DESIGN AND IMPLEMENTATION OF

SCHEDULING STRATEGIES

AND THEIR EVALUAION

IN MAVSTREAM

The members of the Committee approve the master's thesis of Vamshi Krishna Pajjuri

Sharma Chakravarthy
Supervising Professor _____

Jung-Hwan Oh _____

Gautam Das _____

DESIGN AND IMPLEMENTATION OF SCHEDULING STRATEGIES AND THEIR EVALUAION IN MAVSTREAM

by

VAMSHI KRISHNA PAJJURI

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 2004

ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. Sharma Chakravarthy, for his great guidance and support, and for giving me an opportunity to work on this thesis. I am also thankful to Dr. Gautam Das and Dr. Jung-Hwan Oh for serving on my committee.

I would like to thank Dustin and Vihang Garg for their constant support and encouragement throughout this thesis providing valuable tips during the implementation of this project. Thanks are due to my seniors Altaf Gilani and Satyajeet Sonune for their fruitful discussions and guidance. I would like to thank my roommates Kiran Shetty and Sreepraveen Veeramachaneni for their help. I would like to thank all my friends in the ITLAB for their support and encouragement.

I would like to acknowledge the support by the Office of Naval Research and the NSF (grants ITR 0121297 and IIS-0326505) for this research work.

I am thankful to my parents for their constant support and encouragement throughout my academic career without which I would not have reached this position.

November 18, 2004

ABSTRACT

DESIGN AND IMPLEMENTATION OF

SCHEDULING STRATEGIES

AND THEIR EVALUAION

IN MAVSTREAM

Publication No. _____

Vamshi Krishna Pajjuri, MS

The University of Texas at Arlington, 2004

Supervising Professor: Dr. Sharma Chakravarthy

The limitations of the traditional Database Management System to process continuous, unbounded and real time data streams for applications such as tickers, network monitoring, traffic management, and sensor monitoring have necessitated the development of MavStream. It is a Data Stream Management System (DSMS) developed for processing stream data with bursty arrival rates and need for real-time performance requirements.

This thesis investigates the design and implementation of scheduling strategies for stream processing. The scheduling strategies implemented in this thesis minimize the

tuple latency enabling the system to satisfy quality of service (QoS) requirements. Several scheduling strategies have been integrated into the architecture of MavStream. We first introduce the path capacity strategy with the goal of minimization of the tuple latency. As an improvement, a segment scheduling strategy and its simplified version are implemented which aim at minimization of total memory requirement and throughput.

Extensive set of experiments have been designed and executed to evaluate the effectiveness of the proposed scheduling strategies. In addition, a feeder module was developed to simulate real time streams. Extensive experiments have been performed to measure tuple latency, memory usage and their effect on varying data rate and window widths on input streams with varying characteristics.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

Figure Page

CHAPTER 1

INTRODUCTION

Traditional Database Management Systems (DBMS) work on a passive repository and evaluate one-time queries. For newer sensor and other applications which have data coming in the form of streams as input and have requirements that have to satisfy realtime and quality of service (QoS) characteristics, traditional DBMSs are not adequate and hence we need a different architecture termed the Data Stream Management System (DSMS). The DSMS processes continuous stream of input data elements without storing the data in a repository and outputs results that satisfy user-specified realtime requirements. Typically, in stream processing, once the data element is processed it can not be retrieved again without storing it. The size of the data stream is also unbounded and can be thought of as a relation with infinite tuples. In DSMS, continuous queries include blocking operators such as join and aggregate which are implemented using a window concept as the data stream is unbounded.

It is important to understand the characteristics of streaming data and queries over stream data in order to develop an understanding of the DSMS. Streaming data displays the following characteristics [1] :

1. Streaming data is not stored in tables as they arrive continuously and online.
2. Streams have a predefined definition and tuples of data have an ordering (either based on timestamp or of an attribute value).

3. Streaming data are potentially unbounded in size. They are continuously generated by sensor class of devices. In MavStream the data stream is generated by sensors used as part of the MavHome project.
4. When input rate is bursty, streaming data may be lost, garbled or may be intentionally omitted for processing. It is sometimes necessary to reduce load by dropping less important tuples to satisfy QoS requirements. Sampling [2] is a common technique used to handle bursty input rates.
5. Data streams may 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.

Continuous Queries processed on streaming data can be broadly classified into: Predefined 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. Hence queries referring to past information are difficult to evaluate unless the system supports storage of past information. Since Ad-Hoc queries are not known beforehand, query optimization, finding common sub-expressions, etc., adds processing delay and hence are typically ignored or done in a different way.

Predefined and Ad-Hoc [3, 4] Queries are further classified into: One-Time queries or snap-shot queries and Continuous queries:

i. One-Time Queries: These queries are evaluated only once over a given window. Once the query is evaluated, it is removed from the system. It generates output only once at the end of the window.

ii. Continuous queries: These queries are evaluated continuously as data streams arrive. The results are produced incrementally and continuously at the end of every new window. Most queries in streaming applications are continuous. Results may be stored or updated as streaming data arrives or the output may itself be streamed.

The primary differences between DBMSs and DSMSs are:

i. DBMS stores finite, persistent data sets whereas in DSMS the data, received from sensors, is generally unbounded and not stored. Sometimes stream data may be stored to handle bursty traffic or to perform recovery in case of a crash.

ii. DBMS assumes one time queries against data whereas in DSMS deals with continuous queries which are evaluated as stream data arrives. The results are produced over long periods of time.

iii. The Quality of Service (QoS) requirement is not provided by a DBMS where as it is necessary for a DSMS.

iv. In DBMS data is pulled between the operators, whereas in DSMS data is pushed from the leaf node to the root node.

v. The optimizer in a DBMS works on pre-defined queries, whereas optimizer is driven by the Quality of Service (QoS) requirements in DSMS

vi. The results of DBMS queries are accurate whereas in DSMS results are approximate.

vii. In DSMS multiple passes could not be made as the data is not stored in relations.

viii. In a DBMS data is accessed randomly, whereas in a DSMS data is accessed sequentially.

ix. As the operators in DBMS work on the data which is already present, there is no need for a dedicated scheduler, whereas scheduling of operators is required in DSMS.

The areas of use for a DSMS are stream-based applications such as security [5], telecommunications data, network management, link failure in networks and weather monitoring etc. MavHome [6] is a smart home environment that perceives its environment through the use of sensors and acts upon the environment through the use of actuators. MavHome uses MavStream processing system for monitoring of the data gathered from the streams. The current state of the system allows the user to give the query through a user interface. The queries are run using the data collected from the sensors.

This thesis mainly concentrates on scheduling alternatives to support the predefined QoS requirements for a continuous query. Average Tuple Latency and Maximum Memory Utilization are the two QoS requirements addressed in this thesis. The time taken to process a tuple is termed as the tuple latency of that tuple, which need to be minimized for many applications. This could significantly be affected by the way the query is scheduled. In this thesis the scheduling strategies are extended and then a comparison is made to evaluate their effectiveness. The main memory is also an area of

concern. As the size of the main memory and the CPU speed is limited and DSMS requires real time results, optimization of memory usage is important for DSMSs.

A single scheduling strategy would not be able to satisfy all of above properties, as there are tradeoff's among the performance metrics and the usage of limited resources. So we have introduced several scheduling strategies [7].

1. Path Capacity scheduling to achieve minimum tuple latency.
2. The segment scheduling strategy to achieve the minimal memory requirements.
3. The simplified segment strategy, which requires slightly more memory but gives better tuple latency than the segment strategy.

The above summarizes why traditional DBMSs are not suitable for streaming applications. It also explains the need for performance improvement in MavStream and suggests some ways to improve it. MavStream is specifically designed for rapid and continuous loading of individual data items and to directly support continuous queries that are typical of continuous data stream applications.

