RUNTIME OPTIMIZATION AND LOAD SHEDDING IN MAVSTREAM: DESIGN AND IMPLEMENTATION

by

BALAKUMAR K. KENDAI

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE AND ENGINEERING

THE UNIVERSITY OF TEXAS AT ARLINGTON

December 2006

Copyright © by BALAKUMAR K. KENDAI 2006

All Rights Reserved

To my family and friends.

ACKNOWLEDGEMENTS

I would like to thank my supervising professor Dr. Sharma Chakravarthy for providing me an opportunity to work on this project and constantly motivating and encouraging me. I also thank Dr. Leonidas Fegaras and Dr. Gautam Das for taking time to serve in my defense committee.

I would also like to extend my appreciation to all people who were involved in this project for their suggestions and help through the course of my work. I also express my gratitude towards all alumni and present members of ITLAB for their support.

This work was also supported, in part, by the National Science Foundation (NSF) (grants IIS-0326505, IIS 0534611 and EIA-0216500) and the Computer Science and Engineering Department at UT Arlington.

Last but not the least, I am indebted to all my family members for their encouragement and love.

November 20, 2006

ABSTRACT

RUNTIME OPTIMIZATION AND LOAD SHEDDING IN MAVSTREAM: DESIGN AND IMPLEMENTATION

Publication No. _____

BALAKUMAR K. KENDAI, M.S.

The University of Texas at Arlington, 2006

Supervising Professor: Sharma Chakravarthy

In data stream processing systems Quality of Service (or QoS) is extremely important. The system should try its best to meet the QoS requirements specified by a user. On account of this difference, unlike in a database management system, a query 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, data stream processing systems 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 effect the processing of the queries to meet expected QoS requirements.

This thesis addresses runtime optimization issues for MavStream, a data stream management system (DSMS) being developed at UT Arlington. We have developed a runtime optimizer for matching the output (latency, memory, and throughput) of a con-

tinuous query (CQ) with its QoS requirements. Calibrations are made by monitoring the output and comparing it with the expected output characteristics. 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. A master scheduler has been implemented to enable changing scheduling strategies in the middle of continuous query processing and optimize each query individually (that is, different queries can be executed using different schedulers).

In addition, to cope with situations where the arrival rates of input streams exceed the processing capacity of the system, we have incorporated load shedding component into the runtime optimizer as well. We have implemented shedders as part of the buffers to minimize the overhead for load shedding. We also choose load shedders that minimize the error introduced into the result as a result of dropping some tuples. Finally, load shedders are activated and deactivated by the runtime optimizer. Both random and semantic shedding of tuples is supported.

A large number of experiments have been conducted to test the runtime optimizer and observe the effect of different scheduling strategies and load shedding on the output of continuous queries.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS iv

ABSTRACT v

LIST OF FIGURES viii

LIST OF TABLES ix

Chapter

1. INTRODUCTION 1

    1.1 Issues in DSMSs 2

        1.1.1 Queries 2

        1.1.2 Operators 3

        1.1.3 Scheduling and Buffering 4

        1.1.4 Quality of Service (QoS) 4

    1.2 Contributions 5

    1.3 Thesis Organization 7

2. RELATED WORK 8

    2.1 NiagaraCQ 8

    2.2 Cougar 9

    2.3 Telegraph 10

    2.4 Stream 12

    2.5 Aurora 14

    2.6 Summary 16

3. MAVSTREAM ARCHITECTURE 17

    3.1 MavStream Client 17

3.2 MavStream Server 19
3.3 Input Processor 20
3.4 Instantiator 20
3.5 Scheduler 21
3.6 Feeder 23
3.7 Operators and Buffer 23
3.8 Summary 24
4. DESIGN 25
4.1 Inputs to Runtime Optimizer 25
4.1.1 Specification of QoS Measures 26
4.1.2 Priority of QoS Measures 28
4.2 Runtime Optimizer 29
4.2.1 Design Alternative 29
4.2.2 Decision Table 30
4.2.3 System Monitor 31
4.2.4 Choosing the Best Strategy 33
4.2.5 Impact of Strategy Switching 37
4.2.6 Cycling Through Strategies 38
4.2.7 Overhead 39
4.2.8 Decision Maker Algorithm 40
4.2.9 System Monitor Algorithm 41
4.3 Master Scheduler 42
4.3.1 Synchronizing the Switching Process 43
4.4 Load Shedding 44
4.4.1 Notations 44
4.4.2 Location of Shedders 45

4.4.3 When and How Much to Shed 47
4.4.4 Load Shedding by Runtime Optimizer 47
4.4.5 Where to Shed 48
4.4.6 How to Shed Load 51
4.5 Summary 54
5. CONCLUSION AND FUTURE WORK 58
REFERENCES 60
BIOGRAPHICAL STATEMENT 64

LIST OF FIGURES

Figure		Page
3.1	MavStream Architecture	18
4.1	QoS Graph	27
4.2	QoS Graph: Piecewise Approximation	27
4.3	Runtime Optimizer Actions	41
4.4	Load Shedders: Placement	46
4.5	Runtime Optimizer Flowchart	48

LIST OF TABLES

Table		Page
4.1	Decision Table	30
4.2	Single QoS Measure Violated	34
4.3	Multiple QoS Measures Violated	34
4.4	QoS Measures and Weights	36
4.5	Measures Different Priority Classes	37

CHAPTER 1

INTRODUCTION

Database management systems (DBMSs) have allowed application developers to separate the issues of management and persistence of data from the application logic. In addition to querying capabilities, DBMSs enforce the constraints set by the application on persisted data and also provide concurrency and recovery techniques. These mechanisms make sure that the database is always in a consistent state. DBMSs work on a request-response paradigm wherein the user poses a logical query and receives a response, which is always expected to be precise. DBMSs work on a transactional model and has so far satisfied scenarios where most operations on data are read.

Recently, the wide-scale deployment of sensors and handheld devices has created sources that produce data continuously. These kind of data created by sensors are called data streams [1, 2, 3]. Unlike DBMSs where the data is always stored on secondary storage, sensor and other monitoring devices produce continuous data that can be unreliable and rates unpredictable. The size of the data stream is also unbounded and can be considered as a relation with infinite tuples (not stored on a secondary device). The huge volume of data coming from these types of sources makes storing them infeasible. Examples of applications [4] that have streaming input are network monitoring, stock tickers and variable tolling in highways.

Many stream based applications require the results to be produced within A specific amount of time (termed tuple latency). These requirements, known as Quality of Service (QoS), necessitates 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 rules out the possibility of storing the data into secondary storage and processing them. 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 has called for the development of specialized applications for handling them known as Data Stream Management Systems (DSMS) or Stream Processing Systems [5, 6, 7].

1.1 Issues in DSMSs

The unbounded and unpredictable nature of data streams has brought out lots of challenges to researchers in data streams. The key difference between a DBMS and DSMS is the way the system processes the data. DBMSs follow a pull based model as opposed to push based model of DSMS. In a DBMS data is static and queries are changing, but in a DSMS generally queries are long running and data is constantly updated. The data that arrives from the stream flows through the operators of a query tree.

1.1.1 Queries

Streaming queries can be broadly classified [2, 8, 9] into:

- Predefined queries, and
- Ad-Hoc queries

Predefined queries are queries, which are available to the system before any relevant data has arrived. Ad-Hoc queries are submitted to the system when the data stream has already started. Ad-Hoc queries that refer to past information is difficult to evaluate unless the system supports storage of past information. Data streams may also be correlated with data stored in traditional databases. Hence, we cannot preclude processing stream data along with traditional data. For example, a streaming Join operator may combine streams with stored relations. Since Ad-Hoc queries are not known beforehand,

query optimization, finding common sub-expressions, etc., add complexity to the system. Queries are further classified into:

- One-Time queries or Snapshot queries: These queries are evaluated only once over a given window. Once the query is evaluated, it is removed from the system. Snapshot queries may require historical data which requires stream be stored in a secondary storage.
- Continuous queries: Continuous queries are fixed and work on the most recent data. These queries are evaluated continuously as long as there is stream data. The results are produced incrementally and continuously at the end of every new window. As most of the queries in a DSMS are continuous, data that is not required for snapshot queries can be discarded once it is processed.

1.1.2 Operators

Operators [10, 8] in a DBMS are designed to operate on a finite set of data. The unbounded nature of data does not allow blocking operators like join and aggregate in a streaming environment as they require the entire data to be present before they can give results. One of the widely adopted solution to handle this problem is to use windowing which divides the stream into smaller finite set of data. The blocking operators computes result on the smaller set of tuples that fall inside the window. Windows can be either time based, tuple based or even value based.

Time-stamping of tuples is another issue especially when operators combine input from multiple streams. Operators in a DSMS can use the time stamp that is either provided by the source of the stream or apply system generated time stamps to the tuples as they are fed to the operators. If windows are defined on source time stamps, ordering of the tuples is an important issue. Data sources for stream are often low powered sensors that need to conserve power. Data that arrive from sensors may get

dropped or may arrive out of order. Mechanisms like timeout, slack parameters [6] and k constraints [11] have been developed to handle such situations.

1.1.3 Scheduling and Buffering

As data rate of a stream is unpredictable, operators may not always find the required data. There can also be periods where data arrives at a very high rate. This necessitates buffering [10, 9] as there may not be enough capacity in a DSMS to process all incoming data without buffering. The scheduling mechanisms provided by the operating system are ill-suited to DSMS as scheduling needs to be based on the availability of data. Hence specialized scheduling strategies [12, 13, 14, 15, 16] have been proposed for operators in a DSMS. The absence of scheduling and buffering can lead to wastage of resources and loss of tuples.

Amount of memory available in a system is crucial for stream applications as most of the data required for processing is stored in the main memory to meet the real time requirements. 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 [10, 6] and scheduling strategies [14, 16] 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 [17, 18].

1.1.4 Quality of Service (QoS)

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 for a query. The three QoS measures that are mainly used in a DSMS are: tuple latency, memory

utilization and throughput. Tuple latency is specified as the difference in 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. This generally does not include the tuples stored in the internal queues (synopsis) of window-based 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. Also load shedding [19, 20] and other approximation techniques have been proposed to deal with situations when the system is unable to meet the QoS requirements.

1.2 Contributions

The unpredictable nature of stream data and QoS requirements of stream-based applications has nurtured research which has proposed various scheduling and approximation techniques. 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; any strategy that will be able to meet the QoS requirements of query will be sufficient. 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 adjust the scheduling strategy of a particular query to meet the QoS requirements during its lifetime. The overall QoS requirements

can be satisfied better, if a strategy chosen can deliver better performance for the QoS requirements that are not currently satisfied.

Scheduling strategies optimized for QoS measures such as tuple latency and memory utilization are by nature conflicting. Although strategies [14, 16] have been proposed that tries to provide a compromise for both the measures, much better results can be obtained by changing the strategies based on the actual feedback. Although research in DSMS has proposed many scheduling strategies, to the best of our knowledge, none of them have alter the scheduling strategies over the life time of a continuous query or what scheduling strategy to use for a query. This has motivated us to develop a generalized mechanism that allows to select the right scheduling strategy for a query. The approach that we have proposed in this thesis the runtime optimizer adds little overhead to the system. The runtime optimizer monitors QoS measures for each continuous query in the system and chooses a new strategy for it, if any of the QoS measures are violated. This mechanism also tries to ensure that none of the QoS measures that are currently satisfied does not get violated by switching to a new strategy. Moreover the runtime optimizer also tries to minimize the number of switches between strategies based on some heuristics.

Any DSMS is ultimately limited by the physical resources available to the system. There may arise situations where the system may not be able to meet the QoS requirements because of periods of extremely high arrival rates. As many of the stream-based applications can tolerate approximate answers, tuples can be dropped from the system in order to meet the QoS requirements. This process of gracefully dropping tuples from the system is known as load shedding. Load shedding [19, 20, 21] is a well researched problem and various groups have proposed different mechanisms. In this thesis we have proposed a load shedding mechanism that incurs the minimal overhead among all proposed load shedding mechanisms. This is achieved by making load shedding a function of the input queues of operators. The load shedding strategy proposed in this thesis is

a feedback based approach, where QoS measures are monitored and tuples are dropped either randomly or based on the semantics specified in the continuous query. The proposed mechanism also activates load shedders at specific locations which maximizes the gain in processing time while minimizing the error it introduces.

The two mechanisms mentioned above while not dependent on each other work well in conjunction to enhance the overall performance of the system. These modules are part of the runtime optimizer. The runtime optimizer monitors QoS measures of a query to choose the right strategy for meeting QoS requirements. If the runtime optimizer determines that no other better strategy exists it resorts to load shedding by activating load shedders. Once the query meets the desired QoS requirements, the runtime optimizer starts deactivating shedders. Thus, the runtime optimizer always works towards satisfying QoS measures by choosing an appropriate strategy and shedding load.

1.3 Thesis Organization

The rest of the thesis is organized as follows. In chapter 2, we discuss some of the prominent related work. In chapter 3, we briefly describe the architecture of MavStream and its various components. In chapter 4, we discuss the design details of the various mechanisms proposed in this thesis and their benefits. Chapter **??** concentrates on the implementation details of the proposed mechanisms and its analysis. In chapter **??**, we describe the various experiments conducted and analyze the results.Finally, in chapter 5 we give the conclusion and future work.

CHAPTER 2

RELATED WORK

In this chapter we present some of the work that has been carried out on stream processing systems. More emphasis is put on work that are closely related to our system.

2.1 NiagaraCQ

NiagaraCQ [7] 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. The new query is merged into an existing group whose signature matches that of the new query.

A prototype version of NiagaraCQ includes a Group Optimizer, Continuous Query Manager, Event Detector and Data Manager. There are various phases of continuous query processing, which includes continuous query installation during which the query is parsed and the query plan is fed into the group optimizer for incremental grouping. A unique name is generated for every user-defined continuous query, which the user can use to retrieve the query status or to delete the query. Queries are automatically removed from the system when they expire. Continuous query execution sends query id and relevant files to the Continuous Query Manager. The Continuous Query Manager invokes the Niagara query engine to execute the triggered queries.

In NiagaraCQ, both timer-based and change-based continuous queries can be grouped together for event detection and group execution. Incremental evaluation of continuous queries, use of both pull and push models for detecting heterogeneous data source changes, and a caching mechanism assists in making the system scalable. They have also proposed another approach called dynamic regrouping to increase the overall quality of regrouping which otherwise would have deteriorated by continuously adding and removing queries from the group statically. The Rate Based Optimization [22] aims at maximizing the throughput of the query. This technique estimates the output rates of various operators as a function of the rates of their input and uses these estimates to optimize queries.

Recent work from the group has proposed a framework [23] for defining window semantics and a window query evaluation technique based on it. The evaluation technique processes the tuples on the fly even for out of order tuples by using a feature termed WID. The proposed framework is independent of any particular operator implementation algorithm and processes the queries with one pass over the data. To reduce the space and computation of windows they have also mentioned a method called panes [24] which divides overlapping windows into disjoint panes. Panes are used to compute sub aggregate which is then rolled up to obtain the aggregate for the window.

2.2 Cougar

Cougar [25] 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 relational database system.

Sensor queries are defined as an acyclic graph of sequence and relational operators. Sensor data is considered as a combination of stored data and sensor data. Stored data

are represented as relations and sensor data are represented as time series data based on a sequence model. Long running sensor queries are supported by this system. In Cougar, signal-processing functions are represented as an Abstract Data Type (ADT) functions. Sensor ADTs are defined for sensors of the same type (e.g., temperature sensor, seismic sensor etc). Public interface to an ADT corresponds to the signal processing function supported by a type of sensor. Sensor queries are SQL like queries with a little modification that includes ADT specification in the SELECT or WHERE clause of the query.

Query processing takes place on a database front end where as the signal-processing functions are executed on the sensor nodes involved in the query. On each sensor a lightweight query execution engine is responsible for executing signal processing functions and sending data back to the font end. Query optimization is performed in a distributed manner over multiple aggregate queries and is designed to minimize the communication needs of the sensors while observing its computational limits. Several algorithms [26] have been proposed to reduce communication needs of sensors and the computational effort by using techniques such as result sharing and result encoding.

