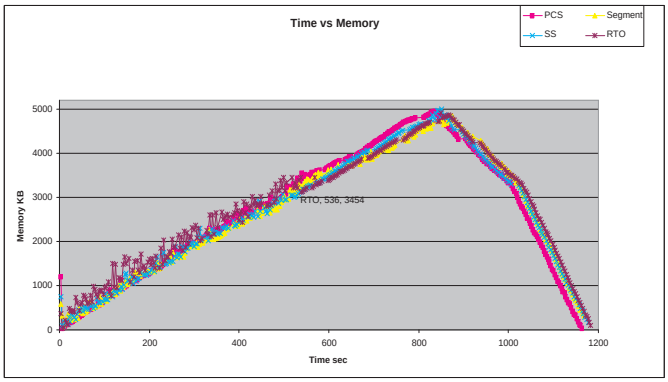

Figure 10: **Memory Utilization: Measures With Same Priority**

same as that of the previous experiment. Both QoS measures belonged to the Must Satisfy class. The mean input rates for the poisson distribution was set to 2000,1800 and 2200 tuples /sec. The initial strategy chosen was Segment as it provides the worst tuple latency. The memory requirement was kept high for an initial time period of 500 seconds. As shown in Fig.9 the memory requirement is satisfied initially and hence the runtime optimizer chooses PCS to improve latency. Later when the time period reaches the second interval specified for memory utilization memory requirement gets violated and the runtime optimizer chooses SS when both tuple latency and memory utilization are violated and Segment if only memory utilization is violated. Since both QoS measures of interest have the same priority the runtime optimizer favors measures that are being violated and chooses the appropriate strategy. As shown in Fig.9 and Fig.10 the tuple latency and memory utilization provided initially by the runtime optimizer is near to that provided by PCS. When memory utilization starts getting violated at time point 539, runtime optimizer chooses the Segment where the memory utilized decreases noticeably as shown in Fig.10. As the tuple latency provided is within the QoS limits the runtime optimizer continues execution in Segment strategy. Hence by choosing strategy to improve the performance of violating QoS measures, the runtime optimizer is able to provide better performance for both measures of the query than by executing the query in a single scheduling strategy.

6 CONCLUSION AND FUTURE WORK

In this paper we have presented the issues involved in the design, implementation, and evaluation of a runtime optimizer. We have introduced a decision table that stores information about performance of various scheduling strategies. The runtime optimizer uses this decision table to select the appropriate strategy. Heuristics to reduce the overhead by reducing the num-

ber of switches and a way to avoid cycling between strategies were also developed. Extensive experimental validation indicates the correctness of the RO under disparate input characteristics.

7 ACKNOWLEDGEMENT

The authors would like to thank Aditya Telang for his help in formatting the paper.

References

[1] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Çetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stanley B. Zdonik. The design of the borealis stream processing engine. In *CIDR*, pages 277–289, 2005.

[2] Daniel J. Abadi, Donald Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stanley B. Zdonik. Aurora: a new model and architecture for data stream management. *VLDB J.*, 12(2):120–139, 2003.

[3] Arvind Arasu and Jennifer Widom. Resource sharing in continuous sliding-window aggregates. In *VLDB*, pages 336–347, 2004.

[4] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Dilys Thomas. Operator scheduling in data stream systems. *VLDB J.*, 13(4):333–353, 2004.

[5] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *PODS*, pages 1–16, 2002.

[6] Brian Babcock, Mayur Datar, and Rajeev Motwani. Load shedding for aggregation queries over data streams. In *ICDE*, pages 350–361, 2004.

[7] Hari Balakrishnan, Magdalena Balazinska, Donald Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Eduardo F. Galvez, Jon Salz, Michael Stonebraker, Nesime Tatbul, Richard Tibbetts, and Stanley B. Zdonik. Retrospective on aurora. *VLDB J.*, 13(4):370–383, 2004.

[8] Philippe Bonnet, Johannes Gehrke, and Praveen Seshadri. Towards sensor database systems. In *Mobile Data Management*, pages 3–14, 2001.

[9] Donald Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, and Stanley B. Zdonik. Monitoring streams - a new class of data management applications. In *VLDB*, pages 215–226, 2002.

[10] Donald Carney, Ugur Çetintemel, Alex Rasin, Stanley B. Zdonik, Mitch Cherniack, and Michael Stonebraker. Operator scheduling in a data stream manager. In *VLDB*, pages 838–849, 2003.

[11] Sharma Chakravarthy and Vamshi Pajjuri. Scheduling strategies and their evaluation in a data stream management system. In *BNCOD*, pages 220–231, 2006.

[12] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel Madden, Frederick Reiss, and Mehul A. Shah. Telegraphcq: Continuous dataflow processing. In *SIGMOD Conference*, page 668, 2003.

[13] Sirish Chandrasekaran and Michael J. Franklin. Streaming queries over streaming data. In *VLDB*, pages 203–214, 2002.

[14] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. Niagaracq: A scalable continuous query system for internet databases. In *SIGMOD Conference*, pages 379–390, 2000.

[15] Altaf Gilani, Satyajeet Sonune, Balakumar Kendai, and Sharma Chakravarthy. The anatomy of a stream processing system. In *BNCOD*, pages 232–239, 2006.

[16] Qingchun Jiang and Sharma Chakravarthy. Scheduling strategies for processing continuous queries over streams. In *BNCOD*, pages 16–30, 2004.

[17] Samuel Madden and Michael J. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *ICDE*, pages 555–566, 2002.

[18] Vamshi Pajjuri. Design and implementation of scheduling strategies and their evaluation in mavstream. Master's thesis, University of Texas at Arlington, Arlington, 2004.

[19] Utkarsh Srivastava and Jennifer Widom. Memory-limited execution of windowed stream joins. In *VLDB*, pages 324–335, 2004.

[20] Nesime Tatbul, Ugur Çetintemel, Stanley B. Zdonik, Mitch Cherniack, and Michael Stonebraker. Load shedding in a data stream manager. In *VLDB*, pages 309–320, 2003.