The rest of the thesis is organized as follows. In chapter 2 we review recent projects on data stream processing, as well as discuss some of the scheduling strategies used in other stream processing systems such as Aurora, STREAM, Eddies. In chapter 3 we describe the architecture of MavStream. Some of the important modules highlighted in this architecture are Scheduler, Run-Time Optimizer, Operators, Buffer Manager, Feeder and the Alternate Plan Generator. In chapter 4 we discuss the design of the scheduling strategies presented in MavStream and their limitations. In chapter 5 we discuss the implementation details and emphasize on implementation problems. In chapter 6, the experiments and results are given to evaluate the difference between earlier scheduling

strategies and the ones presented in this thesis for improving QoS. Finally, chapter 7 has conclusions along with an overview of the contributions and a summary of directions for future work.

CHAPTER 2

RELATED WORK

This chapter discusses the related work which addresses various aspects of streaming applications and focuses on the scheduling strategies proposed in the literature to handle QoS needs of applications that handle data streams.

2.1 Aurora

Aurora [5] is a data flow system that uses the primitive box and arrow representation. Tuples flows from source to destination through the operational boxes. It can support continuous queries, ad-hoc queries and views at the same time. QoS is associated with the output. QoS determines how resources are allocated to the processing elements along the path of query operation. It has a connection point where boxes can be added or removed from the network.

Scheduler deals with operators. It is designed to cater to the needs of large-scale systems with real time response requirements. A key component of Aurora is the scheduler that controls processor allocation. The scheduler is responsible for multiplexing the processor usage to multiple queries according to application-level performance. 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 Two-level [8] 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 has a Storage Management module, which takes care of storing all the required tuples for its operation which is also responsible for queue management.

2.2 STREAMS

STREAMS (*STanford StREam DatA Manager*) [3] is a prototype implementation of a complete Data Stream Management System being developed at Stanford. They use a modified version of SQL for query interface. It supports logical and physical windows. Logical windows are expressed in terms of tuples and physical windows are expressed in terms of timestamp. It supports continuous queries but has not addressed the issue of ad-hoc queries. The system generates a query execution plan upon the registration of a query that is run continuously. System memory is distributed dynamically among the queues in query plans along with buffers handling the streams coming over the network.

STREAMS has a central scheduler that has the responsibility for scheduling operators. The scheduler dynamically determines the time quantum of execution for each operator. Period of execution may be based on time, or on the number of tuples consumed or produced. Different scheduling policies are being experimented. In our system the time quantum assigned to the operator is a static value taken from the configuration file, however it depends on the tuples present in the input queue of the operator.

STREAMS uses the chain scheduling strategy [9] 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. In our system we developed round-robin scheduling to avoid starvation and path capacity scheduling to get optimal tuple latency.

2.3 Rate Based Optimization

In Rate Based Optimization [10] the fundamental statistics used are estimated from the rate of the streams in the query evaluation tree. They estimate the output rates of various operators as a function of the rates of their input and use these estimates to optimize queries. There are two important optimization opportunities, one to optimize for a specific time point in the execution process where they estimate how many output elements the plan (query tree) will have produced by that time. The second one is to optimize for output production size, where given an output size of N, they identify the plan that will reach the specified number of results at the earliest time.

The Rate Based Optimization aims at maximizing the throughput of the query. They use integrals to compute the estimate which is inefficient for practical optimization

purposes as there will be large number of plans and integrating for each plan is costly. They do not consider tuple latency and memory requirement.

2.4 Eddies

Eddies [11] supports dynamic re-optimization in decentralized dataflow query processing. Instead of determining a static optimal global query plan before execution based on knowledge of costs of each operation, it relies on simple dynamic local decisions to get a good global strategy. This is useful when cost information is hard to obtain or data characteristics are continuously changing.

They identify “moments of symmetry” during which operators can be easily reordered when they are subjected to changes in cost. The concept of “Synchronization Barrier” regarding join algorithms provides insights into the possibility of re-optimization of different algorithms. It allows the system to adapt dynamically to fluctuations in computing resources. 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.

Eddy module directs the flow of tuples from the inputs through the various operators to the output, providing the flexibility to allow each tuple to be routed individually through the operators. The routing policy used in Eddy determines the efficiency of the system. Eddy’s tuple buffer is implemented as a priority queue with a flexible prioritization scheme. An operator is always given the highest-priority tuple in the buffer that has the corresponding ready bit set. In a simple priority scheme, tuples enter Eddy with low priority, and when they are returned to Eddy from an operator they

are given high priority which ensures that tuples flow completely through the system before new tuples are consumed from the inputs, ensuring that Eddy does not become “clogged” with new tuples. They use a router to schedule that continuously monitors the system status. This system does not scale as there is a lot of status information associated with the each tuple.

CHAPTER 3

MAVSTREAM ARCHITECTURE

MavStream is being developed for processing continuous queries over streams. MavStream is modeled as a client-server architecture in which client accepts input from the user, transforms it into a form understood by the server and sends the processed input to the server using predefined protocols. 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 application. The various components of MavStream are shown in Figure 3.1

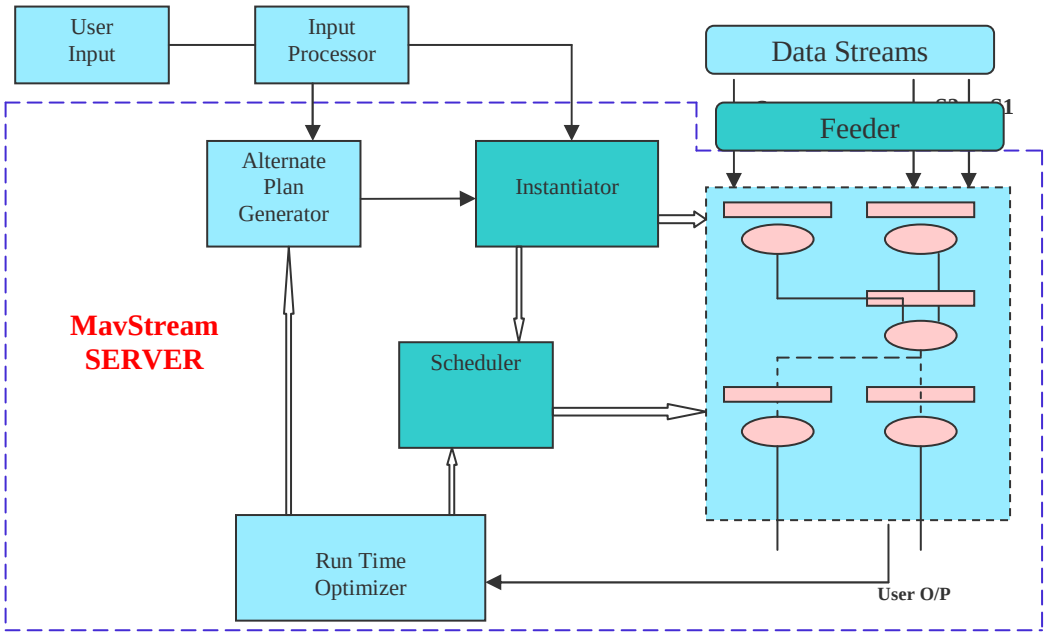


Figure 3.1 MavStream Architecture

The MavStream server upon receiving the query from the client, instantiates the query and one of the outputs of the query is given to the application and the other to the

Run-Time Optimizer. Run-Time Optimizer frequently measures the QoS requirements for the query. If the QoS Specifications are satisfied then the query is further evaluated. Else it can change the scheduling policy or generate an alternate plan to satisfy the QoS specifications.

Following sections provide a brief overview of various modules constituting the MavStream system:

3.1 MavStream Client

The web-based client provides a graphical user interface to pose queries to the system. It constructs a plan object that represents a single complete query that includes QoS requirements from user specifications. The client also generates intermediate schema when base schema is altered by operators such as project or join which shrinks and expands schema, respectively. Once the input is processed completely, all needed information is generated and is sent to the server over a defined set of protocols. 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 Instantiator

Instantiator has the responsibility of initializing and instantiating streaming operators and their associated buffers on accepting user queries from the client. Client constructs a plan object, which is a sequence of operator nodes where every node describes an operator completely. This operator hierarchy defines the direction of data flow starting from leaves to the root. Instantiator traverses the query tree in a bottom-up

manner to ensure that child operators are instantiated prior to parent operators that is required, to respect query semantics as data flows from leaves to root. 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.