2.3 Telegraph

The Telegraph project at UC Berkeley began in early 2000 with the goal of developing an Adaptive Dataflow Architecture for supporting a wide variety of data-intensive, networked applications. Telegraph [5] 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 [27]. 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.

Telegraph does not rely upon a traditional query plan. Telegraph constructs a query plan that contains adaptive routing modules, which are able to re-optimize the plan on a continuous basis while a query is running. Eddies [28] are modules that adaptively decide how to route tuples, choosing orderings among commutative modules. Instead of determining a static optimal global query plan before execution based on the knowledge of costs of each operation, it relies on simple dynamic local decisions to get a good global strategy. Eddies have flexible prioritization scheme to process tuples from its priority queue. Lottery Scheduling is utilized to do local scheduling based on selectivity. By adding a feedback to the control path, it effectively favors the low-selectivity operators and thus corresponds to the selectivity heuristic in static query optimization. The burden of history in routing was one of the fundamental limitation of the original eddies proposal. A mechanism called STAIRS [29] has been proposed that allows the query engine to manipulate the state stored inside the operators and undo the effects of past routing decisions.

Two prototypes extending Telegraph to support shared processing over streams were developed, namely Continuously Adaptive Continuous Queries or CACQ [30] and PSoup [31]. CACQ was the first continuous query engine to exploit the adaptive query processing framework of Telegraph. The key innovation in CACQ is the modification of Eddies to execute multiple queries simultaneously. CACQ uses grouped filters to optimize selections in the shared execution of the individual queries. PSoup extends CACQ by allowing queries to access historical data and support for disconnected operation. It combines the processing of ad-hoc and continuous queries by treating data and queries in a symmetric fashion there-by allowing new queries to be applied to old data and new data to be applied to old queries. In order to support disconnected operation and to

improve data throughput and query response time, PSoup partially pre-computes and materializes results.

2.4 Stream

Stream [2] is a prototype implementation of a complete data stream management system being developed at Stanford. A modified version of SQL (termed CQL) [32] has been chosen for the query interface. 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. System memory is distributed dynamically among the synopsis, queues in query plans along with buffers handling streams coming over the network and a cache for disk-resident data.

Operators in Stream adhere to the update and compute answer model where in an operator reads data from its input queue, updates the synopsis structures and writes results to its output queues. Operators are adaptive and take care of the dynamically changing stream characteristics such as stream flow rates, and the number of concurrently running queries. Operators can produce approximate answers based on the available memory.

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. A comprehensive user interface has been developed that will facilitate the users and administrators to visually monitor the execution of continuous queries, including memory usage and accuracy of output. It also plans to provide the system administrator the capability to modify system parameters such as memory allocation and scheduling policies.

Stream uses the chain scheduling strategy [16] with the goal of minimizing the total internal queue size. The slope of an operator in Chain scheduling strategy is the ratio of the time it spends to process one tuple to the change of the tuple space. Chain scheduling strategy statically assigns priorities to operators equaling the slope of the lower-envelope segments to which the operator belongs (steepest slope corresponds to biggest operator memory release capacity and hence the highest priority). At any instant of time, of all the operators with tuples in their input queues, the one with the highest priority is chosen to be executed for the next time unit. They do not consider the tuple latency of the query which is as important as memory requirement. Chain scheduling strategy suffers from poor response times during the bursts.

Load shedding techniques [20] proposed in Stream focuses only on aggregation queries over sliding windows. The algorithm for load shedding tries to determine most effective sampling rate for each query so that error is distributed uniformly among all queries, and optimal location of load shedder that increases the throughput without violating the load equation. For Join operators a memory usage reduction [18] method has been introduced using max subset (load shedding) and sampling techniques, for a novel age based model and frequency based model. Two algorithms B-Int and L-Int have been developed for sharing resources among aggregate sliding window operators. These algorithms assume that a stream can be split into sub streams which is used to compute multiple aggregates. B-Int algorithm pre-computes aggregates for intervals such that any interval can be expressed as union of base intervals. L-Int uses an interval scheme based on certain landmarks or specific positions of the stream. L-Int is more efficient than B-Int for lookups, but its update and space costs are higher. Other approximation techniques like synopsis compression and static techniques like histogram and wavelets have also been put forward.

2.5 Aurora

Aurora [6, 4] 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. Aurora can handle a variety of characteristics of stream data, such as lost, stale or garbled tuples. Aurora has a storage management module, which takes care of storing all the required tuples for its operation. It is also responsible for queue management. The emphasis is on QoS requirements. Tuples flow through a loop-free, directed, graph of processing operations (i.e., boxes). Ultimately, output streams are presented to applications, which must be programmed to deal with the asynchronous nature of tuples in an output stream.

Input in Aurora is through a GUI. Input begins at the top of the hierarchy and makes use of the zoom capability to further assist in the design. Users can then query the system. Facilities for debugging and single stepping are also provided. Aurora has dynamic optimization policies, which change the data flow computation graph (or network) at run time to improve the performance. Optimization is based on the types of queries running in the system. It does not optimize the whole network at once. But does it in parts by considering a portion of the network at a 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. It is the QoS that determines how resources are allocated to the processing elements along the path of query operation.

Scheduler is designed to cater to the needs of a large-scale system with real time response requirements. The scheduler [15] is responsible for multiplexing the processor usage to multiple queries according to application-level performance. Aurora exploits

the benefits of non-linearity in both intra-box and inter-box tuple processing primarily through train scheduling, which attempts to queue as many tuples as possible without processing, to process complete trains at once, and to pass them to the subsequent boxes without having to go to disk. In order to reduce the scheduling and operator overhead, Aurora relies on batching during scheduling. There are two types of batching, operator batching and tuple batching. Aurora scheduler performs the following tasks: dynamic scheduling-plan construction and QoS-aware scheduling. Aurora employs a two level scheduling approach to address the execution of multiple simultaneous queries. The first level handles the scheduling of superboxes which is a set of operators, and the second level decides how to schedule a box within a superbox.

Aurora handles high load situations by dropping load using a drop operator. Load shedding [19] 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. Load coefficient, which is statically computed represents the number of processor cycles required to push a single input tuple through the network to the outputs. The next step creates a load shedding road map which consists of an ordered sequence of entries. Each entry in the road map has a drop insertion plan which guarantees that number of cycles saved is maximized without sacrificing the utility of the result. At times of overload the road map is searched for the right amount cycle savings to find a drop insertion plan. A cursor is kept pointing to which row was used last, so that the next search moves forward from the cursor if more load needs to be shed or can go backwards if load is within the headroom factor. The predicate for semantic operator is dynamically determined based on the distribution of data using histograms and values based QoS graphs. The load shedding mechanism proposed in Aurora uses static information for deciding dropping

strategies and is therefore highly scalable. The dropping mechanism proposed in Aurora is independent of the scheduling mechanism. But it makes the assumption that cycles gained by dropping tuples will utilized by scheduler effectively.

Borealis [33] is a distributed stream processing engine that inherits the core stream processing functionality from Aurora. The Borealis data model extends Aurora by supporting corrections in erroneous input by way of revision messages. The goal is to process revisions intelligently, correcting query results that have already been emitted in a manner that is consistent with the corrected data. Borealis allows dynamic query modification with the help of boxes that have special control lines in addition to their standard data input lines. Control lines carry messages with revised box parameters.

2.6 Summary

In this chapter we have discussed the various systems for stream processing. Most of the systems follow the same data flow model, where the data flows through the operators. The work described in this thesis is closely related to load shedding and optimization techniques used in Aurora.

CHAPTER 3

MAVSTREAM ARCHITECTURE

MavStream is a data stream management system being developed at the University of Texas at Arlington for processing continuous queries over data streams. MavStream is a complete system where in, a query submitted by the user is processed at the server and the output is returned back to the user. MavStream is modeled as a client-server architecture in which client accepts input from the user and sends it to the server. The server processes the input and creates server specific data structures that will be used while the continuous query is being processed. The various components of MavStream are shown in Fig. 3.1.

The MavStream server upon receiving a query from the client passes it to the input processor which transforms the input to create the query plan object. 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 [13]. 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. Following sections provide a brief overview of various modules constituting the MavStream system:

3.1 MavStream Client

The MavStream functionality has been extended to accept queries and stream definitions from a web enabled GUI for query specification. The GUI generates an ASCII file of the user input. Continuous Queries (CQs) are defined in the file, by giving the

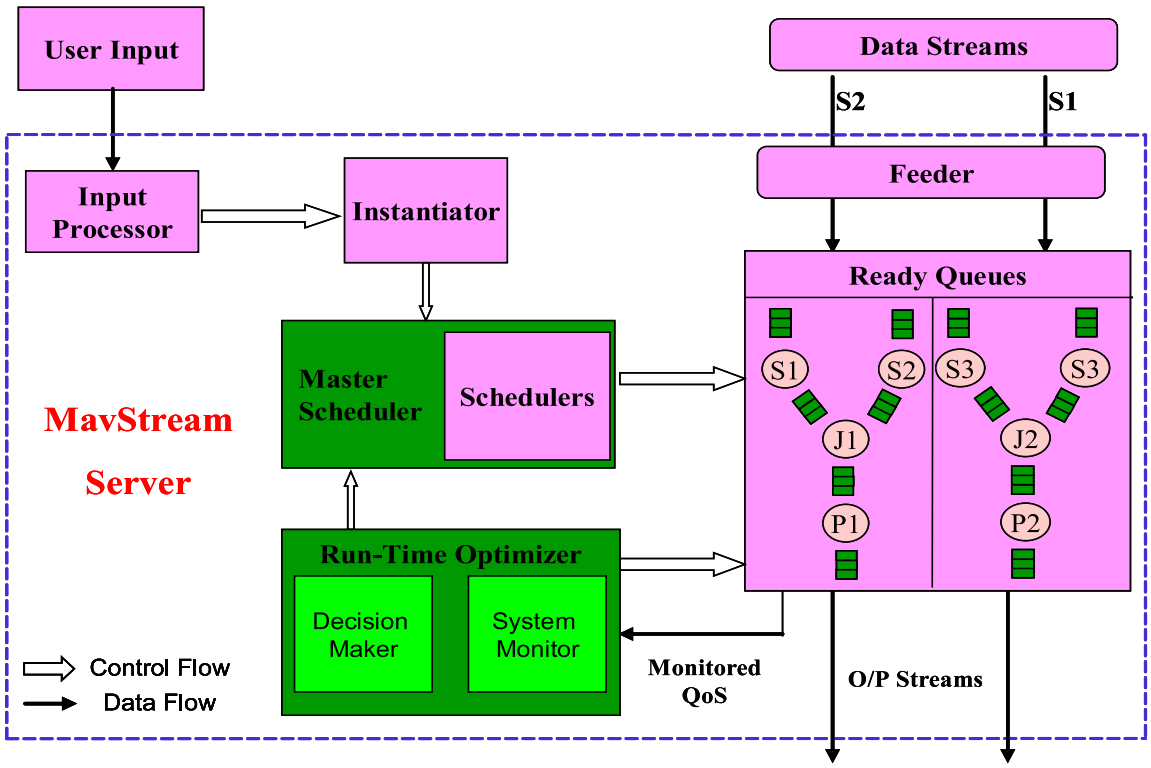

Figure 3.1 MavStream Architecture

stream operators and the information required for instantiating each stream operator. The CQ specification also requires the user to give the association between operators to form a query tree. This ASCII file can either be generated using the GUI or can be manually written as a parameter delimited file. The server takes in the text input and performs all the essential duties.

Communication between client and server is command driven and protocol oriented. Each command is associated with a protocol that defines the communication between the client and the server for the service to be successfully offered. Client needs to be aware of the protocols in order to obtain the desired service from the MavStream server.

3.2 MavStream Server

MavStream server is a TCP Server which listens on a chosen port. It is responsible for executing user requests, and producing desired output. It accepts commands and requests from a client that describes the task to be carried out. It provides integration and interaction of various modules such as input processor, instantiator, operators, buffer manager and scheduler for efficiently producing correct output. It provides details of available streams and schema definitions to clients so that they can pose relevant queries to the system. It also allows new streams to register with the system. It initializes and instantiates operators constituting a query and schedules them. It also stops a query, which in turn stops all operators associated with the query on receiving command for query termination. Some of the commands supported by the server are given below:

- Register a stream
- Receive a query plan object
- Start a query
- Send all stream information to the client
- Stop a query

In addition to the above the server also maintains mapping between user given query names and a unique id assigned to it by the system. The server on receiving a query invokes the input processor, which processes the input to create the query plan object . The server also populates the data structures required by the system monitor, interacts with the master scheduler to start scheduling queries. It also stores the query plan object of each query during its lifetime and also maintains information required for house keeping and statistics.

3.3 Input Processor

The input processor processes text input to generate a query plan object. This module extracts the information of operators and query tree from the input. Each operator definition is populated in a data structure called *Operator Data*. The *Operator Data* is wrapped in an *Operator Node* that has references to the parent and child operators. The entire query plan object is accessed using a reference of the *Operator Node* for the root operator.

The query plan object is a sequence of operator nodes where every node describes an operator completely. The operator hierarchy defines the direction of data flow starting from leaves to the root. On visiting each node input processor calls the instantiator module to instantiate the operators. The query tree is traversed in a bottom-up manner to ensure that required child operators are instantiated prior to parent operators to respect query semantics as data flows from leaves to root. A reference to the root operator is also kept in the query plan object.

Once the operators are instantiated the input processor processes the query tree again to create operator paths, segments and simplified segments. It also computes the processing capacity of paths and memory release capacity of segments and sorts them based on their respective capacities. A reference to these data structures is maintained in the query plan object. The references to these data structures is used later when the query changes the scheduling strategy. Further the module processes the QoS parameters and stores them in the appropriate data structures to be used by system monitor.

3.4 Instantiator

Instantiator has the responsibility of initializing and instantiating streaming operators and their associated buffers on accepting user queries from the client. It creates an

instance of each needed operator and initializes it on reading *Operator Node* data. It then associates input and output queues (or buffers) with desired parameters to operators for consuming and producing tuples. Every operator is an independent entity and expects predicate condition in a predefined form. Instantiator extracts the information from the operator node and converts it into the form required by each operator. It also associates a scheduler with the operator to facilitate communication for scheduling. Instantiator does not start the operator rather it does all the necessary initialization.

3.5 Scheduler

The scheduler is one of the critical components in MavStream. 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 (unbounded). On the other hand, scheduling at the query level looses flexibility of scheduling, 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. Following are the scheduling policies [13] supported by MavStream:

1. Round-Robin: In this strategy, 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 oper-

ators at leaf nodes can be given more priority as they are close to data sources. Similarly, Join operator, which is more complex and time consuming, can be 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: It uses the same segment strategy but the way segments are constructed is different 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. Master scheduler allocates time quantum to each scheduler to execute. At any instance of time only one scheduler is allowed to run by the master scheduler. Master scheduler also provides an interface to add and remove queries from the ready queue of the scheduler. Other modules request master scheduler to add or remove queries or to change the scheduling strategy of queries.

3.6 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.

3.7 Operators and Buffer

The processing requirements of stream-based applications are different from traditional applications. Operators of traditional DBMSs are designed to work on the data that is already available and cannot produce real-time response for queries over high volume, continuous, and time varying data streams. The operators of DSMSs are designed to handle long running queries producing results continuously and incrementally. Blocking operators (an operator is said to be blocking if it cannot produce output unless all the input is used) like Aggregates and Join may block forever on their input as streams are potentially unbounded. Stream operators are designed using the window concept so as to overcome the blocking nature of operators. During the life span of an operator, it can either be in ready, running, suspended or stop state. MavStream supports the following operators - split, select, project , join (hash and nested versions) , group by and various aggregate operators (sum, average, max, min, count).

The objective of buffer management is to provide a mechanism to handle the mismatch between input rates and the processing capacity by using available memory. But when we have limited main memory, there is an upper limit on the number of tuples that can be stored in the main memory. If tuples exceed this limit, they have to be either discarded or stored in secondary storage. An interface is provided to store tuples either in main memory buffers or on secondary storage (using a configuration option) and retrieve the tuple stored in secondary storage. The management of main memory buffers and secondary storage for tuples is transparent to the user. It is also transparent to the operator and is completely handled by the buffer manager. The option of storing excess tuples onto a disk is a hindrance for meeting QoS requirements like throughput and latency. Load shedding functions have been incorporated into the buffer to deal with situations of high load. The load shedderS are controlled (activated and deactivated) entirely by the runtime optimizer with out involving the operators.