3.3 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 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 streams to the client.
- Stop a query.

3.4 Alternate Plan Generator (APG)

The query submitted by the user is converted into a query plan object by the client. This plan is an executable plan and may not be the optimal plan. A plan object is an ordered tree indicating the flow of data and instantiation order. Many alternate plans may be possible which are equivalent to the previous plan but is more efficient than the one generated by the client. Consider a plan object in which a Join is performed prior to Select where Select has low selectivity. Here it would have been more appropriate to execute Select prior to Join, which would have improved resource utilization. In order to generate these alternate plans, an Alternate Plan Generator is needed. This module takes the plan given by the user as input and gives alternate plans which are more efficient than the input plan. All the queries running in the system are merged [12] as a global plan object combining their operators to save computations and memory.

Run-Time Optimizer will make use of alternate plan generator in order to dynamically select an alternate plan when the result produced by the current plan does not satisfy the QoS requirements. The best alternate plan (locally optimal) of a query tree may not be the optimal with respect to a global plan. An alternate plan is considered the best when most of its operators are merged with the existing global plan. That plan if considered alone without a global plan may not be the most favorable plan.

3.5 Run-Time Optimizer (RTO)

Run time optimizer uses alternate plan generator in order to dynamically select an alternate plan when the result produced by the previous plan does not satisfy the quality

of service requirements. It aims at decreasing the tuple latency, minimizing the memory utilized by the system and maximizing the output rate of query evaluation plans. In addition, to ensure QoS, optimizer may ask the scheduler to change the scheduling policy or increase the priority of the operators or assign different time quantum for specific operators which need more time. The runtime optimizer may use the alternate plan generator to generate a different plan that is better with the global plan running in the system. Optimizer continuously monitors the output and compares it with the QoS requirement. Run time optimizer is expected to use all the parameters intelligently to improve QoS. It is supposed to continuously monitor the output and compare with the QoS requirement. It may also monitor selectivities to determine whether to generate new paths for scheduling. It may also monitor input streams to determine bursty periods and enable load shedding, another measure for preserving the QoS.

3.6 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 likely to be 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 scheduler 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 be scheduled. Operator goes through a number of states while it is being scheduled.

Following are the scheduling policies implemented in MavStream:

1. Round-Robin: In this strategy, all the operators are assigned the same priority (time quantum). Scheduling order is decided by the ready queue. This policy is not likely to dynamically adapt to quality of service requirements as all operators have the same priority.

2. Weighted round-robin: Here different time quanta are assigned to different operators based on their requirements. Operators are scheduled in a round robin manner, but some operators may get more time quantum over others. For example operators at leaf nodes can be given more 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: Schedule 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 breaking operator path into many segments, we break an operator path into only two segments. This

strategy takes slightly more memory than the segment strategy giving improvement in tuple latency.

3.7 Feeder

A feeder module is used 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 on the stream. The feeder is implemented as a single thread. All the tuples from the stream sources are put in one big buffer called the feeder buffer. Feeder thread reads the tuples from the feeder buffer and based on the split condition, splits into different streams and 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 data collected from MavHome and feed tuples to leaf operators simulating a real time environment.

3.8 Operators & Buffer Management

Operators of traditional DBMSs are designed to work on the data that is already present and cannot produce real-time response to queries over high volume, continuous, and time varying data streams. The processing requirements of real time data streams are different from traditional applications. The operators for DSMSs are designed to handle long running queries to produce 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 window concept so as to overcome the blocking nature of operators. During the life span of operator, it can either

be in ready to run, running, suspended or stop state. The objective of buffer management is to provide an illusion to the user that it has an infinite memory at its disposal. It is easy to provide an infinite main memory buffer if the system has a huge amount of main 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 need to be stored in secondary storage buffers. An interface is provided to store tuples in the buffer and retrieve the stored tuples from the buffer. The information that the tuple is stored in or retrieved from main memory or secondary memory should be transparent from an operator's viewpoint.

3.9 Chapter Summary

In this chapter we discussed about the MavStream architecture. Various components such as instantiator, server, run-time optimizer, scheduler, alternate plan generator and feeder were explained in detail. Interaction between these components was also covered.

CHAPTER 4

DESIGN

4.1 Scheduler

Most of the operating systems provide a coarse control over thread scheduling and is not useful when scheduling needs some prioritization derived from application or query semantics. Hence, for scheduling stream queries, we need to have a separate scheduler that can be tailored to the needs of the DSMS. Granularity of scheduling can be at different levels:

i. Tuple based scheduling: It is not feasible to schedule at a tuple level as the total number of tuples in the system is unbounded and context switching is likely to be very expensive.

ii. Query based scheduling: When we schedule the query as one unit where all the operators have the same priority, it may be acceptable but not good for merging operators that have same input streams and the same condition. At query level, optimizations are difficult since entire query overlap is difficult to achieve.

iii. Operator based scheduling: Scheduling at operator level is a good compromise where each operator of the query can have a different priority. This gives a better control than the above two extremes of the spectrum and reduces the overhead of scheduling and improving the QoS characteristics of the results.

Hence, in MavStream, we do operator level scheduling which has a granularity that is in between tuples and queries.

Some of the Quality of Service (QoS) specifications are average tuple latency, system throughput and maximum memory utilization. In order to meet the predefined QoS requirements, a scheduler is needed. Design of a scheduler and the scheduling strategies used have a significant impact on meeting the QoS specifications. A good scheduling strategy should be aware of unexpected overload situation and be implementable easily with low scheduling overhead. Unfortunately, all of the QoS requirements listed above cannot be satisfied by a single scheduling scheme. Hence, in this thesis we have implemented three scheduling strategies to deal with different QoS metrics:

i. Path capacity strategy to achieve best tuple latency.
ii. Segment scheduling strategy to achieve minimum memory requirement.
iii. Simplified segment strategy that takes slightly more memory than the segment strategy but better tuple latency than the segment strategy.

MavStream supports non-windowed operators such as select, split, project and windowed operators such as aggregate and join. User query is made up of these operators where the tuples are pushed from the bottom node (leaf node) to the root. The path from the leaf node to the root node is called an operator path. The number of operator paths is equal to the number of the leaf nodes.

The following notations are used in the description of scheduling strategies.

i. Operator selectivity σ_i: The ratio of the number of tuples going out to the number of tuples coming in.

ii. Operator processing capacity C^{P}_{Oi} : The number of tuples that can be processed in one time unit at operator O_i.

iii. Operator memory release capacity C^{M}_{Oi} : The number of memory units that can be released within one time unit by operator O_i.

iv. Operator path processing capacity C^{P}_{Pi} : The number of tuples that can be processed within one time unit by operator path P_i.

v. Operator service time ($1/ C^{P}_{Oi}$) : The number of time units needed to process one tuple at this operator O_i.

vi. Path memory release capacity C^{M}_{Pi} : The number of memory units released within one time unit by the operator path P_i.

In the following sections we will discuss the current scheduling strategies currently available in MavStream, the new scheduling strategies implemented as part of this thesis, and the design of the Ready Queue and associated design issues.

4.1.1 Current scheduling strategies present in MavStream

MavStream supports round-robin and weighted round-robin scheduling strategies based on the time quantum assigned to the operator.

i. Round-robin strategy: In this strategy equal weights are assigned to all operators. As and when the operators are instantiated they are put in the scheduler's ready queue. It is a bottom-up scheduling strategy where the leaf nodes are scheduled first, then the operators above it next with the root operator being scheduled last. Every operator is scheduled for the same time quantum. This kind of strategy avoids starvation but has bursty throughput and higher tuple latency.

ii. Weighted round-robin: In this strategy different operators have different weights. Weights correspond to the time quantum assigned to the operator. Higher the priority of operator, higher the time quantum assigned to it. For example join can be assigned more weight than the select operator; leaf operators can be assigned more weight than non-leaf operators. It avoids starvation and has better tuple latency than the round-robin strategy.

4.1.2 Path capacity scheduling strategy

Round-robin and weighted round-robin does not consider the properties of the operator path such as processing capacity and memory release capacity or selectivity. As they need to schedule all of the operators, scheduling overhead is high. This does not promise good tuple latency which motivates us to develop the path capacity strategy. Before understanding the path capacity scheduling strategy, we should know how to calculate the processing capacity of an operator path. In the coming subsections we discuss on how to calculate the processing capacity of the operator path and path capacity scheduling algorithm.

4.1.2.1 Calculation of processing capacity of operator path

Consider an operator path 'P' with k operators having selectivity σ_i and processing capacity C^P_{Oi} where i=1 to k, with i=1 as the leaf node and i=k as the root node. If we pass 1 tuple from the leaf node the output produced is σ_1 tuples of operator O_1. The time spent by the tuple at leaf node is $1/ C^P_{O1}$, passing the σ_1 tuples to next operator will produce σ_1 σ_2 tuples. The total time spent by the tuple to pass the second node is $1/ C^P_{O1} + \sigma_1/ C^P_{O2.}$ Similarly the total time spent by the tuple to pass the root node is $1/ C^P_{O1} + \sigma_1/ C^P_{O2} + \ldots\ldots + \sigma_{1\ldots.}\sigma_{k-1}./ C^P_{Ok}$. The time taken by one tuple to go

through the leaf node to the root node is the service time of the operator path. Processing capacity of the operator path is the inverse of service time of the operator path.

4.1.2.2 Algorithm for path capacity scheduling strategy

At any time, consider all the operator paths in the system, schedule the operator path with maximum processing capacity as long as there are tuples present in the input buffer of the operator path or there exists another operator path with more processing capacity than the current operator path.

Once the operator path is chosen for scheduling, a bottom-up approach is used to schedule the operators along the operator path.

It is a static scheduling strategy, as the processing capacity of operator completely depends on the system and the selectivity of the operator which can be found by monitoring the output of the operator. As the number of operators along the operator path increases the processing capacity decreases.

4.1.3 Segment scheduling strategy

As main memory is a limited resource in the system, utilization of main memory is as important as the tuple latency. Path capacity strategy buffers all the unprocessed tuples at the beginning of the operator paths and hence the memory utilization by the query increases. Segment strategy is developed with the goal of minimizing the total internal queue size. In the coming subsections we discuss the construction of segments, calculating the memory release capacity and the segment scheduling algorithm

4.1.3.1 Construction of segments from operator path

Segment construction has two main steps. First is partitioning the operator path into segments and the second is to prune the segment list. For each operator path we do

the following. Consider the operator path with k operators O_1, O_2, ..., O_k, with O_1 being the leaf operator and O_k the root operator. Start a segment with operator O_1 as current operator and check if the next operator has higher memory release capacity. If so, add the next operator to the current segment and repeat the same with the newly added operator as the current operator. If the next operator does not have a higher memory release capacity than the current operator then complete the present segment and start a new segment with next operator in it and repeat the same with the next operator as current operator. In the pruning procedure, a new segment is added to the segment list only if:

i. any of its subsets has already been in the list, we have to remove all its subsets from the segment list, and then add the new segment into the list.

ii. none of its supersets has been in the list, we add it to the list; otherwise, the new segment is discarded.

4.1.3.2 Calculation of memory release capacity of a segment

Consider a segment 'S' with k operators having selectivity σ_i and processing capacity C^P_{Oi} where i=1 to k, with i=1 as the leaf node and i=k as the root node. If we pass 1 tuple of size *InputTupleSize* through the leaf node the output produced is σ1 tuples from operator O_1. After passing through all the operators in the segment the number of tuples produced is $\sigma_1\ \sigma_{2....}\sigma_k$, Let us assume that the size of the output tuple is *OutputTupleSize.* Number of memory units released is (*InputTupleSize* - *OutputTupleSize* * $\sigma_1\ \sigma_{2....}\sigma_k$) for one unit of tuple. Segment can process C^P_S tuples in one time unit. Memory release capacity of the segment is C^P_S (*InputTupleSize* - *OutputTupleSize* * $\sigma_1\ \sigma_{2....}\sigma_k$).

4.1.3.3 Algorithm for segment scheduling strategy

At any time, consider all the segments in the system and schedule the segment with maximum memory release capacity as long as there are tuples present in the input buffer of the segment or there exists another segment with more memory release capacity than the current segment. Once the segment is chosen for scheduling, a bottom-up approach is used to schedule the operators along the segment.

It is a static scheduling strategy, as the processing capacity of an operator depends completely on system and the selectivity of the operator which can be found by monitoring the output of the operator.

4.1.4 Simplified segment scheduling strategy

Simplified segment scheduling strategy is a variant of the segment scheduling strategy. It employs a different segment construction algorithm. As the number of segments is not significantly less than the number of operators in the query plan and much of the memory is released by the leaf operators we divide the operator path into at most two segments which are called simplified segments. In the coming subsections we discuss the construction of simplified segments.

4.1.4.1 Construction of simplified segments from operator path

We partition the operator path into at most two segments. The first segment includes the leaf operator and its consecutive operators such that the ratio of the memory release capacity of the next operator to the current operator is more than the constant factor γ which is less than one to reduce the number of segments. All the other operators along the operator path will form the second segment. In the previous segment construction algorithm, $\gamma = 1$.

4.1.5 Design of Scheduler's ready queue

The previous scheduler maintains a ready queue (containing the operators) that provides the order in which operators are scheduled. Once these operators are instantiated by the instantiator, server populates the ready queue. Scheduler chooses an operator for execution. The operator chosen by the scheduler processes the tuples. Operators must be in ready state in order to be scheduled. The problem with this approach is that we will not be able to switch the scheduling strategy depending on the QoS requirement. Based on the requirement of the new scheduling strategies that have been implemented we propose a new design where the ready queue contains the list of operator paths, segments, simplified segments and operators in the bottom-up order. Figure 4.1 shows this new design, which will allow us to switch from one scheduling strategy to other when ever required. This design is extensible because if we add a new scheduling strategy, in the future, we just need to add another list (which ever order needed by this new scheduling strategy) to the ready queue corresponding to the added scheduling strategy.

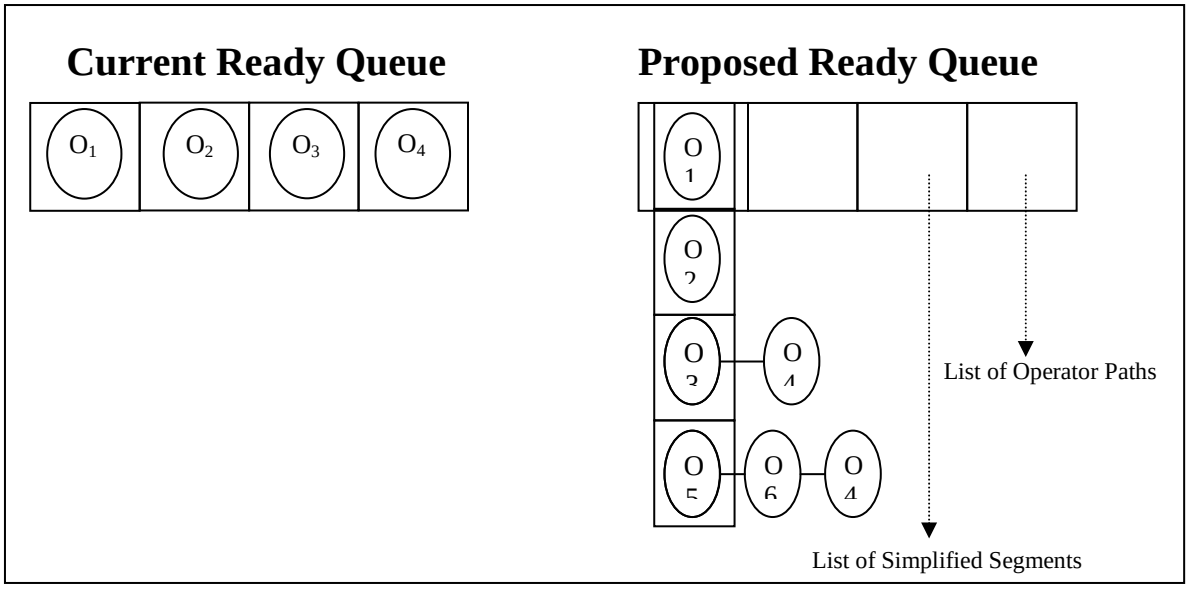


Figure 4.1 Scheduler's Ready Queue

4.1.6 Design Issues

In this section we refer to operator paths, segments and simplified segments as paths. Some of design issues which we faced in introducing these new scheduling strategies are:

i. As the query tree given by the user is known to the client and the server, paths can be created on either side. Which is better and why?

ii. Is it better to create paths before instantiation or after?

iii. The lists of paths in the ready queue should be sorted based on their priorities. For example, in segment scheduling strategy the list of segments in the ready queue should be sorted in the descending order of their memory release capacity. When is it appropriate to sort this list of paths in the ready queue? Is it better to sort them whenever a new query is added to the system or for every iteration of the scheduler?