3.8 Summary

In this chapter we discussed about the major modules of MavStream. Various components of server such as instantiator, scheduler, input processor and feeder were explained briefly. Interaction between these components was also covered.

CHAPTER 4

DESIGN

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, 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 chapter, we discuss the issues encountered and assumptions made while designing the runtime optimizer. In the latter part of this chapter, we illustrate the design of load shedders and techniques utilized to achieve maximum gains.

4.1 Inputs to Runtime Optimizer

In order to guarantee (or even come close to user expectation), we need to accept user expectations in some form. Once the expected qos measures are specified for a query, the runtime optimizer can make use of it to meet the requirements in the best possible way. The basic problem is to match the user requirement with the resources available on the system and to make best of the 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. For example in a system with limited memory, memory violation is likely

to happen earlier during periods of high load and in turn increase the tuple latency. 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. Also not all queries may require the services of runtime optimizer. A DSMS must therefore be capable of identifying queries that need to be optimized from those that need not be.

4.1.1 Specification of QoS Measures

QoS is specified in Aurora using delay based, drop based and value based graphs [1]. 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 piecewise linear function as it helps us to model complex functions as shown in Fig.4.1. 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.4.2. 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

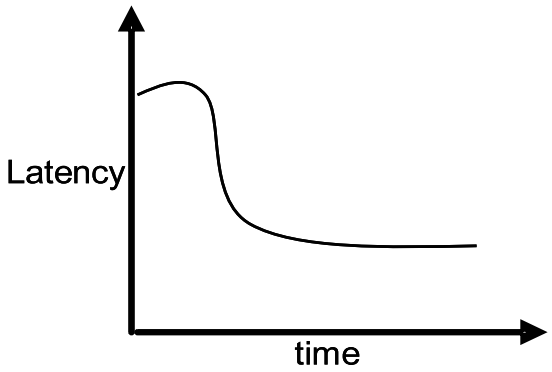

Figure 4.1 QoS Graph

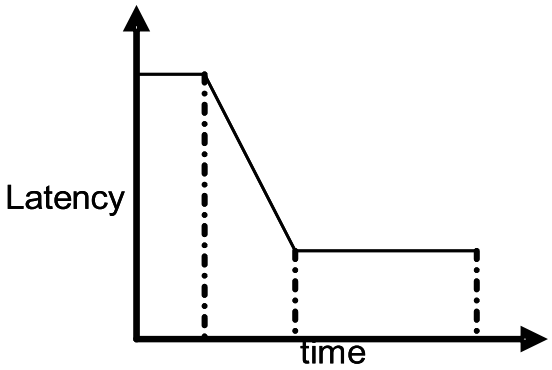

Figure 4.2 QoS Graph: Piecewise Approximation

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.

4.1.2 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 and also controls load shedders we have categorized the QoS measures to fall into one of the three classes of priority based on the possible decisions that can be taken by the runtime optimzer. The three classes of priority are Must Satisfy, Best Effort and Don't Care.

- Must Satisfy: This is a critical QoS measure for the query/application. Hence it is given this categorization. Internally, this priority class has the highest weight associated with it. The runtime optimizer evaluates the QoS measures in this class of priority first. If the QoS measures in this level are violated for any query, runtime optimizer tries to find a better strategy and if no better strategies are available it activates load shedders for that query.
- Best Effort: This class of measures have medium weights. 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 error 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. The QoS measures falling into this priority class are evaluated after evaluating the ones in Must Satisfy class.
- 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. QoS measures of this priority class are evaluated last.

Though we have limited the number of priority classes to three, 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.

4.2 Runtime Optimizer

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. In this section we explain mechanisms used for choosing scheduling strategy of a query. The parameters involved in deciding the scheduling strategy of a continuous query are extent of violation of QoS measures and the weights of priority class to which they belong. The runtime optimizer makes decisions for each query separately. As decisions for a query are taken when any of the QoS measures are likely to be violated, the decision making process itself must be of very little overhead. If the scheduling strategy being used succeeds in meeting the QoS requirements of a query, runtime optimizer does not take any actions for that query.

4.2.1 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 porocessing capacity. The runtime optimizer therefore can carry out decisions based on the arrival rate of streams. As the

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 4.1 Decision Table

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 a technique for choosing strategies using the feedback obtained by monitoring actual QoS measures and a static table called decision table.

4.2.2 Decision Table

Research in DSMS has proposed many scheduling strategies each providing a varying level of performance for different QoS measures. For example Chain scheduling [16] is an optimal strategy to minimize the memory requirement while Path Capacity Scheduling [14] 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 information provided in the decision table along with some heuristics to chose a scheduling strategy for a query violating its QoS measures. An example decision table is shown in table 4.1.

The values in the decision table hold the rank for a scheduling strategy represented in the column for the QoS measure denoted by the row. The ranks can take values based

on the number of strategies available in a system. For example in the decision table 4.1, Path Capacity Scheduling (PCS) has the highest value 4 for rank among the four strategies for tuple latency while segment has the lowest rank of one. 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 considered 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.

The usage of decision table also prevents the logic from being hard wired into the code. By making the decision making process and decision table separate the runtime optimizer becomes flexible enough to accommodate new scheduling strategies or remove existing ones with little changes to the decision making process. The addition and deletion of scheduling strategies confines the changes to be made only to the decision table. Also the decision table allows the runtime optimizer to explore different scenarios and make the right decision for any query.

4.2.3 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 (4.1). The total score for a strategy is computed by summing up scores obtained from equation 4.1 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.

$$Score\ of\ Strategy = Weight\ of\ Priority\ Class * \frac{Score\ in\ Decision\ Table}{Number\ of\ Strategies} \quad (4.1)$$

Since there is an overhead associated with changing a query from one strategy to another, the algorithm for 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 long pepriod. 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.

4.2.4 Choosing the Best Strategy

In this section we illustrate the possible transitions and actions taken by the runtime optimizer. The strategy chosen uses the scores given in table 4.1. 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 taken care of.

1. Violated measures belong to same priority class
2. Violated measures belong to different priority classes

4.2.4.1 Violated Measures Belong to the Same Priority Class

The actions taken when all measures are violated depends on the priority class of QoS measures. The runtime optimizer uses load shedding option for the *must satisfy* class only. Load shedding is always chosen as a last resort when the optimizer does not have a better strategy. When only some of the measures are violated runtime optimizer tries to find a better strategy for the violating measures. As shown in table 4.2, 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 4.3. Using the scores in table 4.3 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 as the measures belong to *must satisfy* class. If the measures belong to *best effort* or

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

Table 4.2 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 4.3 Multiple QoS Measures Violated

don't care priority class the runtime optimizer takes no further action and continues monitoring.

4.2.4.2 Violated Measures Belong to Different Priority Classes

The runtime optimizer takes actions for lower priority measures only if higher priority measures are satisfied. The initial design was to consider violating QoS measures from all classes and select a better strategy using the weights of QoS measures and scores of strategies. But this option would not be the best as this will disallow runtime optimizer from choosing the best strategy for higher priority QoS measures even if they are violated. For example, consider a scenario where memory utilization belongs to *must satisfy* class and throughput, tuple latency belong to *best effort* class for a query. The scheduling strategy chosen by runtime optimizer when all measures are violated will be Simplified Segment when considering measures of all classes together. This prevents the query from switching to Segment scheduling which provides lowest memory utilization. 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. There is also the question if higher priority measures are not satisfied, does it make sense to satisfy lower priority measures. In our opinion, we would like to first satisfy the user critical intent first and then satisfy the others as closely as possible. This will happen for tuple latency as load shedding will reduce tuple latency and then other measures will be handled as if the critical measure has been satisfied. Below, we present our approach for doing this.

$$Reduction\ Percentage = \frac{Expected\ Value - Observed\ Value}{Expected\ Value} \tag{4.2}$$

$$Reduced\ weight = Initial\ Weight - (Reduction\ Percentage * Weight\ Range) \tag{4.3}$$

The priority-wise decision making scheme disallows switching for lower priority measures till 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 larger weights associated with higher priority measures can dominate computation of scores when decisions are made for lower priority measures. In our approach, this is alleviated by reducing weights for higher priority measures. The margin by which the weights are reduced are computed based on the percentage by which the monitored values are less tha the expected values as shown in equation (4.2). The reduction percentage is multiplied with the range of

Measures	**Weights**
Tuple Latency	1
Memory Utilization	0.5
Throughput	0.01

Table 4.4 QoS Measures and Weights

variation allowed for a priority class, which is the difference in the specified values of weight for the priority class and next lower priority class to obtain the amount by which weight has to be reduced using the equation (4.3). The intuition behind using percentage reduction is that more the QoS measures are less than the expected values, more they can tolerate adverse strategies. The lowest weight a measure in higher priority class can take must be greater than the weight of next lower priority class to avoid priority inversion. Thus by using the reduced weights of satisfied measures, we ensure that the strategy chosen is not detrimental to higher 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 4.4 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 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 (4.3). 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 4.5 .

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 4.5 Measures Different Priority Classes

4.2.5 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 picks up the constructs and starts scheduling them. Further the effect of the strategy becomes visible only after all the operators are scheduled in the new strategy for few times. This requires some elapsed time before which the effect of the new strategy cannot be ascertained. As a consequence, the expected values given in the QoS graph can be different by the time the effect of the new strategy comes into effect. In case all QoS measures have higher requirements the new strategy will strive towards achieving expected values as violated measures remain the same. But in a scenario where only some QoS measures have higher requirements the chosen strategy may not be the best for the new set of expected QoS values. This is also the case if some or all of the QoS measures have lower expected values. In the worst case the switch to the new strategy would be totally unnecessary.

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.

4.2.6 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 choose strategies in a cycle. This cycling through 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 and we may be better of choosing a compromising strategy to mitigate such situations.

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. The strategy that comes out is chosen without further checks. This method, though it prevents cycling, can sometimes prevent a genuine change to an older strategy. But this situation is much better than cycling through strategies that can introduce a lot of overhead.

The above mentioned technique can be easily extended to check for cycling through more than two strategies. But this will incur the additional overhead of keeping track of multiple strategies and more comparisons. The approach also increase the probability of preventing genuine changes. The initial idea of keeping track of merely strategies that was already used and avoiding them for later decision making was not taken as it limits the number of strategies available and can prevent from changing to those when it is necessary. It is entirely possible that strategies alternate (or cycle) due to changes in expected qos values. This should not be prevented as these are genuine changes of scheduling strategies dictated by the qos requirements. Hence, we do cycle checking within a piece-wise segment of the qos input. When the segment is crossed, checking for cycles will start from scratch.

4.2.7 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

1. Minimizing the frequency of monitoring, and
2. Minimizing the number of strategy switches

4.2.7.1 Monitoring Interval

The number of strategy switches and computation performed for that purpose are determined by the number of times (or the intervals at which) a query is monitored. Higher the monitoring frequency more is the overhead of comparing measures and switching strategies. Hence the frequency of monitoring is critical in determining the overhead of the runtime optimizer. If the number of times a query is monitored is too small, the possibility of not taking decision for achieving QoS at the right times increases. Hence there should be a compromise between the two extremes.

The monitoring interval can be determined from the QoS graphs by specifying the number of times a query should be monitored over each interval of input. This assumes that all the intervals specified in the QoS graphs are almost of the same length. For longer intervals the query should be monitored more number of times during each interval. Since output for a query is produced only when all the operators are scheduled at-least once, the minimum monitoring interval should be greater than the time required to schedule all the operators of the query. An upper limit on the monitoring interval can also be placed by specifying a multiple of the minimum monitoring interval time.

4.2.8 Decision Maker Algorithm

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. 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

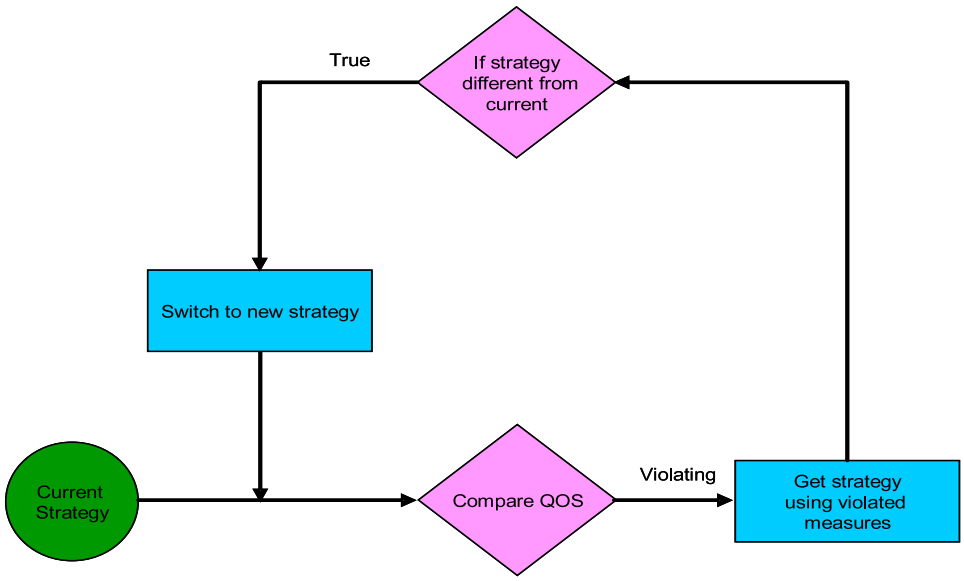

Figure 4.3 Runtime Optimizer Actions

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. 4.3

4.2.9 System Monitor Algorithm

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

4.3 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 compete 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. This will lead to a situation where a scheduler never gives up control of CPU causing other schedulers to starve.

The Master Scheduler is similar to the two level scheduling proposed in Aurora. But, 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. In this thesis, we introduce two level scheduling scheme wherein the first level scheduler termed Master schedulers controls the time allocated for each scheduling strategy. Master scheduler runs as a separate thread allocating time for each schedulers to execute. The master scheduler follows a fair scheduling scheme by going in 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.

4.3.1 Synchronizing the Switching Process

Changing the scheduling strategy of a query involves removing the schedulable objects of a query from the ready queue of current scheduling strategy and inserting it into the ready queue of new scheduling strategy. As schedulers and runtime optimizer execute in different threads the issues of synchronization need to be correctly handled. To avoid synchronization issues that arise due to independent threads, we introduce a strategy change request queue for the master scheduler. When the runtime optimizer decides to change strategy for a query it notifies the master scheduler to change the strategy and continues execution. This alleviates the problem of optimizer thread waiting for a query to change strategy.

Master scheduler, prior to allocating time for all schedulers, checks its strategy change request queue. If any requests are present in the queue, all of them are processed before it allows the schedulers to execute. Though this process also requires synchronization between master scheduler and runtime optimizer, the amount of time runtime optimizer would have to wait is less than waiting time for a query to change strategy. Further as the master scheduler processes the change request before allocating time for all scheduler no operators of the query will be executing when the strategy is changed. This provides the the advantage of reducing the synchronization waits that occurs between individual schedulers and the master.

4.4 Load Shedding

The resources available to any DSMS is limited. It is possible that the arrival rate of input streams can exceed far beyond processing capabilities of the system. These situations lead to the system being clogged with a large number of tuples, increasing tuple latency and memory utilization. As some of the stream processing applications can tolerate approximate answer but not delay in results, dropping some tuples can provide better results for QoS measures. This process is known as load shedding. *Load shedding* is formally defined as the process of gracefully discarding unprocessed or partially processed tuples improving the QoS constraints of continuous queries. If sufficient number of tuples are discarded, the processing capacity will match the load and satisfy QoS measures. Due to dropping of tuples, errors are introduced in the output values. Load shedding is an optimization problem and consists of the following subproblems

- When and How much to shed load
- Where to shed
- How to shed.

Load shedding is a trade off between tuple latency and accuracy and various techniques for load shedding have been proposed. In this thesis we have proposed a feedback based load shedding approach that adds the least overhead to the system. The predefined QoS requirements considered also includes the most-tolerable relative error (MTRE) of a continuous query in its final query results.

4.4.1 Notations

The following notation are used in this section.