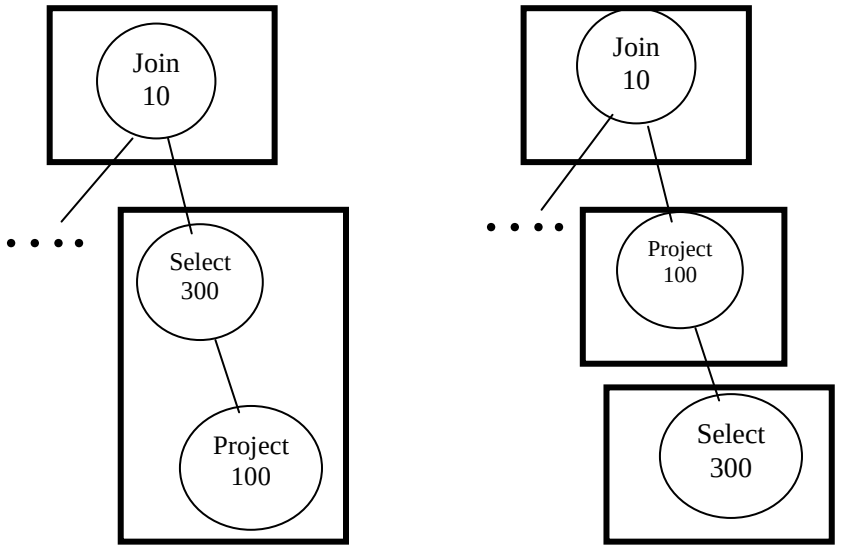
iv. One of the available paths is given higher priority (e.g., in path capacity scheduling strategy, operator path with higher processing capacity is given priority) in the scheduling strategies and the other non-scheduled paths will not get scheduled for a long time and tuples increase in their input buffers creating starvation. How to reduce this starvation?

The following subsections addresses the design issues specified above

4.1.6.1 Creating paths on client side vs. server side

Figure 4.2 shows a query tree in which each circle denotes an operator, the type of the operator and its memory release capacity. One or more operators are grouped into a segment (shown by rectangles in Figure 4.2).

Consider the operator path of the initial query tree shown in the Figure 4.2a given by the user, where the operator's project and select are grouped together as one segment and join operator as another. During this query execution, Run-Time Optimizer monitors the QoS requirements. If the QoS requirements are not met, it might probe the Alternate Plan Generator to generate an alternate plan (for e.g., Figure 4.2b). In this newly generated plan each operator becomes a different segment as the query tree changes. The processing capacity of operator paths and the memory release capacity of the segments depend on the selectivity of each operator. The starting selectivity of each operator is assumed by the server and during the course of query execution the selectivity of each operator is monitored and updated. This might change the processing capacity of operator paths and memory release capacity of the segments. These situations will not be considered if the paths are created on the client side. Hence, we create paths on the server side.



(a) Initial Query (b) Alternate Query

Figure 4.2 Example Query Tree – 1

4.1.6.2 Creating paths before or after instantiation

Creating paths after instantiation of query tree will result in a delay for the start of the query. For example, suppose a user wants to start a query at a future time. We instantiate the query tree at that time and if paths are created after instantiation the start time of the query is delayed.

Creating paths before instantiation will start the query without any delay. As the query given by the user is a tree of operator nodes which has complete information about the flow of data, the starting selectivity of each operator is assumed by the system and the paths are computed from the query tree.

4.1.6.3 When to sort the ready queue

Figure 4.3 shows a query tree given by the user which has two operator paths. Assume the selectivity of Operator S_1 as ‘a’ and Operator S_2 as ‘b’ and the processing

capacity of operator_path$_1$ as ‘x’ which completely depends on operator S_1 and the processing capacity of operator_path$_2$ as ‘y’ which completely depends on S_2. If x > y, then operator_path$_1$ is given higher priority than operator_path$_2$. But during the query execution as the selectivity of operators is monitored, the values of the selectivity might change and because of this the priorities of the operator paths might change as well. If we sort the ready queue only when a new query is added it will not consider the change in selectivity of operators and hence will be a pessimistic approach.

If we sort the ready queue at the start of every iteration of the scheduler, at any point the paths will be in sorted order which is an optimistic approach. This approach introduces an overhead of sorting. As the number of paths in a ready queue is quite less, the overhead induced is minimal.

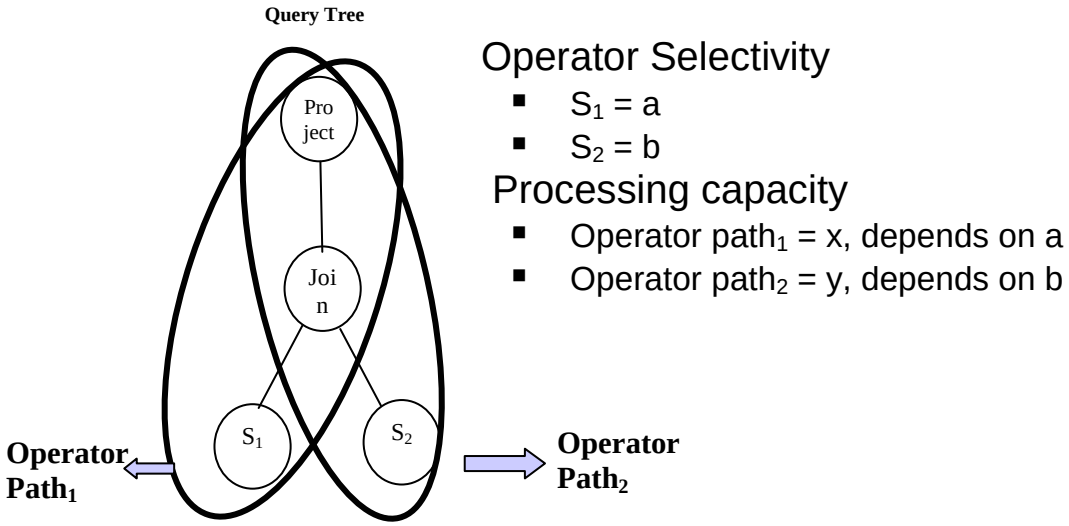


Figure 4.3 Example Query Tree – 2

4.1.6.4 Reducing Starvation

Tuples waiting in the base buffers of the non-scheduled paths will starve. If we schedule paths based on the minimum threshold (τ) on the number of tuples present in the input queue we can reduce starvation even though we cannot completely avoid it. The threshold value is chosen based on the system load. If we choose a small value then the scheduling overhead increases, and a large value might result in increased tuple latency.

4.1.7 Evaluation of scheduling strategies

4.1.7.1 Path capacity scheduling strategy

In the round-robin and segment scheduling strategy, as the tuples are buffered in the intermediate queues, the waiting time of the tuples increases, which increases the average tuple latency. At the time of temporary overload situations where the input rate is more than the processing capacity of the operator there is no throughput from the system in a segment scheduling strategy as most of the computation resource are given to the operators at the bottom of the tree. In round-robin and segment scheduling strategies the operators of two operator paths are scheduled in an interleaving manner [13], which increases the tuple latency and throughput. Path capacity strategy has starvation problem as most of the resources are given to the operator path having maximum processing capacity and the operator paths which have less processing capacity will starve. Path capacity strategy has less scheduling overhead than the round-robin as in any query plan the number of operator paths is less than the number of operators. As each operator is implemented as single thread, context switching by both scheduling strategies are same.