- *Operator processing capacity* $C_{O_i}^P$: the number of tuples that can be processed within one time unit at operator O_i. Inversely, the operator service time is the number of time units needed to process one tuple at this operator. A join operator

or k-way operator is considered as two or k semi-operators. Each of them has its own processing capacity, selectivity, and memory release capacity.

- *Operator selectivity* σ_i: it is the same as in a DBMS except that the selectivity of a join operator is considered as two semi-join selectivities.
- *Operator path processing capacity* $C^P_{P_i}$: the number of tuples that can be processed 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 a simple 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 [14], as follows:

$$C^P_{P_i} = \frac{1}{\frac{1}{C^P_{O_1}} + \frac{\sigma_1}{C^P_{O_2}} + \frac{\sigma_1\sigma_2}{C^P_{O_3}} + \cdots + \frac{\prod_{j=1}^{k-1}\sigma_j}{C^P_{O_k}}} \tag{4.4}$$

  where $O_l, 1 \leq l \leq k$ is the l_{th}operator along P_i starting from the leaf node. The denominator in (4.4) is the total service time for the path P_i to serve one tuple. 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* $C^P_{S_i}$: the number of tuples that can be processed within one time unit by the operator segment S_i.

4.4.2 Location of Shedders

Load shedders come into action when DSMS is unable to meet QoS requirements of a query. The inability to meet QoS requirements denotes arrival rates that cannot be processed by the system indicating high load. Hence any additions to a query will increase the load on an already loaded system. This necessitates that load shedders

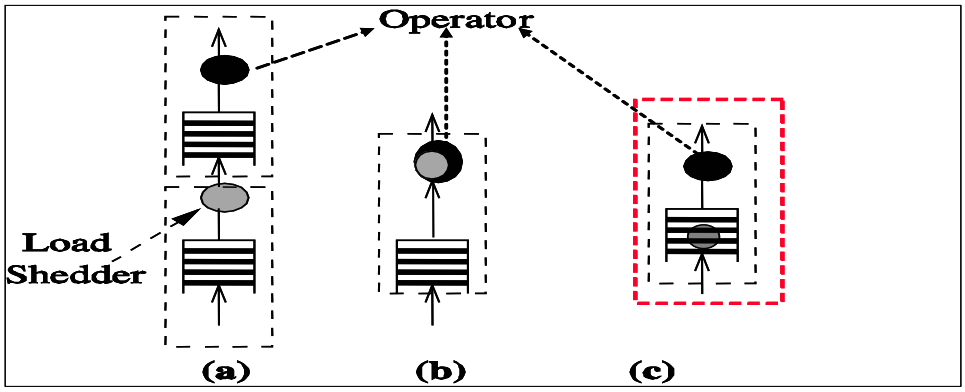

Figure 4.4 Load Shedders: Placement

introduce the least possible overhead. Most of the current load shedding mechanisms have proposed special purpose operators [19, 20] to drop load. This requires the insertion of new operators in to a query plan when the system is loaded. In addition to being a special operator, load shedding can be either part of the operator or the queues as shown in Fig. 4.4.

Among the three possible locations, the option of adding a load shedder using a special operator incurs the highest overhead as it needs to be inserted into a query plan and requires scheduling. Also, the load shedding operator has an input queue which buffers tuples before the operator decides to drop a tuple or not. The option of adding load shedding as function of an operator costs less than the first option as the load shedder need not be scheduled separately. But it still buffers tuples that are likely to dropped. This makes the last option of adding load shedders as part of the queues most viable. By making load shedding a function of input queues, decision to drop tuples is made when a tuple is enqueued. In addition to saving CPU cycles this option reduces the memory requirement as well by dropping tuples as early as possible.

4.4.3 When and How Much to Shed

The goal of load shedding is to drop tuples to meet QoS requirements of query while maintaining the error in results within tolerable limits. The load shedding techniques proposed in Aurora [19] estimates the current load on the system based on the arrival rates and tuples waiting in the queues of operators. The system starts inserting drop operators when it detects that load is greater than H x System Capacity where H is the headroom factor. A congestion avoidance technique has been proposed in [21], which estimates the load of the system based on the current arrival rates and tries to prevent an overloaded state from occurring. Since these techniques require the estimation of system load which is based on the system characteristics, they may not be accurate and requires additional overhead. Since load shedding mechanisms work to achieve the QoS requirements of a query the feedback on their performance can be used to determine whether to shed load or not.

4.4.4 Load Shedding by Runtime Optimizer

The runtime optimizer determines the measures violated by comparing monitored values to expected values. If the runtime optimizer does not find better strategies to meet QoS it can invoke the services of the load shedder to save computational time. Load can be continued to shed as long as the error introduced is within the tolerable limits specified by MTRE. As soon as QoS measures start meeting the expected values, load shedding can be deactivated. As the runtime optimizer makes decisions based on a the *looakahead factor*, runtime optimizer with the help of load shedders make sure that the right actions are taken to meet QoS requirements of a query. The use of feedback from the system monitor provides a reliable metric on the actual performance of a query in the system . Hence, this feedback can be used to control the operation of load shedders without adding any overhead. The algorithm for Decision Maker with load shedding in

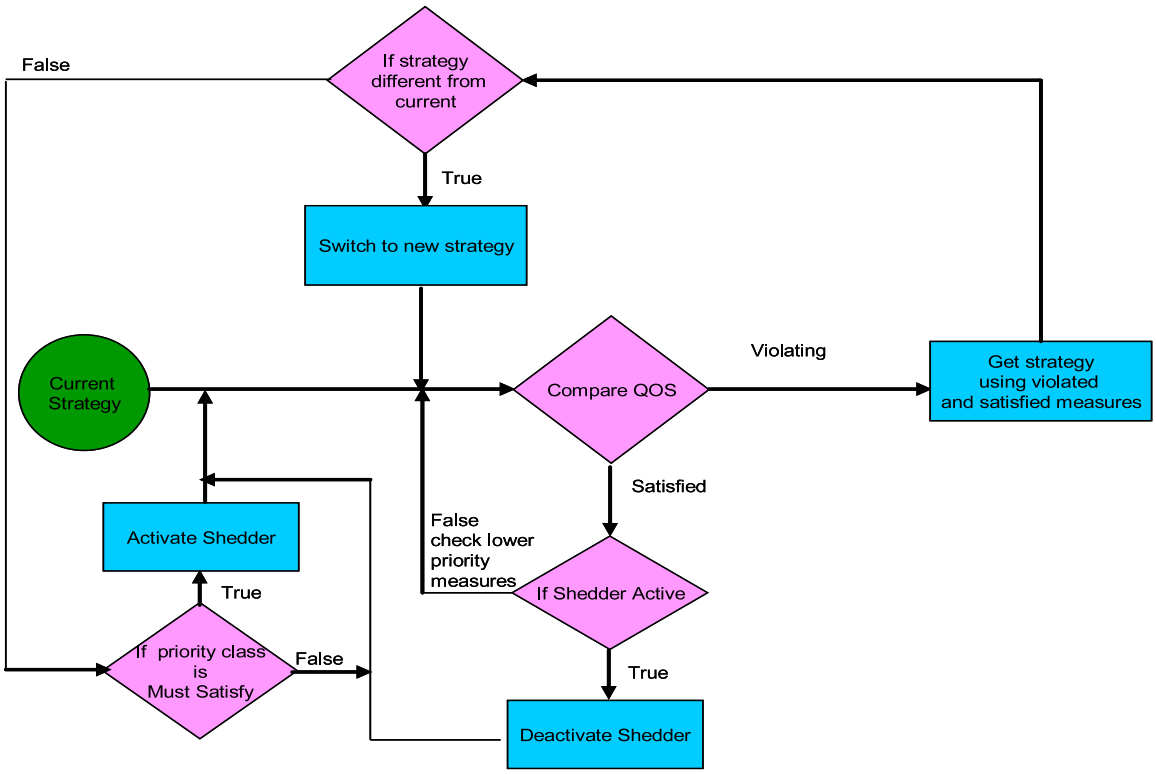

Figure 4.5 Runtime Optimizer Flowchart

shown in Algorithm 4. The flowchart for runtime optimizer with shedding is depicted in Figure 4.5.

4.4.5 Where to Shed

As discussed earlier, load shedders are incorporated into the buffers to minimize the overhead. As each active load shedder needs some CPU cycles to decide whether to keep or drop a tuple it adds some overhead to the system. To reduce this overhead, the number of active load shedders should be minimum. Towards this goal, for each operator path, we place at most one potential load shedder in its path. This requires finding an optimal position for the load shedder. We use the concept of place weight of shedders developed in [21] which is some what similar to the loss/gain [19] ratio used in Aurora.

The placement of load shedders have a different and significant impact on the accuracy of the final results, and on the amount of computation time units it releases. Placing a load shedder earlier in the query plan is most effective in saving the computational time units but its effect on the accuracy is likely to be the most. On the other hand, placing a load shedder after the operator which has biggest output rate in the query plan has the lowest impact on accuracy when a tuple is dropped, but the amount of computation time units released and the storage saved may not be the largest. Therefore, the best candidate location for a load shedder along an operator path is the place where the shedder is capable of releasing maximal computational time (and possible storage) units while introducing minimal relative errors in final query results by dropping one tuple. The formula for calculating place weight described in [21] is reproduced below 4.4.5.1.

4.4.5.1 Calculation of Place Weight

In a simple operator path $\mathcal{X}$ with k operators there are k candidate places to place a load shedder. Let $\{x_1, x_2, \ldots, x_k\}$ be its path label string, and v be the input rate of the data stream for this operator path. Let $b_1, b_2, \cdots, b_k$ be its k candidate places, where $b_i, 1 \leq i \leq k$ is the place right before the operator x_i. The place weight W of a candidate place is the ratio of the amount of saved percentage of computation time units α to the relative error ϵ in its final results introduced by a load shedder at that place by discarding one tuple. The place weight W of a shedder at a particular location of an operator path with its Most Tolerable Relative Error ($MTRE = E_i$) is defined as:

$$\mathcal{W} = \frac{\alpha}{\epsilon} \tag{4.5}$$

$$\alpha = \frac{v(d)}{\mathcal{C}^S} - \frac{v_{shedder}}{C^O_{shedder}}$$

$$\epsilon = \begin{cases} \frac{v(d)}{v_{shedder}} & \text{for a random shedder;} \\ f\left(\frac{v(d)}{v_{shedder}}\right) & \text{for a semantic shedder;} \end{cases}$$

$$v(d) = \begin{cases} E_i * v_{shedder} & \text{for a random shedder;} \\ E_i * f(\frac{1}{v_{shedder}}) & \text{for a semantic shedder;} \end{cases}$$

$$v_{shedder} = v \prod_{i=x_1}^{x_n}(\sigma_i), x_1 \text{ to } x_n \text{are operators before the shedder}$$

where $\mathcal{C}^S$ 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. If there is no operator after the shedder, $\mathcal{C}^S$ is defined as infinity and ($\frac{v(d)}{\mathcal{C}^S} = 0$). The computational units that can be saved is therefore zero and the shedder itself introduces extra overhead by $\frac{v_{shedder}}{C^O_{shedder}}$. $v(d)$ is the maximal drop rate at which shedder can drop tuples at this location without violating the MTRE E_i defined for that operator path. $\frac{v(d)}{\mathcal{C}^S}$ is the total computation time units it saves by dropping tuples at a rate of $v(d)$. However, a shedder also introduces additional overhead, which is $\frac{v_{shedder}}{C^O_{shedder}}$, to the system as it needs to determine whether a tuple should be dropped or not. $v_{shedder}$ is the input rate of the load shedder, and 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. If an operator is a leaf node the shedder, then $v_{shedder} = v$ and v is the input rate of the stream for the operator path $\mathcal{X}$. $\mathcal{C}^O_{shedder}$ is the processing capacity of the load shedder. If the load shedder is a semantic one, $f(.)$ is a function from selectivity of the shedder to the relative error in final query results.

In equation (4.5), the input rate of stream is the only parameter that changes over time. All other items[1] do not change until we revise the query plan. Therefore, for an

[1]Selectivity of an operator may be revised periodically. However, it is assumed that it does not change over a long period of time.

operator path $\mathcal{X}$, there are k candidate places for A load shedder. We compute the place weight for each of those k candidate places. The partial order of load shedders in a path do not change as input rate changes because all of them have the same input at any time instant. Hence, the place where the load shedder has the biggest place weight is the most effective one.

The arrival rates of input streams can be monitored and the mean rate of arrival can be used for calculating place weight. The processing capacity of operators and shedders can be determined by collecting statistics for the system. The processing capacities can be determined from the average service time required to process a single tuple. The drop rate is assumed to be the maximum that is allowed without violating MTRE. This can be determined by using mean arrival rate and selectivity of operators as mentioned in the above equation (4.5).

4.4.6 How to Shed Load

The location of shedders are determined before the query starts executing. Since the number of active shedders has to be kept to a minimum to minimize the overhead, we have to ensure that maximum savings is achieved by activating any shedder. Since the shedders are initialized at positions having the highest place weight we can obtain the maximum gains by activating the shedders with the highest place weight. Therefore when the runtime optimizer detects that queries are likely to violate QoS requirements, it takes a greedy approach and activates shedder with the highest place weight in the list of non active shedders.

To achieve highest gains the list of load shedders are kept sorted by their place weights. The list is kept sorted when location of shedders are determined and hence does not involve any overhead. When activated, the load shedder starts dropping tuples at maximum allowed drop rate. The runtime optimizer keeps the load shedder active until

either the QoS requirements are met or when it finds a better strategy for the query after dropping some tuples. The load shedders are deactivated when a new strategy is found so that the results produced by the query is closer to the actual. Further the availability of a new strategy can denote potential availability of resources to meet QoS requirements without dropping load.

The place weight of a load shedder changes with the change in input rate of a data stream. Although this change does not change the partial order of shedders along the same operator path, it can change the partial order of shedders of two different operator paths when the ratio of the current input rate to the input rate used to compute the place weight changes dramatically. Therefore, we have to reorder the non-active shedders in the system when we allocate the total shedding load among all non-active shedders. But sorting of load shedders when input rate changes may add high overhead in a heavily loaded system with dynamic input rate. Hence we do not sort the load shedders. Also unlike Aurora [19] where load can be dropped at any location when system is determined to be overloaded, we drop load individually for each query. Load shedders are activated only in the input queues of operators in a query that violate QoS requirements. This approach will result in higher savings for a query than approaches where load shedding can be performed at any location. As the number of load shedders for a single query will be small, the sorting overhead will be minimal. Further, as shedders are activated for each query separately keeping the shedders sorted may be unnecessary.

Based on the feedback, runtime optimizer frequently activates non-active shedders or deactivates active shedders. The overhead for activation and deactivation is minimal as load shedding is a function of the buffer. This involves setting and unsetting a Boolean value. The actual determination of shedding a tuple can be done either randomly or using some semantics specified by the application. This has lead to development of two variants on shedders.

4.4.6.1 Random Load Shedders

As the name indicates, these load shedders drop tuples randomly. The random load shedder is modeled as p gate function. The maximum drop probability is set when the place weights are calculated. The random load shedder can drop tuples at probability set by the runtime optimizer or its maximum probability. The shedder is designed such that it does not allow tuples to be dropped at a rate above the maximum probability. For every tuple a random value is generated which is compared against the drop probability. The value determines whether a tuple will be dropped or not. One of the key assumptions in random load shedding is that the error introduced is directly proportional to the drop rate. This may not be the case with operators like join which results in higher error rate as a single tuple can potentially join with many tuples.

4.4.6.2 Semantic Load Shedders

Semantic load shedders are similar to filter operators. These shedders require the semantics to be specified along with the query. The user, based on the requirements of application, can specify the range of an attribute that can be dropped. For example, in military triage application [4], at times of high load, tuples that report normal temperature and heart beat rate for soldiers can be dropped. As the specification requires some knowledge about the data, we have implemented a simple semantic load shedding technique for numeric attribute where the user can specify the attribute and range of values that can be dropped. This requires very little knowledge about the characteristics of data. The three values for range of the attributes are

- Lowest first : Drops the values in the lower range for the specified attribute first.
- Center first : Drops the values in the center range for the specified attribute first.
- Highest first: Drops the values in the higher range for the specified attribute first.

The list of load shedder buffers that are likely to be activated are the ones with highest place weight which is obtained while computing the location of shedders. For semantic shedders only buffers whose input stream contains the attribute to be used for load shedding are added to the list of load shedders. The semantic load shedder keeps track of values for the attribute of interest and when activated determines the range of values dynamically that will be used to determine whether a tuple has to be dropped. This monitoring is done at all the buffers where semantic load shedder can be activated. The lowest first drops tuples that fall into the range of lowest values, while center and highest first drop tuples that fall into the range of the mean and highest value, respectively. The semantic load shedder assumes that range of values for the attribute are equally distributed. If there is an uneven distribution of values, the gains from the load shedder will vary. Also by dropping tuples whose values are of less interest the semantic load shedders introduce minimal errors in the results.

4.5 Summary

In this chapter we have discussed the design issues and assumptions made by run-time optimizer for choosing the best available scheduling strategy. We have also introduced the decision table using which we base our selection of strategies and *lookahead factor* and other mechanisms to ensure proper selection of strategies. Later we have explained the problem of load shedding and the low overhead solution that we have proposed.

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 ;
```

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             ;
11         end
12         wait minnexttimetomonitor − currenttime
13         ;
14     end
15 end
```

Algorithm 3: Master Scheduler Algorithm

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

Algorithm 4: Runtime Optimizer Algorithm With Load Shedding

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 AND(
       isLoadShedderActive == False OR PriorityClass = Mustsatisfy )
       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 AND PriorityClass = Mustsatisfy
       then
20     isLoadShedderActive ← true;
21     activate load shedders;
22     else
23         previousstrategy ← currentstrategy;
24         currentstrategy ← newstrategy;
25         if isLoadShedderActive == true then Deactivate shedders;
26         Record current strategy details and switch to new strategy;
27     end
28     else
29         if isLoadShedderActive == true then Deactivate shedders;
30         else
31             HigherPrioritySatisfied ← true;
32         end
33     end
34 end
35 return get next time to monitor ;
```

CHAPTER 5

IMPLEMENTATION

In this chapter we go in depth into the implementation details of the runtime optimizer and load shedder discussed in the chapter on design **??**. We also present the reasons for our implementation choices and some of the issues encountered. We also delve into the implementation aspect of specifying QoS measures and how it is used by the runtime optimizer.

5.1 QoS Specifications

A query has to be provided with QoS requirements for the runtime optimizer to take necessary actions. The three QoS measures considered are: Tuple Latency, Memory Utilization and Throughput. As mentioned in chapter 4, the QoS measures fall into one of the three priority classes. For applications that can tolerate error in their results the Max Tolerant Relative Error (MTRE) also needs to be specified for the system to perform load shedding.

The QoS requirements are optional components of a query and the user needs to specify only those measures that are of interest. QoS measures are expressed as piecewise linear functions along with a query as a series of x, y pairs. The x values denote the relative time from the start of a query and y values denote the value of expected measure in appropriate units. For simplicity the x values and their corresponding y values are presumed to be given in increasing order of time. The input processor on parsing the text input stores the name of each QoS measure, priority and list of x, y values in a separate list for each measure of interest. This list is part of query plan object of a query.

5.1.1 Extensions to Input Processor and Query Plan Object

The runtime optimizer and load shedders require QoS parameters and tolerant error limits to be specified along with a query. The parameters of interest can be specified along with a query in the input text file as *Parameter Name: Priority Class:* $x_1, y_1, x_2, y_2, ...x_n, y_n$. The Input Processor processes the text input and adds all information for a QoS parameter into a list. The first element in the list is the name of QoS measure and second element is the priority class to which it belongs. This is followed by a series of *x,y* pairs. The information for all QoS measures are stored in an array of lists. Each index of the array holds all the information for a specific QoS measure. The *QueryPlanObject* therefore has been extended to hold the QoS information in an array of *Vector* objects. The size of the array is determined by the number of QoS measures the system supports. The array of QoS measures have their corresponding getter and setter methods.

The data for each interval is placed into a structure called *QosParamData* when input processor processes query plan object. This structure consists of two *(x, y)* value pairs that denote the beginning and end of an interval. The list of intervals along with the name of QoS measures and priority are stored in another data structure called *QoSParameters*. The *QoSData* object encapsulates an array of *QoSParameters* object. *QoSData* object organizes the QoS measures in specific indices of an array based on their type. The runtime optimizer uses the *QoSData* object to get the expected values for the QoS measures. Query plan object also stores the information about the percentage error a query can tolerate and their corresponding methods. The Input Processor sets the *isOptimizerTrue* to true if a query has QoS parameters associated with it. This variable is used to determine whether a query needs the services of runtime optimizer. The Input Processor later uses these information to create *QoSParamData* and *QoSParameters* objects and compute optimal locations of load shedders . The *QoSParameters* object and list of load

shedders are populated into a *QosData* object and given to the runtime optimizer. The following are the methods that process QoS measures in Input Processor.

- *processQosParameters* : The method processes the array of QoS measures from the *QueryPlanObject* and creates a *QoSParamData* object for each interval and places them into the *QoSParameters* object along with priority and name of measures. This object is given to the runtime optimizer.
- *computeLocationOfShedder* : This method finds the optimal location of shedder by computing the place weight at input buffer of each operator in the paths using the formulas given in chapter 4. For each path the input buffer with the highest place weight is put into a list which is kept sorted based on the place weight. This is performed using the *insertToSortedList* method.
- *initializeShedder*: The *initializeShedder* method calls appropriate methods to initialize load shedder based on the type of the load shedder for all the optimal locations in a query. For a random load shedder, the maximum drop probability that will not violate the error tolerant limit is set for each location using the equation (4.5). For semantic shedder the method looks for the presence of attribute on which semantics have been specified in input streams of load shedder buffers. If the attribute is present, the position of the the attribute and the range to shed is initialized in the buffer. This information is used to monitor the values and drop tuple when shedder is activated. After the initialization the buffer is inserted in to the sorted list using the *insertToSortedList* method.

5.2 Runtime Optimizer

The runtime optimizer is a separate thread in the system that monitors and makes decision for all queries. Rather than using a separate thread to monitor each query, a single thread to monitor all queries was chosen as this option reduces the overhead

that arises due to cost associated with creating multiple threads. Also the single thread option allows the optimizer thread to be bound to a separate processor so that it does not interfere with query processing. The runtime optimizer is made up of the following two classes: *System Monitor* and *Decision Maker*.

5.2.1 System Monitor

System Monitor monitors QoS measures of queries and arrival rates of input streams. System Monitor also encapsulates *Decision Maker* object which makes decisions for altering scheduling strategies and invoking load shedders. System Monitor runs as a separate thread which is started on system start up and waits for queries. When a query that requires the services of runtime optimizer arrives, the System Monitor is notified by the server. On notification the System Monitor wakes up and starts monitoring queries at appropriate time instances.

The system monitor provides the following methods to the server for monitoring a query and removing it after completion.

- *addQueryToMonitor*: This method is used by the server to add a query which requires the services of runtime optimizer. The server has to provide *SystemMonitorData, QoSData* objects along with query ID and reference to scheduling objects. The *QoSData* object is given to the Decision Maker by the System Monitor.
- *removeQueryFromMonitor* : This method is used by the server to remove all data structures used by System Monitor and Decision Maker after a query completes execution. The method takes in query ID and removes all the data objects of query from runtime optimizer.

The System Monitor also maintains a Hashtable which uses query id as the key and *SystemMonitorData* object as value. *SystemMonitorData* contains methods and data members to monitor QoS measures of a query and also holds the next time at

which a query should be monitored. The System Monitor enumerates 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 at that instant it uses the methods in the *SystemMonitorData* to obtain the QoS values and provides it to the Decision Maker. The Decision Maker compares QoS values and takes the appropriate action and returns the time the query needs to be monitored next. The System Monitor updates the next time to schedule for the query in *SystemMonitorData* and also keeps track of the earliest time for monitoring a query among all those that need to be monitored. This is used to determine the amount of time the monitor should sleep.