4.1.7.2 Segment scheduling strategy

In the query plan construction we generally push the lower selectivity operators down the tree, due to which there will be less number of the tuples present in the system. Segment scheduling takes advantage of lower selectivity and higher processing rates of the bottom side operators where these operators consume more number of tuples and give out less number of tuples (releases more memory) by scheduling the segment with maximum memory release capacity more often. In the segment scheduling we buffer the partially processed tuples at the start of the segments which contributes towards higher tuple latency than the path capacity strategy. Segment scheduling has less scheduling overhead than the round-robin as the number of segments is less than the number of operators in a query plan.

4.1.7.3 Simplified segment scheduling strategy

Memory requirement is slightly more than the segment scheduling strategy because the first segment releases more memory. The tuple latency is less than the segment scheduling strategy because the tuples are buffered at most two times in the operator path. The scheduling overhead is less than the segment scheduling because the number of segments is more than the simplified segments in a query plan.

4.2 Query Instantiator

Query given by the user is a tree of operator nodes. But the server needs the operators to be instantiated in order to run a query. So, we need to instantiate the query tree to have an executable version of operators. The server gives the query to the Instantiator based on the query start time.

4.2.1 Requirements of query Instantiator

- Traverse the query tree in post order form and instantiate the operator nodes and create output buffer to each Operator.
- Associate the output buffer of one operator to the input of another operator as specified by the query tree.
- Instantiator should give the output buffer of the query, which is the output buffer of the root node to the application as needed.
- Instantiator should assign the corresponding operator property of the Operator Node to point to the instantiated Operator.

4.2.2 Design of Query Instantiator

User is presented with a GUI interface where he is allowed to submit a query plan. Interface provides a set of operators on which a user can query. User can give schema definition of new streams and also specify the source of the streams using the interface. At present, as there are no real streams that are being received from the sensors. Users can associate the flat files which have some data with stream names. Once the operator is selected user should also be able to give the condition on the operator. For example, if the user selects PROJECT operator he should give the fields to project on the particular stream and for JOIN the user should specify the joining attribute and condition. User is provided with all the stream definitions that are present in the system so far. For operators, user should also specify the input source stream. The input source can be a new stream or an intermediate stream output from other operators. If the operator output produces a new stream, a new schema is generated and added to the schema definitions which are already present in the system. After user specifies the entire query, the client

creates a query tree consisting of operator nodes. User also specifies the QoS specifications such as tuple latency and memory requirement for the query and variations tolerable as percentage from the above specifications (e.g., average tuple latency for query is 20 seconds with 10 % tolerability, i.e., range of tuple latency is 18-22 seconds).

The query given by the user is made into a query tree which is a tree of operator nodes. Each operator node is embedded with operator data (operator description), pointers to the left, right and parent operator nodes and a pointer to the operator which is null at the client side that is populated when the operators are instantiated.

4.2.2.1 Operator Data

These are the some of the additional requirements for the operator data:

- It should specify the initial selectivity of the operator which is needed to calculate some of the values such as processing capacity of an operator path and memory release capacity of an operator.
- It should be able to store the processing capacity of the operator. Processing capacity of an operator depends on the system which does the query processing.
- It should be able to compute and store the memory release capacity of the operators.

Based on the above additional requirements, operator data data structure was extended to have the following parameters.

- Selectivity: This specifies the initial selectivity of the operator. This can be updated by monitoring the output from the operator. For operators like JOIN selectivity can be more than one.

- Processing Capacity: It is the number of the tuples an operator can process in a time unit, which is dependent on the system on which the server is running.
- Memory Release Capacity: It is the number of bytes of memory an operator can release in a time unit. It depends on the processing capacity of the operator and the input and output tuple size. For operators like SELECT, the input and output tuple size is equal.

4.2.2.2 Association of Buffers and Operators

In order to instantiate operators, the query tree is traversed in post order. It picks up an operator node, identifies the type of the operator by using the field OperatorType in operator data and the appropriate operator is instantiated. Each operator's output buffer is used as input buffer to the next higher level operator instantiated. If the operator is the leaf operator then it should read the input stream name and associate the corresponding buffer of the stream. As the operator design requires the input parameters to be in a specified format, the input parameters given by the user are changed to the operator understandable format by the Instantiator. For example, the input parameters given to the PROJECT by the user will be the fields to project. But the operator wants them in the form of positions in the stream to project. Some of the operators also require the Instantiator to set the window information. For windowed operators like JOIN it should know when the query ends.

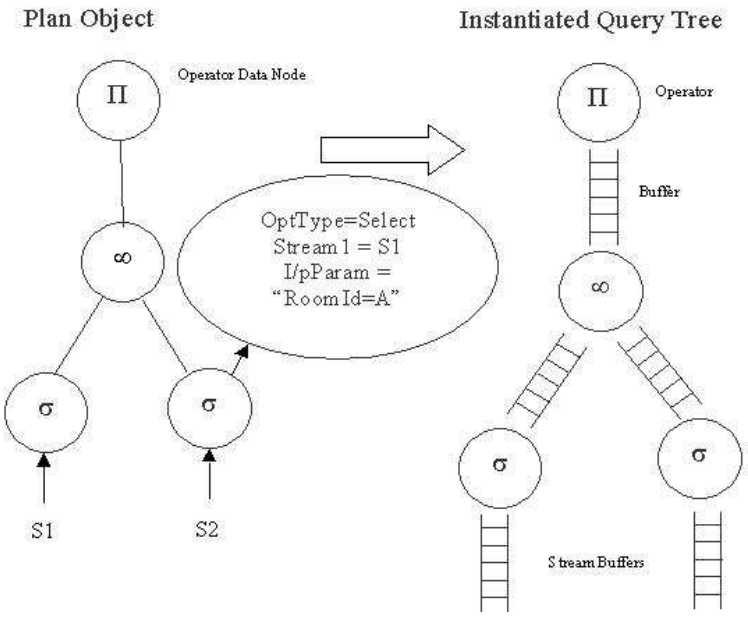


Figure 4.4 Process of Instantiation

Figure 4.4 shows the process of instantiation. Instantiator takes a plan object from the Server which is a tree of operator nodes. It traverses the tree in a bottom up fashion, instantiates each operator and the output buffer is given as input buffer to the higher operator.

4.3 MavStream Client-Server Model

MavStream uses client-server architecture with client requesting services provided by the server. Client provides GUI where user can give the schema definition, query, window specifications and QoS requirements for the query. These requests from the user require some pre-processing on the client side after which will be sent to the server using pre-defined protocols. Server initializes some of the parameters needed for other components like buffer and scheduler. Figure 4.5 illustrates the communication between the client and server.

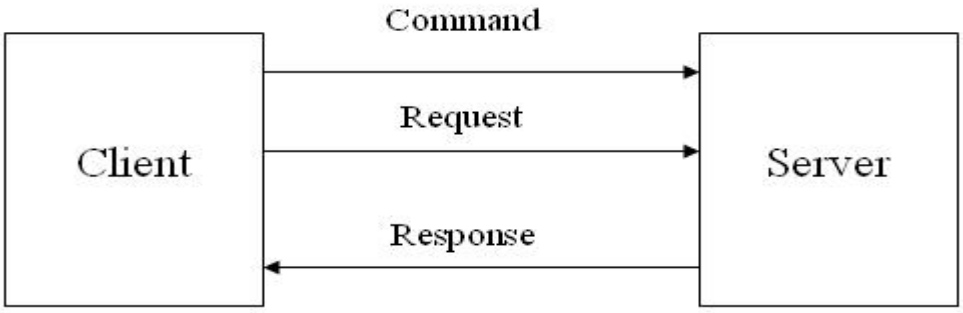


Figure 4.5 Client-Server Communication Model

Client and server communicate over network using sockets; the other alternatives are CORBA and RMI. CORBA is used when different vendors want to interpolate across networks and operating systems, which is ruled out as MavStream is implemented in java and is less complex. RMI is used when the server is distributed; as MavStream is not distributed using sockets keeps the implementation simple and easy to maintain.