5.2.1.1 System Monitor Data

The *SystemMonitorData* class stores all data required to monitor QoS values and arrival rates of input streams for a query. This includes a reference to all input buffers, leaf nodes of a query and output buffer of root operator of a query. The next time a query should be monitored is maintained in *lngTimeToMonitor* variable. The time maintained is the absolute time. The following are the methods in the *SystemMonitorData* class.

- *getMemoryUtilization* : This method gives memory used by tuples in the input buffers of operators in the current query.
- *getThroughput* : Returns the throughput as tuples/sec over the last period of monitoring. It is calculated by keeping track of the number of tuples output by the root operator.
- *getArrivalRate* : Returns the arrival rate during the last period for each input stream of the query.
- *getTimeToMonitor/setTimeToMonitor*: These methods allows to set and get the next time at which query needs to be monitored.

5.2.2 Decision Maker

Decision Maker is the brain of the runtime optimizer. It makes all decisions for a query when requested by THE System Monitor. The System Monitor provides the Decision Maker ID of a query and monitored values. Decision Maker maintains all required information regarding the QoS of a query in *QoSData* object. Decision maker obtains the expected values of query from methods provided in *QoSData* object, compares them with monitored values. If it finds that QoS measures are being violated, Decision Maker tries to find a better strategy using the decision table. If no better strategy is available, then the Decision Maker uses methods in *QoSData* object to activate load shedders. Decision maker also holds a reference to Master Scheduler for requesting scheduling strategy changes for a query.

The following are the member variables of *DecisionMaker* class

- *decisionTable*: This is the variable that holds Decision Table. It is a two dimensional array where each row corresponds to a specific QoS measure and each column corresponds to a specific scheduling strategy. The values of array correspond to the scores of strategy .
- *hstqueryIdQosData*: This Hash table holds *QoSData* objects for all queries with query ID as the key value.

The public methods in *DecisionMaker* class are

- *compareQosMeasures*: This method takes in a query ID, monitored values and the arrival rates for that query and returns the absolute time when query needs to be monitored next.
- *getCurrentStrategy*: Returns current scheduling strategy for a particular query.
- *addQosData* : Adds the *QoSData* for a query into the Hash table of Decision Maker. This method is called when a query that requires the services of runtime optimizer arrives.

Decision Maker also requests the Master Scheduler to change strategy of a query if it decides upon new strategy for a query. The private methods of Decision Maker do all the required work.

- *compareQosValue*: Compares the monitored values to expected value for all measures of a particular priority class.
- *getReducedWeights*: Computes the reduced weights of QoS measures that are satisfied.
- *getViolatingMeasures*: Compares expected and measured values for a particular measure and adds QoS measures that are violated to list of violating measures.
- *getSchedulingStrategy*: Uses the decision table to get better strategy using violating measures and satisfied measures with their reduced weights.

5.2.2.1 QoS Data

The *QoSData* object holds all data of QoS measures given as part of the query. It also holds the list of load shedders for the query and provides an interface to activate and deactivate load shedders.The data members of the class are as follows.

- *qosValues*: This variable stores an array of *QoSParameters* object. When the QoS parameters of a query are processed by the Input Processor, it creates a list of intervals for each QoS measure of interest. The list for each QoS measure is stored at a particular index in the array based on the type of QoS measure. This allows the QoS values to be accessed without any delays.
- *hstPriorityIndex*: This hash table stores priority and list of measures that fall into that priority. Using a hash table provides efficient means for the Decison Maker to find out all the measures that fall into the priority being considered. The values in the list consists of the index of the QoS measures.

- *hstSatisfiedMeasureWeights* : This hash table stores list of measures whose QoS requirements are satisfied and their reduced weights. For every monitoring cycle this hash table is cleared.
- *previousViolated, previousSatisfied*: These two lists are populated when Decision Maker decides to alter the scheduling strategy of a query. They keep track of the measures that are violated and satisfied when scheduling strategy of a query is changed. These lists are used by *checkWithPreviousStrategy* method which does the cycle prevention check.
- *loadShedder* : This list stores references to buffers where load shedding can be performed. The list of buffers are kept sorted according to place weight of load shedder.

In addition to the above the *QoSData* object also has information about the start time of a query, current and previous scheduling strategies, monitoring intervals and references to all scheduling constructs of the query. The methods provided by the *QoSData* object to the Decision Maker are

- *setQoSPositions*: This method is called when a new query is added to the runtime optimizer. The method takes in the QoS parameters processed by the Input Processor and places them in the *qosValues* array at a predetermined position based on the type of QoS measure. The priority information is also taken and the index of the QoS measure is put in the hash table *hstPriorityIndex* with priority as key.
- *getExpectedQosValue*: This method returns the expected QoS values for a particular measure give the time instant. The method takes in the index of the QoS measure and the time elapsed from start of the query. As intervals for QoS measures are stored in an increasing order of time the method checks if the time elapsed falls in the interval at the head of the list. If it falls into the range of the interval at the head of the list the expected QoS values is calculated using the slope of interval

and the boundary values. Once the time elapsed crosses the interval specified at the head of the list, the interval at the head of the list is removed. This will ensure that the time elapsed lies inside the interval at the head of the list.

- *checkWithPreviousStrategy*: This routine is present to make sure that runtime optimizer does not keep cycling between two strategies. The routine checks for measures currently violated and previously satisfied and vice versa. If all measures in the list are the same it returns false which denotes a possibility of cycling . The Decision Maker then uses previous violated and current violated measures to find a new strategy.
- *activateShedder, deactivateShedder*: These methods are used to activate and deactivate load shedders for a query. These methods also keep track of the buffers where the load shedder has to be activated or deactivated from the sorted listed of load shedder buffers.
- *deactivateAllShedders*: Calls the deactivate shedder method repeatedly until all the active shedders are deactivated.
- *getNextTimeToMonitor* : This methods looks into the nearest QoS interval and determines the absolute time at which the query should be monitored next. The time between monitoring the periods is made sure to lie between the minimum and maximum interval periods specified in the system configuration.

5.3 Load Shedders

The load shedders are implemented as part of input queues of operators as they incur the least overhead and drop tuples, the earliest. Load shedding is implemented as a function in the *Buffer* class. The initial approach was to create load shedder object and to place them inside the *Buffer* object when it needs to be activated. As this involves the overhead of creating an object and then performing the checks for drop,

the simpler approach of making load shedding a function of the buffer was taken. The state of shedders are inactive initially in all the buffers. The state of load shedder is checked when every tuple is enqueued. If shedder is inactive tuples are enqueued into the buffer else the functions to determine whether a tuple has to dropped are called. The *QoSData* object maintains a list of buffers where load shedders have the highest place weight. When QoS requirements begin to get violated, Decision Maker starts activating load shedders using *activateShedder* in *QoSData* object . The activation of shedders only involve setting the state of load shedders inside the input queues to true. Once activated every tuple enqueued into the buffer is subjected to additional checks for determining whether it needs to be dropped using *checkDropCondition* function. This function determines the type of load shedder and calls the appropriate function. The way the tuples are determined to be dropped classifies the type of shedders as

- Random Load Shedder
- Semantic Load Shedder

5.3.1 Random Load Shedder

The random load shedder drops tuples at random based on a set probability. The runtime optimizer has the option of setting the drop probability with an upper limit that is determined by maximum probability. The maximum drop probability is computed when the location of load shedders are computed using the arrival rates and max tolerant relative error. The maximum drop probability specifies the highest probability at which tuples can be dropped without violating the tolerable error limits. This value is set by the Input Processor module using *initializeRandomLoadShedder* function. On activation of a random load shedder, *checkRandomDropCondition* is called. A random value is generated for each tuple enqueued. If the value generated is greater than the drop probability of the shedder, tuple is enqueued else it is dropped.

5.3.2 Semantic Load Shedder

Semantic load shedders drop tuples based on the semantics specified along with query and work similar to *select* operators. The Input Processor performs the additional check of adding a buffer to the list of load shedder only if the input stream of that path contains the attribute that will be used for load shedding. Once semantic shedder is initialized by the Input Processor using *initializeSemanticLoadShedder* function, the *monitorSemanticShedderRange* function is called every time a tuple is enqueued into the buffer. This function keeps track of the highest and lowest values for the attribute specified. On activation by the runtime optimizer the semantic shedder analyzes the values seen so far and uses them to drop the tuples based on the range specified. For example the center first shedder will drop all the tuples that fall with in a fixed percentage around the mean. The percentage of tuples to be dropped around the range of interest can be specified optionally. The *checkSemanticDropCondition, checkCondition* functions of the *Buffer* class performs the check for semantic shedder.