4.3.1 MavStream Client

Client is responsible for taking the user requests and to convert them into a form understandable by the server. Client requests a service from the server that listens to it on a specified port. Presently client is of the following two types:

- Non-web based client: This is mainly used for carrying out experiments. Client reads the stream definition and plan object from the text file and sends it to the server by using appropriate protocols. Client does not make use of web features and hence lacks worldwide accessibility.

- Web based client: Web based interface is provided for creating new schema definitions, submitting queries and other requests. Client provides the user with the

schema definitions present in the server. MavStream client being web-based, it can be accessed from anywhere.

There are two types of queries given by the user. One requires the server to start the query immediately and the other kind of query requires starting it at a later time. Client has the following responsibilities:

1. It constructs a query plan object from the user input, which has the query tree of operator nodes defining the order of operator instantiation depending on the flow of data. It has a list of all operator paths, segments, simplified segments and operator nodes in bottom-up order in the query tree.

2. It provides user with the status of the queries, whether they are active or inactive. User may request server to stop a query. User also provides the QoS specifications associated with queries.

3. The query plan object is sent to the server, whether the query needs to be started immediately or at later time.

4. Client generates intermediate schema definitions for operators like JOIN and PROJECT. These intermediate schema definitions should be registered with the server to support the higher operators of the query.

4.3.2 MavStream Server

Server is responsible for accepting the query plan object, stream definitions and QoS requirement from the client. Server is also responsible for invoking some of the components shown in the architecture such as scheduler, Instantiator and Run-Time Optimizer. Server processes the query given by the user over the data streams and gives the output to the application needed. If the QoS specifications given by the user are not

met, Run-Time Optimizer should probe alternate plan generator to get alternate new plan or probe the scheduler to change the scheduling policy. Server integrates and instantiates various modules including operators, buffers and scheduler.

Server receives a command from the client which is unique for each request. Client will then send the corresponding request which the server is expecting. On receiving the query plan object from the client, server populates the list of operator paths, segments and simplified segments which are not populated at the client side because it requires the selectivity of the operator which can be found by monitoring the output of an operator at server. Server performs the following functions:

i. Accepts the query plan object from the client.
ii. Parses the query tree and populates the operator paths of query plan object.
iii. Parse the operator paths and populate the segments of query plan object.
iv. Parse the operator paths and populate the simplified segments.
v. Inputs the query to the Instantiator at the start time of the query, which instantiates the operators.
vi. Server puts these operator references in the scheduler ready queue.
vii. Starts the scheduler thread, which will schedule the operators present in the scheduler's ready queue.

4.4 MavStream Feeder

Presently there are no real streams that are coming to the MavStream system. We need a Feeder, in order to perform experimental evaluation of the MavStream with different input rates. The Feeder uses Poisson distribution in simulating the real-time environment where streams of tuples come in random fashion. The data needed by the

Feeder is taken from flat files. In the coming section we will discuss the design issues and algorithm when tuples are read from the files.

4.4.1 Design Issues

The feeder can be designed using threads with one thread for each base buffer reading from a separate file. This will lead to the usage of a number of resources resulting in low efficiency, since only one thread is allowed to execute at a given time, where as the remaining threads sleep. An alternative design is to use a single thread to read from many files and still able to feed all base buffers. This design is superior compared to the former since there is better utilization of the resources.

4.4.2 Algorithm

Variables: PoissonValues, totalTimeThreadSlept=0, allBuffers

1. When new Buffer is added to 'allBuffers'; generate a poisson value(which is with respect to 0), update the poisson value by incrementing it with 'totalTimeThreadSlept' (to make it relative time with the system); add to 'PoissonValues'
2. Sort the 'PoissonValues' in ascending order
3. Take the first value in 'PoissonValues' (least value), calculate 'timetosleep'='poissonvalue'-'totalTimeThreadSlept'
4. Sleep for 'timetosleep' and en queue tuple in to particular buffer
5. 'totalTimeThreadSlept'=totalTimeThreadSlept + timetosleep
6. Generate another poisson value (which will be with respect to the previous value of this buffer); add it to previous poisson value (to make it absolute); add it to 'PoissonValues'

7. Repeat steps 2 to 6 till all the tuples have been exhausted

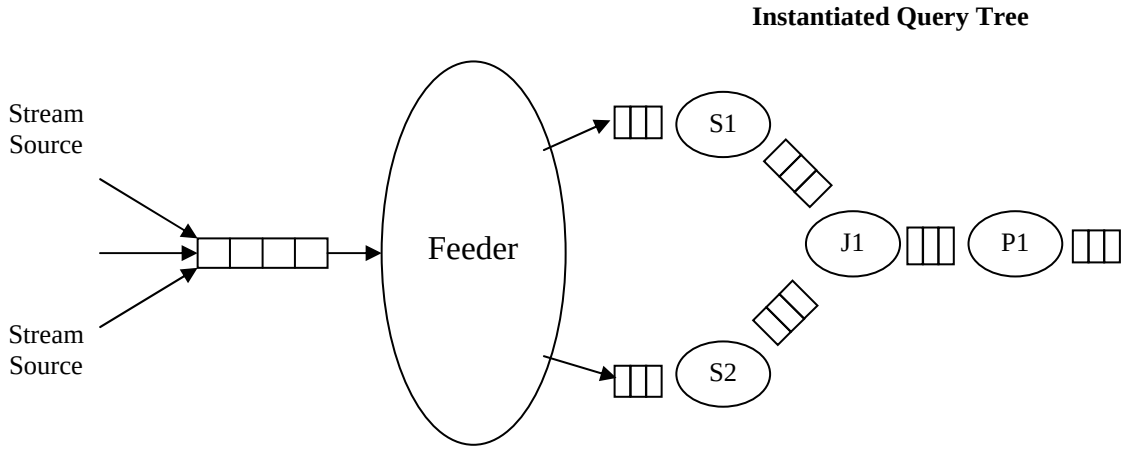


Figure 4.6 Real-Time Feeder

When there are real streams coming to the system, we remove the Poisson distribution and feed the tuples to the base buffers in the order of arrival which is shown in Figure 4.6.

4.5 Chapter Summary

In this chapter we discussed the various scheduling strategies that have been integrated into MavStream such as path capacity scheduling strategy, segment scheduling strategy and simplified segment scheduling strategy. The design of the scheduler's ready queue and some of the design issues that we faced in introducing these new scheduling strategies have been explained. The design issues and requirements of the query instantiator, server and the feeder module have also been elaborated.

CHAPTER 5

IMPLEMENTATION

5.1 Scheduler

After the instantiation of the operators, operators or operator paths are put in the scheduler ready queue. Using a configuration file it reads which scheduling policy to be used for these operators and schedules operators according to that. As the query progresses, the run-time optimizer monitors the QoS requirements and based on this it can change the scheduling policy. Scheduler is implemented as a thread; it removes the operator reference from the ready queue and schedules it for a specific amount of time. Once the scheduler picks the operator, it starts the operator thread and waits. Operators must register themselves with the scheduler, which facilitates communication between the scheduler and operator. Operator can be in any of the four states (ready, run, stop and suspend) during the course of execution. Scheduling strategies that have been implemented are path capacity, segment and simplified segment. All these scheduling strategies are extended from an abstract class Scheduler. In the following sections we will discuss the operator states, issues in the implementation, implementation of scheduling strategies and scheduling classes.

5.1.1 Operator states during the course of execution

Operator can be in any of the four states. Transition from one state to other is controlled by the scheduler or operator itself. Figure 5.1 shows the State transition diagram.