5.4 Master Scheduler

All the schedulers are inherited from the base class *Scheduler*. The *Scheduler* class has been extended to provide Master Scheduler with an uniform interface for the various schedulers. The schedulers now include a variable *blnThreadSuspended* that indicates the state of scheduler. This variable is set to true when it is suspended. The following methods have been added to the *Scheduler* class.

- *startScheduler* : This methods starts the scheduler thread . The thread is started with a suspended state.
- *suspendScheduler* : Suspends the execution of scheduler by setting *blnThreadSuspended* to true.

- *resumeScheduler* : Resumes the execution of scheduler by setting *blnThreadSuspended* to false.
- *runScheduler* : This method is an abstract method sub classes must implement.
- *run* : The execution of schedulers starts at this method. The *run* method calls *runScheduler* of corresponding subclass when state is not suspended. When suspended schedulers wait till it gets notified by Master Scheduler. The original implementation of schedulers used the *run* method to start scheduler which also incorporated scheduling logic . This method had been changed to *runScheduler* method with appropriate changes to notify Master Scheduler when a schedulers ready ready queue is empty. The thread for scheduler start in the *run* method of base class.

The Master Scheduler encapsulates various schedulers. On instantiation, the Master Scheduler creates objects of various schedulers and starts them in a suspended state. When a new query arrives the Master Scheduler adds the schedulable objects of that query into the ready queue of the appropriate scheduler and notifies it. Master Scheduler also allocates time to each scheduler. Master scheduler follows a weighted round robin approach as it does not give rise to any form of starvation and provides a fair amount of time to all schedulers. Master Scheduler allocates time quantum to each scheduler proportional to the number of operators in its ready queue. It also provides an interface for adding and removing queries from the schedulers. The Master Scheduler has also its own queue *timeSlicerQueue* where it receives requests to start queries and change strategies. Master scheduler checks its queue and processes the requests before allocating time to schedulers. The following are the methods in *Master Scheduler* class

- *addToBeScheduled*: This method is called to add a new query to be scheduled or change the strategy of an existing query.
- *run*: Master Scheduler executes as a separate thread. On startup it checks its ready queue and processes all the request to start query or change scheduling strategy

using the following private methods *processTimeSlicerQueue, addAndRemoveFrom-Scheduler*. After processing its queue, Master Scheduler allows each scheduler to execute for a period of time based on the number of operators in its ready queue and goes to wait. After the expiration of time quantum or on notification from the scheduler Master Scheduler suspends the execution of scheduler. After allocating time quantum for all schedulers it again processes the input queue and repeats the process.

5.5 Summary

In this section we have explained the implementation details of load shedder and runtime optimizer. The important data members and methods in each class were described. In addition we have explained the extensions made to the Input Processor module to handle QoS measures specified along with the query.

CHAPTER 6

EXPERIMENTAL EVALUATION

The design and implementation of mavstream were discussed in earlier chapters. To validate the theoretical analysis of various scheduling strategies and load shedding effectivness, experiments were performed. In this chapter we explain the various experiments conducted and analyze the results that were obtained. MavStream is implemented in Java and experiments were run using synthetically generated streams. The streams are stored in binary files and each stream is fed using a separate thread. Each feeder thread runs on a separate processor and delays between tuples follow a poisson distribution. The experiments were conducted on an unloaded machine running Red hat Enterprise Linux Application Server 4 with four dual core AMD Opteron 2GHz processors and 16GB of RAM. The maximum allowed heap for Java runtime was kept at 8GB. The Java version used is 1.5. The feeder threads and the runtime optimizer were bound to separate cores. All the query processing and scheduling was performed on a single core.

6.1 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 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.

The runtime optimizer was not allowed to activate load shedders for this experiment. Among the three strategies, as PCS provides the best performance for tuple latency 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 measures 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.**??** 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.**??** shows the latency of the query for a smaller period of time over the lifetime of query. From Fig.**??** 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

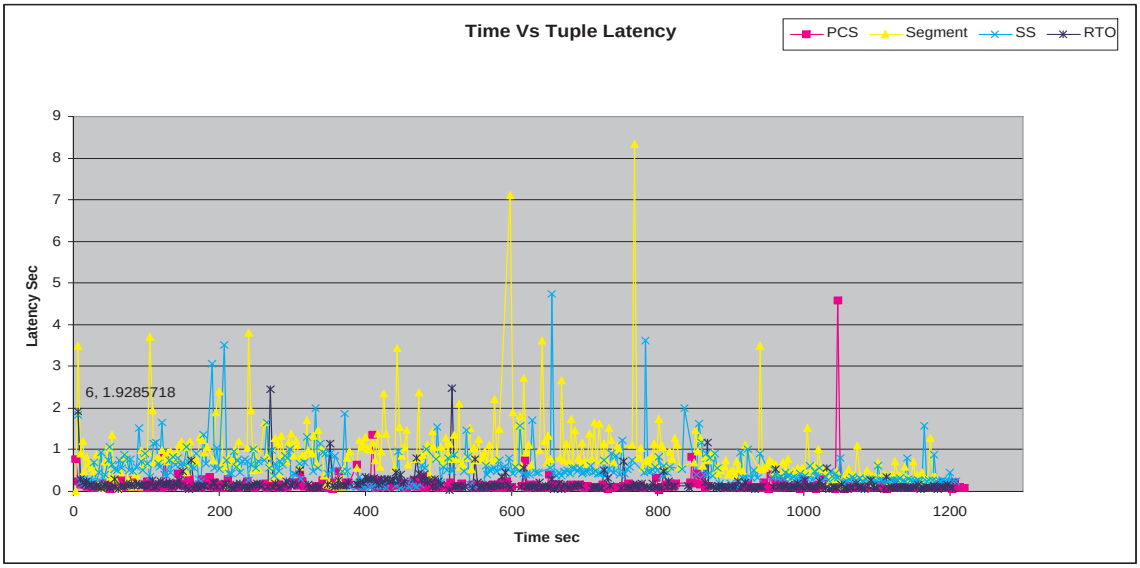

Figure 6.1 Latency: Single QoS Measure

to provide optimal performance for tuple latency in-spite of starting with an adverse strategy and monitoring overhead.

6.2 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. Load shedding was disabled for this set of experiments.

6.2.1 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

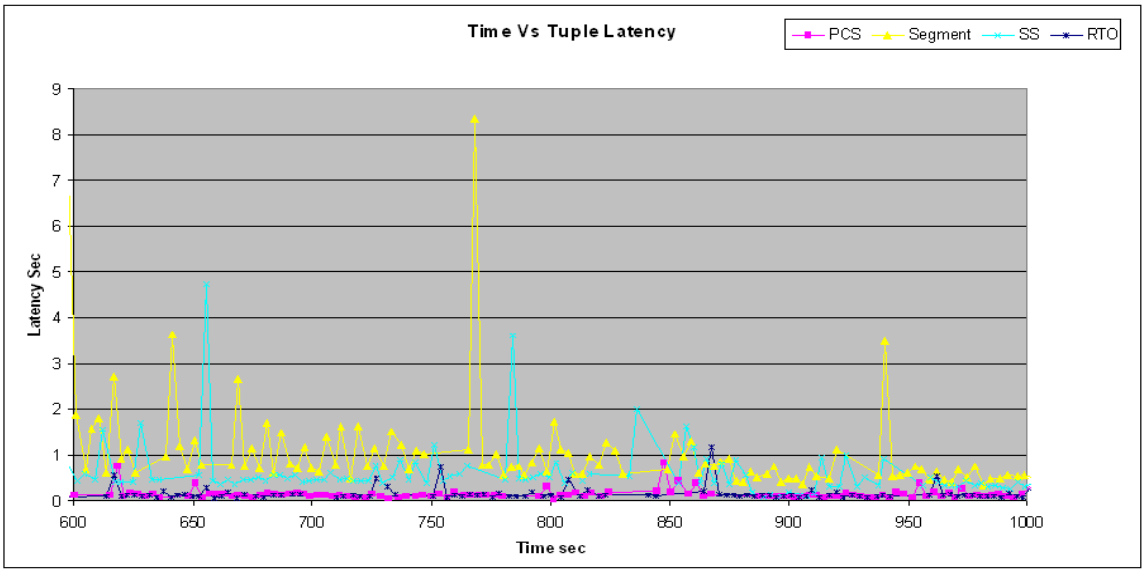

Figure 6.2 Latency: Single QoS Measure

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.**??**. 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.

As show in Fig.**??**, tuple latency using the runtime optimizer is nearly equal to latency provided by PCS which is the strategy that provides the least latency. Even though the initial strategy chosen is Segment that gives a high tuple latency the runtime optimizer is able to switch to PCS and output tuples with a delay very near to that of PCS. As no optimization is done for memory utilization the runtime optimizer and PCS both utilize higher memory than other strategies as is evident from Fig.**??**.

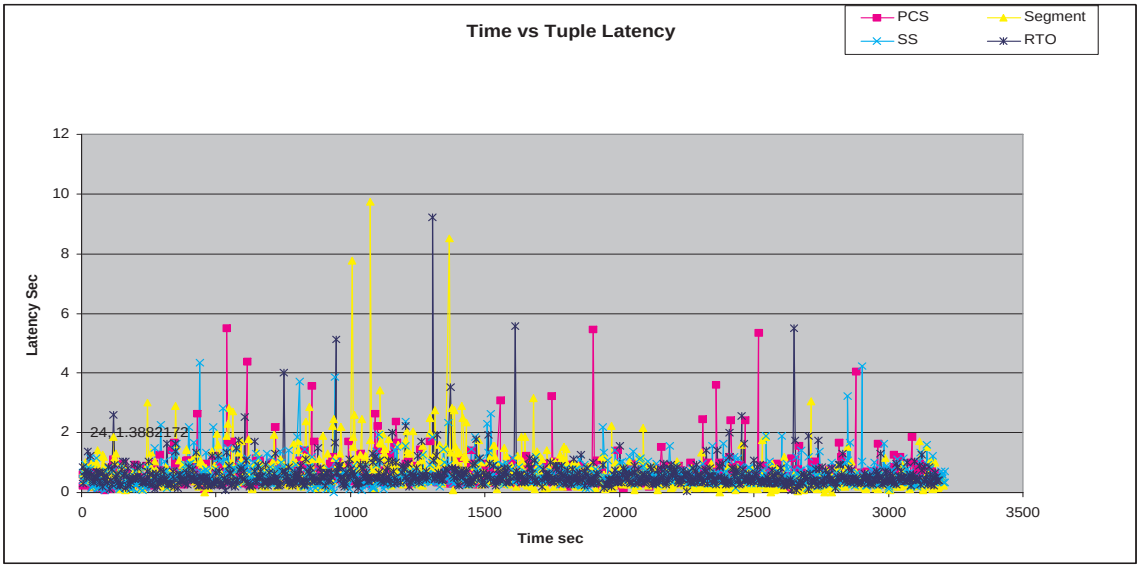

Figure 6.3 Latency: Measures With Different Priority

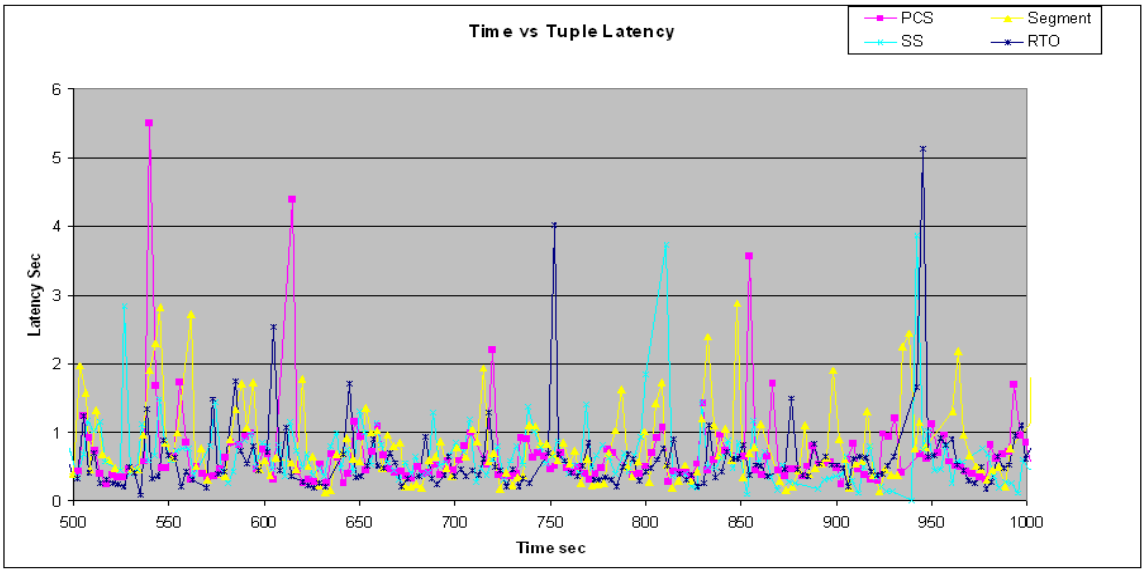

Figure 6.4 Latency: Measures With Different Priority

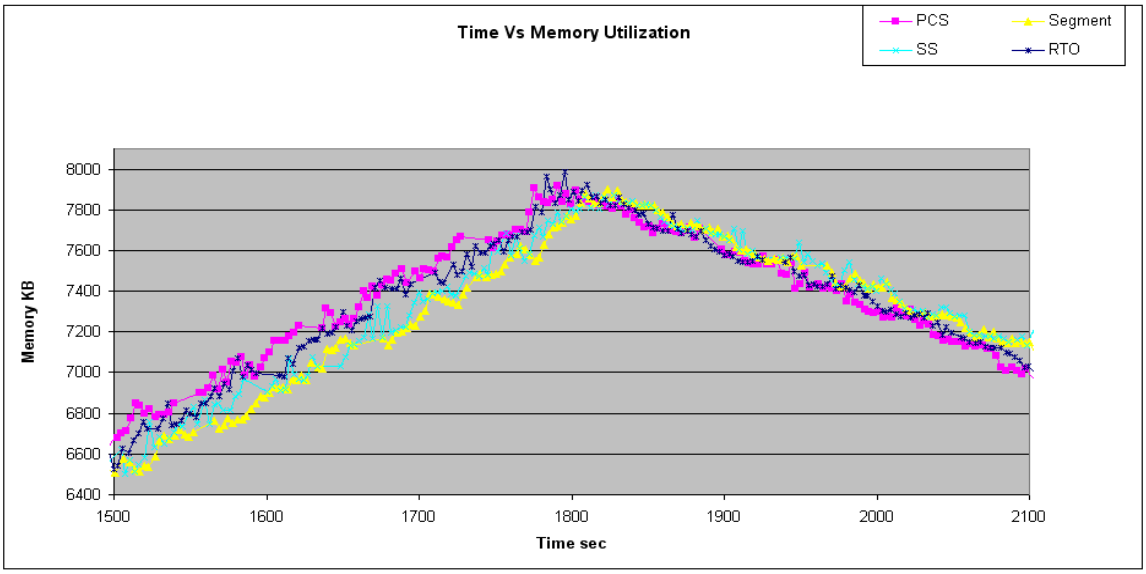

Figure 6.5 Memory Utilization: Measures With Different Priority

6.2.2 QoS Measures With Same Priority

The QoS intervals specified for this experiment was 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.**??** 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.**??** and Fig.**??** the tuple latency and memory utilization provided initially by the runtime optimizer is near to that provided by PCS.

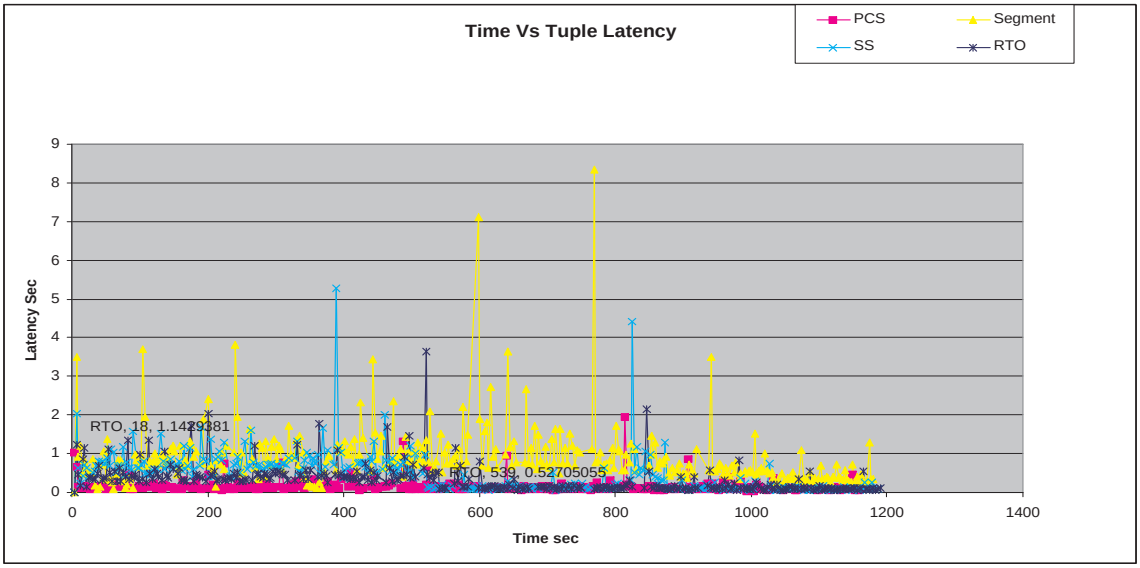

Figure 6.6 Latency: Measures With Same Priority

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.**??**. 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.

6.3 Effect of Load Shedding on QoS Measures

This set of experiments were conducted to observed the effect of load shedding on QoS measures. The performance of QoS measures provided by runtime optimizer without shedding was compared to performance of QoS measures with various error tolerance limits for shedding. The higher error tolerance limits translate to higher drop probability for the random load shedders. This will lead to more tuples being dropped

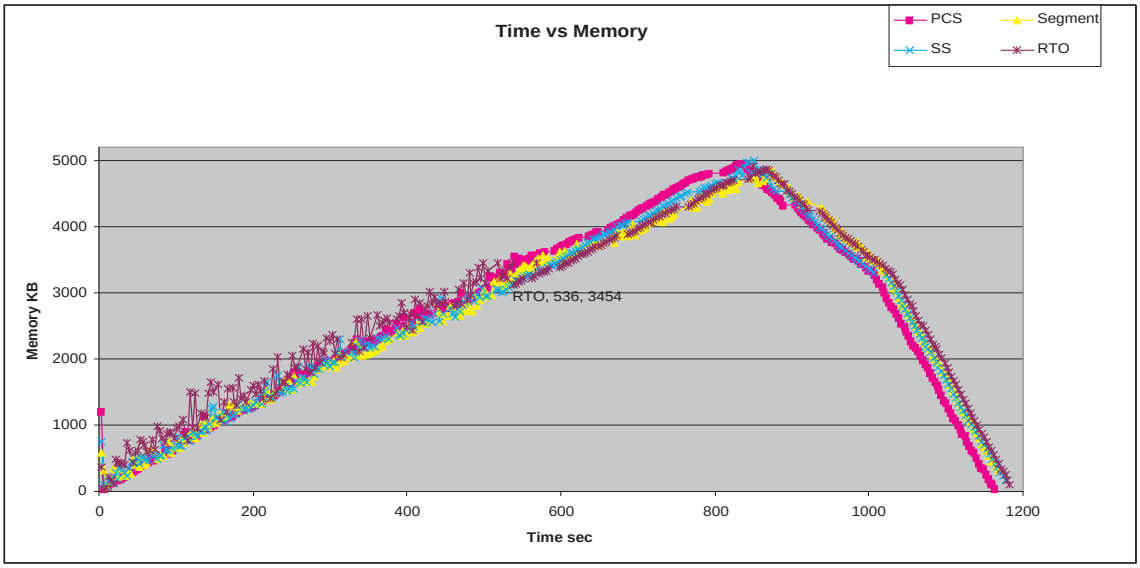

Figure 6.7 Memory Utilization: Measures With Same Priority

from the system decreasing the tuple latency and memory utilization as more tuples are dropped. The query used for these experiments was same as that of the previous experiment. Three input streams each containing 2 million tuples was fed to the query.

For the first experiment a tuple based join of 1000 tuples/window was used. The streams were fed using poisson distribution with a mean rate of 1000, 750 and 500 tuples/ second. The mean for the poisson distribution was doubled at different points in time to simulate the bursty nature of streams. The QoS measure considered was tuple latency and a single interval was specified with start and end values of 1 second. Memory utilization was set to Don't Care class and hence does not affect the decisions made by runtime optimizer. The error tolerance in results of the query was varied from 10 to 30 percent. The runtime optimizer based on the decision table choses PCS, the optimal strategy for tuple latency. The runtime optimizer after changing scheduling strategy of the query to PCS determines that latency is still being violated and hence starts

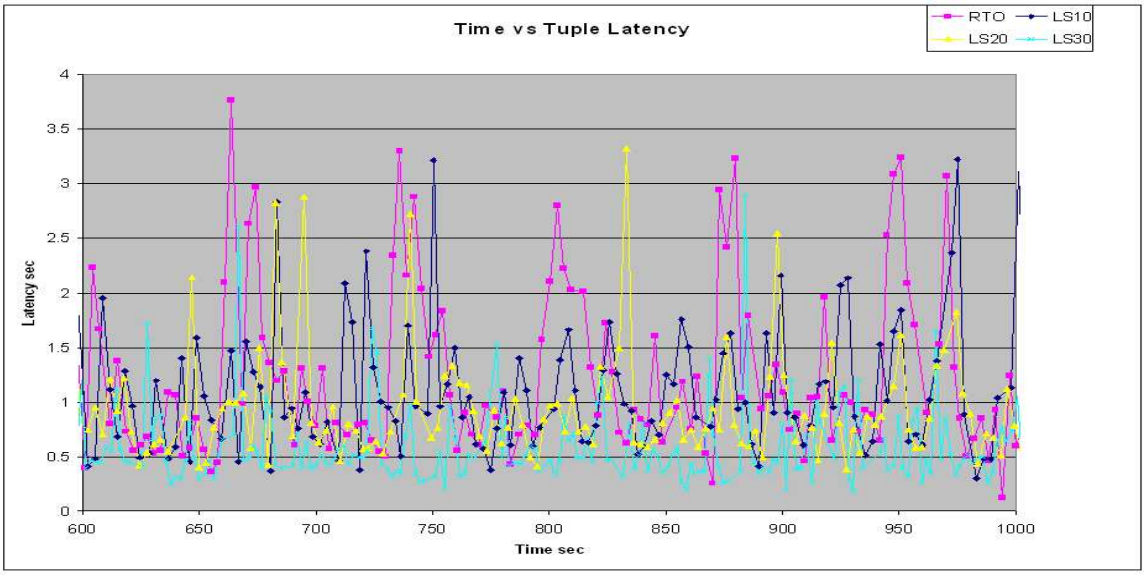

Figure 6.8 Effect of Load Shedding on Tuple Latency

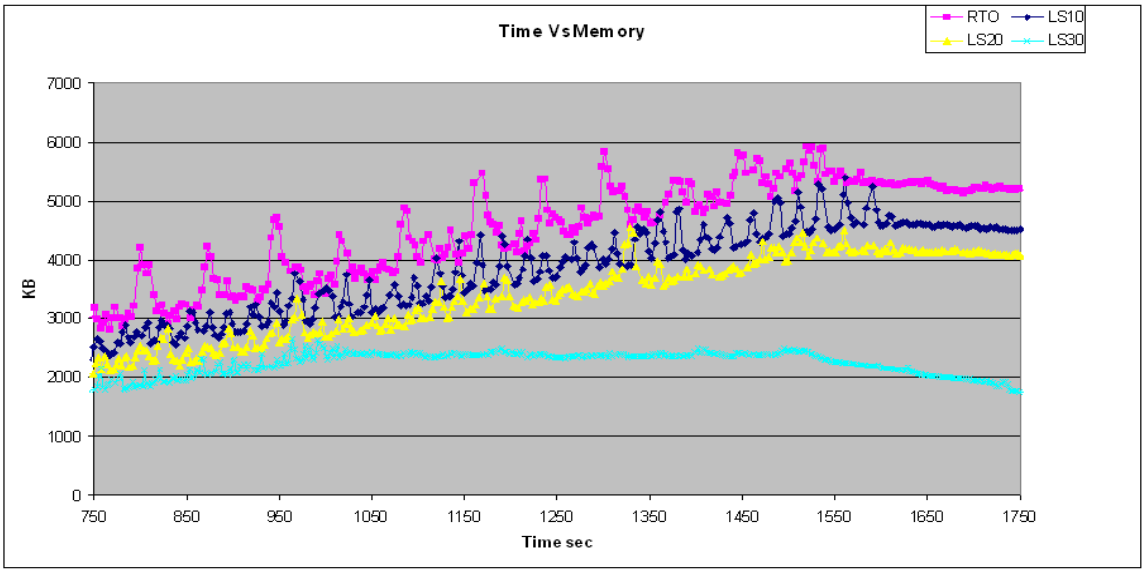

Figure 6.9 Effect of Load Shedding on Memory Utilization

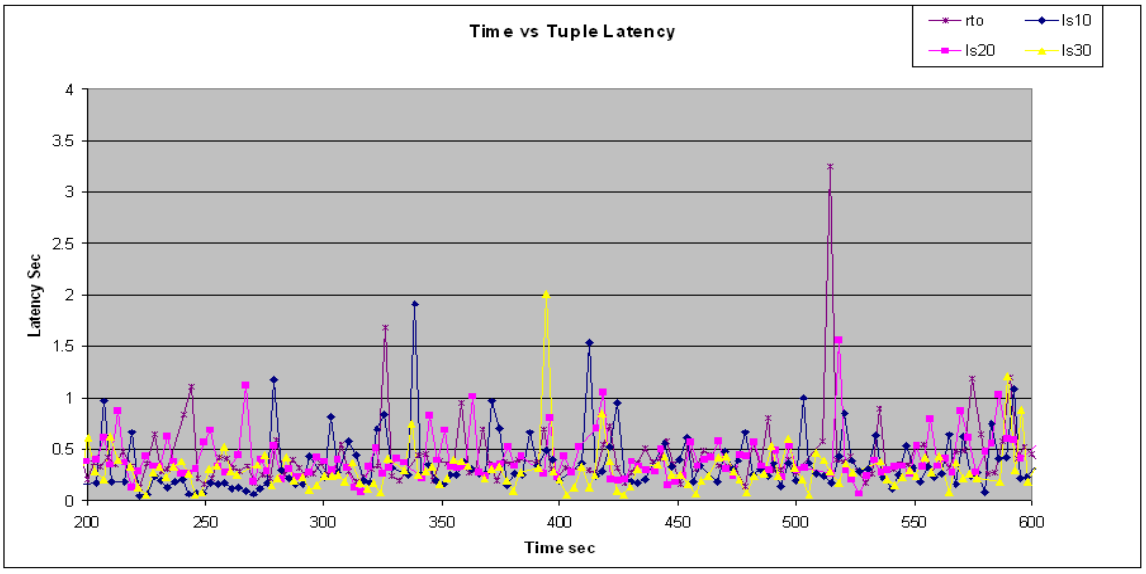

Figure 6.10 Effect of Load Shedding on Tuple Latency

activating load shedders. As seen from the Fig.**??**, higher the error tolerance limit, lower is the tuple latency. This is because more tuples are dropped from the system and hence the system has less number of tuples to process. Also as the load shedders are part of the input queues of operators tuples are dropped immediately and hence they provide lower memory utilization as can be seen from Fig.**??**.

The same experiment was repeated with a tuple based window of 500 tuples. The mean rates for poisson distribution was set to 1100, 1450 and 1000 tuples/sec. The mean was double at different points in time to simulate bursty nature of stream. The tuple latency expected was kept at 15 seconds for the first ten seconds and was reduced to 1 second in the next interval. The initial strategy chosen was segment. Memory utilization was set to Don't Care priority class. As can been from the Fig.**??** higher error tolerance limits allows a query to provide lower tuple latency. The runtime optimizer does not change scheduling strategy for the initial time period when expected value for tuple latency is high. When tuple latency starts getting violated it first changes strategy to

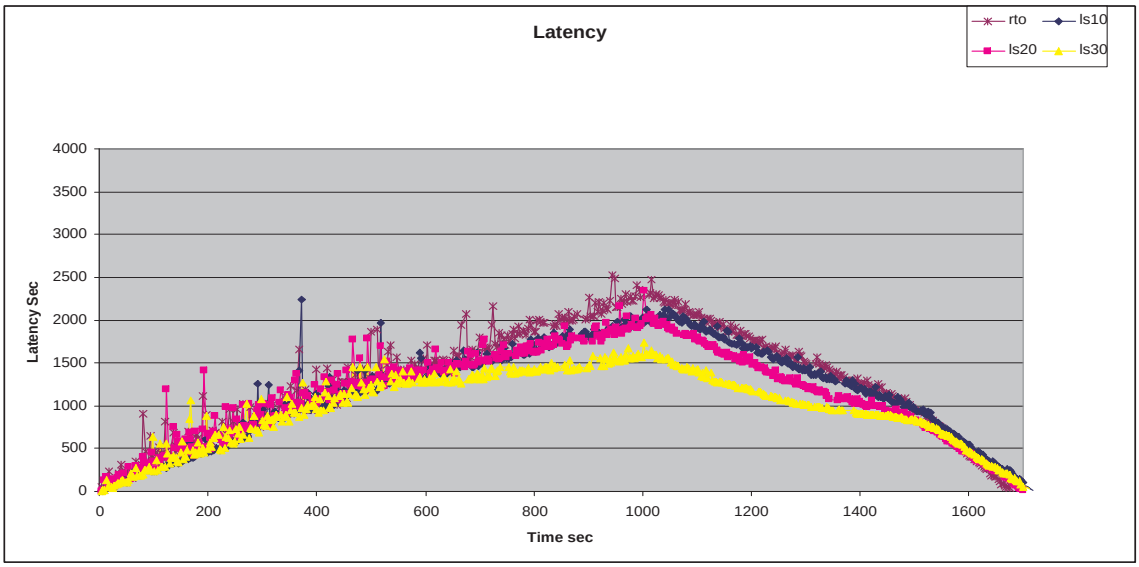

Figure 6.11 Effect of Load Shedding on Memory Utilization

improve latency. Since the expected QoS is still not met, load shedders are activated. As more tuples are shed latency and memory utilization decreases as shown in Fig.**??** and Fig.**??**. The gains achieved are comparatively less than that of the previous experiment as the effect of dropping a tuple is less pronounced. This is because when the window size decreases a single tuple combines with less tuples in a join. With larger window size a single tuple has higher probability of joining with more tuple and therefore shows higher gains.

6.4 Effect of Load Shedding on Error in Results

This experiment was conducted to observe the error introduced by load shedders in the results. The query used consisted of two operators select and an aggregate. The data set used was a modified version of linear road benchmark [**?**] data set. The input rate of the stream followed a poisson distribution with the mean set to 2000 tuples/sec. The mean was double at different points in time. A time based window of 20 seconds

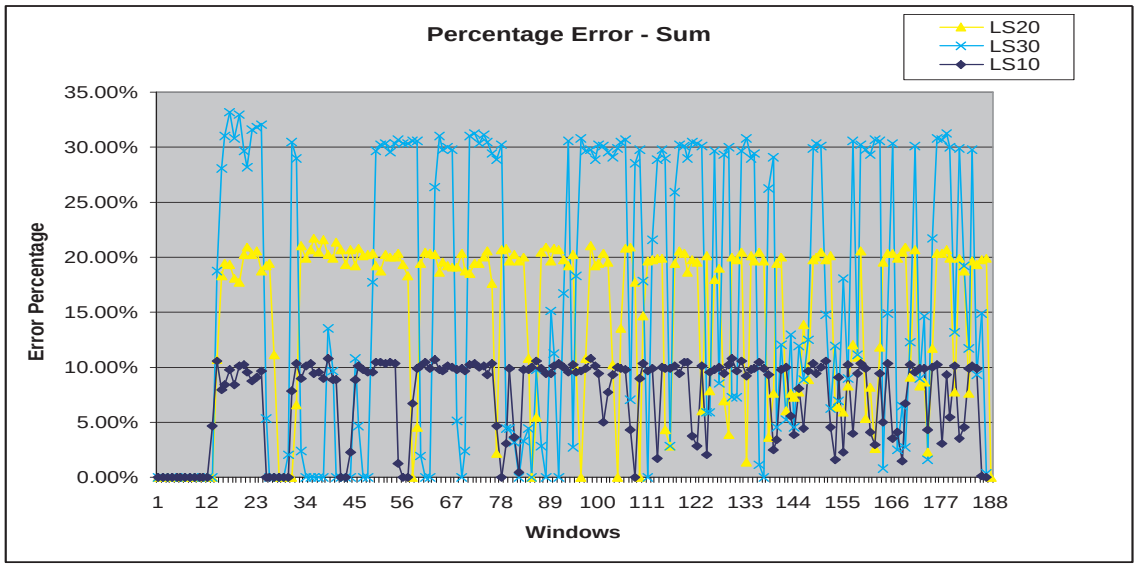

Figure 6.12 Random Shedders: Error in sum

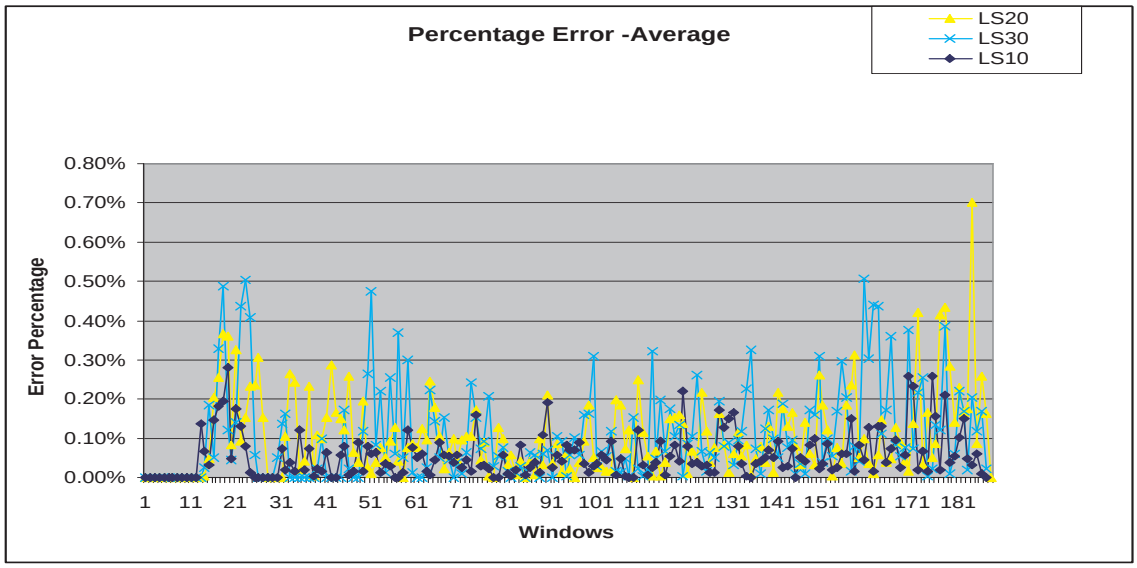

Figure 6.13 Random Shedders: Error in Average

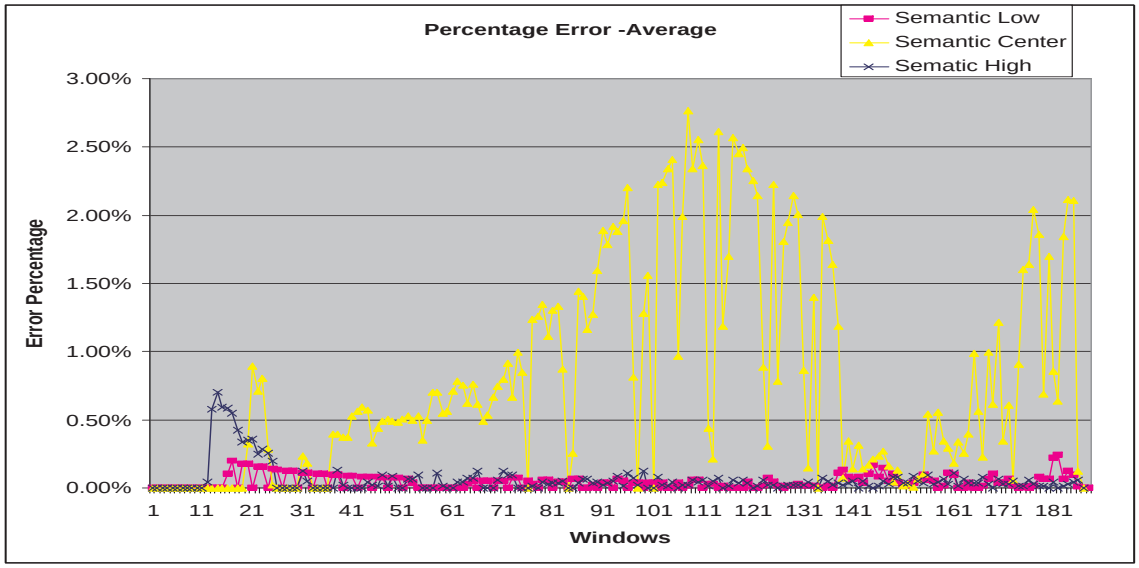

Figure 6.14 Semantic Shedders: Error in Average

was used for this experiment. The sum and the average speed of cars was calculated for each window without shedding and with various error tolerance levels for random load shedding. The error tolerance limits translate to maximum allowed drop probability for random shedders. Hence the error in the sum over each window is near to the error tolerant limit as shown in the Fig.**??**. The error introduced in the average speed of car was much lower than the tolerant limits as indicated in Fig.**??**.

The same experiment was carried out for the three variants of semantic shedder. As expected the semantic shedders introduce lower error as compared to that of random load shedders. The center first shedder introduces a higher error as the values at center affect the results most by dropping more tuple tuples near mean. The lowest and the highest first shedder introduce very small error as shown in Fig.**??** and Fig.**??**. As the query used in semantic shedders calculate the average and sum over each the impact of highest and lowest first shedders are minimal due to outliers in the data. A few outliers in the data can affect the range of tuples dropped and hence they show little impact. The

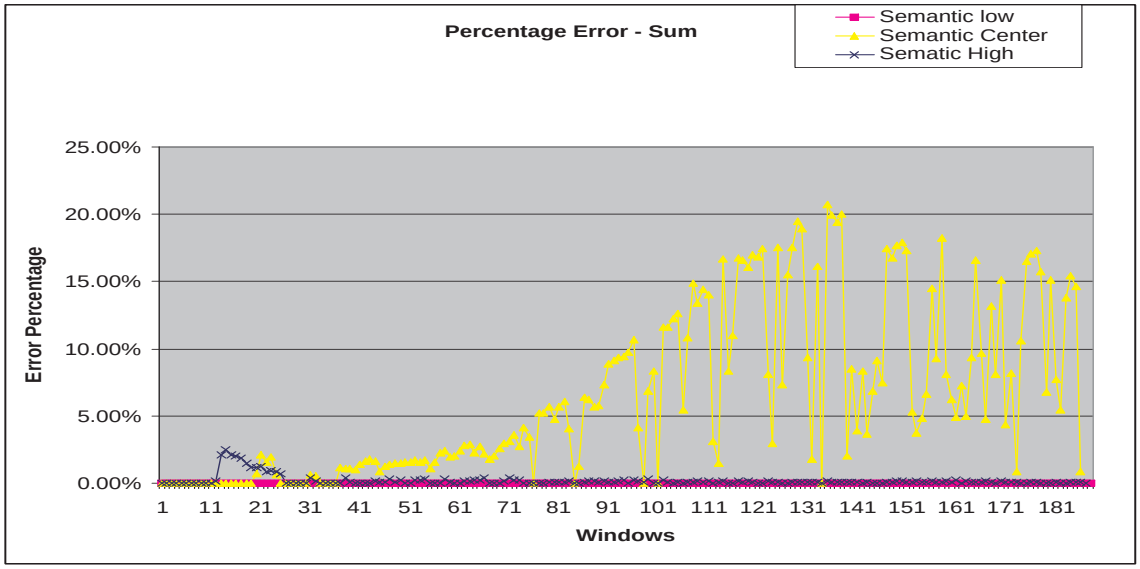

Figure 6.15 Semantic Shedders: Error in Sum

center first shedder drops tuple around the mean and therefore has the highest impact on the results. As most of the tuples fall in the range of the mean more tuples are dropped. The results emphasize the importance of knowledge about the distribution of data while using semantic shedders.

6.5 Summary

In this chapter we have discussed the various experiments conducted. We have shown experimentally, the benefits of runtime optimizer and load shedding on QoS measures. Further we have also shown that, the overhead introduced by runtime optimizer and load shedders are negligible and error introduced by load shedders remain within the tolerance limits specified.

CHAPTER 7

CONCLUSION AND FUTURE WORK

In this thesis we have have designed and implemented a runtime optimizer that selects the best scheduling strategy for a query based on the performance characteristics of scheduling strategies and the monitored values of the QoS measures at runtime. 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 number of switches and a way to avoid cycling between strategies were also developed.

The load shedders have been incorporated as function of buffers (or input queues) to minimize the overhead. Placement of shedders was done using place weights that were calculated using equations based on input rate. We have implemented both random and semantic load shedders for numeric attributes which require little knowledge about characteristics of data. The experiments show that runtime optimizer and load shedders do not add any noticeable overhead and significant gains can be made by using them.

The MavStream server has been extended to run queries using multiple scheduling strategies and change the strategy of a query during runtime using the Master Scheduler. Master scheduler and various scheduling strategies implemented in MavStream form a two level scheduling scheme where in the first level allocates time quantum to schedulers and the second level allocates time quantum to operators, paths or segments. The query plan object has also been extended to accept QoS measures of interest and error tolerance limits from user specifications.

Streaming applications are closely related to situation monitoring applications. This has encouraged us to integrate stream and event processing so as to detect complex events on data streams. Other future work includes finding techniques for query optimization, as stream processing system unlike DBMS do not have clear cut techniques for estimating costs. Handling disorder in the stream and means to tolerate lost and stale data is another area where more work needs to be done.

REFERENCES

[1] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. B. Zdonik, "Monitoring streams - a new class of data management applications." in *VLDB*, 2002, pp. 215–226.

[2] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, "Models and issues in data stream systems." in *PODS*, 2002, pp. 1–16.

[3] S. Chandrasekaran and M. J. Franklin, "Streaming queries over streaming data." in *VLDB*, 2002, pp. 203–214.

[4] H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, E. F. Galvez, J. Salz, M. Stonebraker, N. Tatbul, R. Tibbetts, and S. B. Zdonik, "Retrospective on aurora." *VLDB J.*, vol. 13, no. 4, pp. 370–383, 2004.

[5] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, F. Reiss, and M. A. Shah, "Telegraphcq: Continuous dataflow processing." in *SIGMOD Conference*, 2003, p. 668.

[6] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. B. Zdonik, "Aurora: a new model and architecture for data stream management." *VLDB J.*, vol. 12, no. 2, pp. 120–139, 2003.

[7] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang, "Niagaracq: A scalable continuous query system for internet databases." in *SIGMOD Conference*, 2000, pp. 379–390.

[8] S. Sonune, "Design and implementation of windowed operators and scheduler for stream data," Master's thesis, University of Texas at Arlington, Arlington, 2002.

[9] A. Gilani, S. Sonune, B. Kendai, and S. Chakravarthy, "The anatomy of a stream processing system." in *BNCOD*, 2006, pp. 232–239.

[10] A. Gilani, "Design and implementation of stream operators, query instantiator and stream buffer manager," Master's thesis, University of Texas at Arlington, Arlington, 2002.

[11] S. Babu, U. Srivastava, and J. Widom, "Exploiting -constraints to reduce memory overhead in continuous queries over data streams." *ACM Trans. Database Syst.*, vol. 29, no. 3, pp. 545–580, 2004.

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

[13] S. Chakravarthy and V. Pajjuri, "Scheduling strategies and their evaluation in a data stream management system." in *BNCOD*, 2006, pp. 220–231.

[14] Q. Jiang and S. Chakravarthy, "Scheduling strategies for processing continuous queries over streams." in *BNCOD*, 2004, pp. 16–30.

[15] D. Carney, U. Çetintemel, A. Rasin, S. B. Zdonik, M. Cherniack, and M. Stonebraker, "Operator scheduling in a data stream manager." in *VLDB*, 2003, pp. 838–849.

[16] B. Babcock, S. Babu, M. Datar, R. Motwani, and D. Thomas, "Operator scheduling in data stream systems." *VLDB J.*, vol. 13, no. 4, pp. 333–353, 2004.

[17] A. Arasu and J. Widom, "Resource sharing in continuous sliding-window aggregates." in *VLDB*, 2004, pp. 336–347.

[18] U. Srivastava and J. Widom, "Memory-limited execution of windowed stream joins." in *VLDB*, 2004, pp. 324–335.

[19] N. Tatbul, U. Çetintemel, S. B. Zdonik, M. Cherniack, and M. Stonebraker, "Load shedding in a data stream manager." in *VLDB*, 2003, pp. 309–320.

[20] B. Babcock, M. Datar, and R. Motwani, "Load shedding for aggregation queries over data streams." in *ICDE*, 2004, pp. 350–361.

[21] Q. Jiang and S. Chakravarthy, "Load shedding in a data stream management system," *TR CSE-2003, UT Arlington*, Nov 2003.

[22] S. Viglas and J. F. Naughton, "Rate-based query optimization for streaming information sources." in *SIGMOD Conference*, 2002, pp. 37–48.

[23] D. Maier, J. Li, P. A. Tucker, K. Tufte, and V. Papadimos, "Semantics of data streams and operators." in *ICDT*, 2005, pp. 37–52.

[24] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker, "No pane, no gain: efficient evaluation of sliding-window aggregates over data streams." *SIGMOD Record*, vol. 34, no. 1, pp. 39–44, 2005.

[25] P. Bonnet, J. Gehrke, and P. Seshadri, "Towards sensor database systems." in *Mobile Data Management*, 2001, pp. 3–14.

[26] N. Trigoni, Y. Yao, A. J. Demers, J. Gehrke, and R. Rajaraman, "Multi-query optimization for sensor networks." in *DCOSS*, 2005, pp. 307–321.

[27] S. Madden and M. J. Franklin, "Fjording the stream: An architecture for queries over streaming sensor data." in *ICDE*, 2002, pp. 555–566.

[28] R. Avnur and J. M. Hellerstein, "Eddies: Continuously adaptive query processing," in *SIGMOD Conference*, 2000, pp. 261–272.

[29] A. Deshpande and J. M. Hellerstein, "Lifting the burden of history from adaptive query processing." in *VLDB*, 2004, pp. 948–959.

[30] S. Madden, M. A. Shah, J. M. Hellerstein, and V. Raman, "Continuously adaptive continuous queries over streams." in *SIGMOD Conference*, 2002, pp. 49–60.

[31] S. Chandrasekaran and M. J. Franklin, "Psoup: a system for streaming queries over streaming data." *VLDB J.*, vol. 12, no. 2, pp. 140–156, 2003.

[32] A. Arasu, S. Babu, and J. Widom, "The cql continuous query language: semantic foundations and query execution." *VLDB J.*, vol. 15, no. 2, pp. 121–142, 2006.

[33] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. B. Zdonik, "The design of the borealis stream processing engine." in *CIDR*, 2005, pp. 277–289.

BIOGRAPHICAL STATEMENT

Balakumar K. Kendai was born in Madurai, India. He received his Bachelors degree in Electrical and Electronics Engineering from Bharathiar University, India, in May 2000. After a brief tenure in industry he started his graduate studies in the Spring of 2004 at The University of Texas at Arlington. He is a member of Tau Beta Pi. He received his Masters in Computer Science and Engineering in December 2006.