- Ready: When MavStream server receives a signal to start the query, operators are instantiated by the Instantiator and operator references are placed in the scheduler's ready queue. Operators which are ready to be scheduled (in ready queue) are said to be in ready state. The state of the operator which is suspended due to lack of resources is changed to ready on the availability of resources.
- Run: When a scheduler selects an operator from ready queue to schedule it, then operator will be in the run state. At this point the operator thread will be started and it processes its input tuples. If the time quantum assigned for the operator has elapsed it can go to either to the ready state or suspend state based on the availability of resources. If there are no more tuples to process, operator goes to the stop state.
- Suspend: When all the resources needed for the operator are not available then it will go to the suspended state (e.g., when there are no tuples in the input queues to process). If operator gets pre-empted by higher priority operator then the operator will be in suspended state.
- Stop: Where the operator has encountered the end of the query, then it will go to the stop state. If many queries use a common operator, then termination of all of these queries will result in the operator entering the stop state.

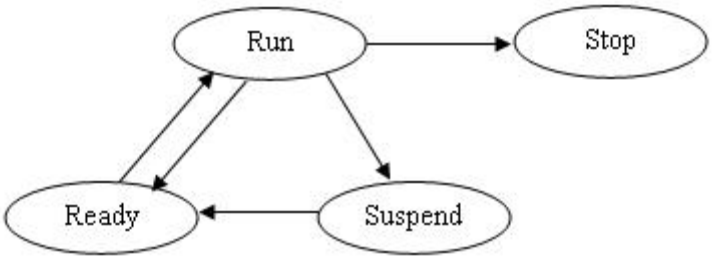


Figure 5.1 Operator State Transition Diagram

5.1.2 Implementation Issues

When a query plan is received by the MavStream server, the operators associated with the query tree are not yet instantiated. Scheduler schedules the operator paths of instantiated operators. It is not possible to create paths (which have operators) when the server receives query plan. Hence we create dummy paths of the query plan which contains the operator nodes and put it in the query plan object. Operator node has a pointer to the instantiated operator, which is populated whenever the operators are instantiated. When the start time of the query occurs, the query tree is instantiated and the path of operators are moved from the dummy paths and put in the scheduler's ready queue.

Scheduler removes the operator reference from the ready queue and schedules the operator for its assigned time units. Scheduler starts the operator thread and wait. Consider round robin scheduling in which every operator is assigned the same time units say 10. If an operator finishes its operation after consuming 5 time units, it would be more appropriate if scheduler thread wakes up immediately rather than sleeping for another 5 time units (completing its full waiting time). If sleep () method had been used,

scheduler thread would have woken up only after 10 time units while wait () method allows it to wake up as soon as operator finishes its operation thus saving time and improving efficiency.

5.1.3 Implementation of scheduling strategies

Figure 5.2 shows the pseudo code of the scheduler. Three strategies have been implemented for scheduling which are as follows:

5.1.3.1 Path capacity scheduling strategy

For each iteration of the scheduler, we sort the ready queue in the descending order based on the processing capacity of the operator paths. The operator path with maximum processing capacity will be first in the queue. Scheduler thread picks up the first operator path and the first operator from that path. If there are enough tuples present in the base buffer of the first operator, then for each operator in the operator path it starts the operator thread (if it has not been scheduled earlier, else it resumes the thread) and executes it for the assigned time quantum. If the time quantum is elapsed and the operator has not finished its operation completely, it picks the next operator in the operator path and schedules it. If the bottom operator does not have enough tuples, scheduler picks up the next operator path (with next higher processing capacity) for scheduling.

5.1.3.2 Segment scheduling strategy

For each iteration of the scheduler, we sort the ready queue in the descending order based on the memory release capacity of the segments. The segment with maximum memory release capacity will be first in the queue. Scheduler thread picks up the first segment (maximum memory release capacity) and the first operator from that segment. If there are enough tuples present in the base buffer of the first operator, then

for each operator in the segment it starts the operator thread (if it has not been scheduled earlier, else it resumes the thread) and executes it for the assigned time quantum. If the time quantum is elapsed and the operator has not finished its operation completely, it picks up the next operator in the operator path and schedules it. If bottom operator does not have enough tuples, scheduler picks up the next segment (with next higher memory release capacity) for scheduling.

5.1.3.3 Simplified segment strategy

It is same as the segment strategy, but the segment construction algorithm is different. Instead of creating many segments, we create only two segments from the operator path. Based on the scheduling strategy we create either segments or simplified segments.

```
addToReadyQueue – adds path to ready queue
getThreshold -  gives the number of tuples in the
                input buffer of the operator
τ - Configurable Threshold parameter
 while( there are paths in the ReadyQueue )
 {
      sort (ReadyQueue)
      for ( int i=0; i <ReadyQueue.size() ; i++)
      {
        if(number of tuples to schedule in leaf buffer of path > τ
      )
        {
          schedule all the operators in the path
          i = ReadyQueue.size() + 1;
        }
      }
  }
```

Figure 5.2 Pseudo Code for Scheduler

5.1.4 Implementation Details

Data structures for paths are implemented using java Vector class as they are synchronized (i.e., only one thread can handle an operator at any given time). For sorting the paths we used java Collections API which sorts using modified merge sort where the worst case of sorting is of order O(nlogn). Paths implement java Comparable class as they must be mutually compared in order to sort.

5.1.5 Algorithm to Handle Subsets and Supersets in Segment Construction

```
RemoveSubsetsAddSuperset (Segment   toBeChecked)
{
     for each CurrentSegment in the SegmentList
     {
        String result=
CurrentSegment.subsetOrSuperset(toBeChecked);
        if (result.equals("EQUAL") || result.equals("SUBSET"))
          {
          return;
        }
        if ( result.equals("SUPERSET") )
          {
          REMOVE  “CurrentSegment  from SegmentList”
        }
     }
     ADD   “toBeChecked to SegmentList”
}
```

Figure 5.3 Pseudo Code for Handling Subsets and Supersets

Figure 5.3 gives the pseudo code for handling supersets and subsets. Whenever a new segment (toBeChecked) is added to the segment list, we iterate through the segment list. If the new segment is either subset or equal to any of the segments in the segment list

we will not add the new segment. If the new segment is a superset of any of the segments in the segments list, we remove those segments. After we iterate through all the segments in the list, if the new segment is not equal or subset to any of the segments then we add this segment to the segment list.

5.1.6 Classes

All the scheduling strategies extend an abstract class scheduler which has the general functionality. In the following subsections we will explain each class in detail.

5.1.6.1 Scheduler

This class is an abstract class, where all the methods of the class are not implemented. It implements a Runnable class. It is used to group objects which have similar behavior. Any scheduler should extend from this abstract class. These are the methods of the Scheduler class.

- addReadyQueue: This is an abstract method. It takes an object (it can be either operator or path) as its input. This method adds the input parameter to the ready queue of the scheduler.
- removeReadyQueue: This is an abstract method. It takes the input as the object (it can be either operator or path). This method removes the object specified in the input parameter from the ready queue.
- run: This abstract method is required since it implements a Runnable class. All the scheduling strategies have their own method of picking the operators in their order and schedule them.

All three methods are abstract methods, which are implemented in specialized schedulers. Their implementation differs from one scheduler to another based on scheduling policies.

5.1.6.2 PathCapacityScheduling

This class extends the abstract class scheduler. It has all the methods of the scheduler class and some additional methods. The following are the methods of the PathCapacityScheduling class.

- addReadyQueue: It takes an object (here operator path) as input. This method adds the input parameter to the ready queue of the scheduler.
- removeReadyQueue: It takes an object (here operator path) as input. This method removes the object specified in the input parameter from the ready queue.
- getThreshold: It takes an operator as input and gives out the number of tuples present in the input buffer of the operator. If the operator is join which has two input buffers, it gives the sum of all the tuples present in the input queues.
- run: This method takes the operator path which has maximum processing capacity and processes all the operators in the path and so on.

5.1.6.3 SegmentScheduling

This class extends the abstract class scheduler. It has all the methods of the scheduler class and some additional methods. The following are the methods of the SegmentScheduling class.

- addReadyQueue: It takes an object (here segment) as input. This method adds the input parameter to the ready queue of the scheduler.

- removeReadyQueue: It takes an object (here operator path) as input. This method removes the object specified in the input parameter from the ready queue.
- getThreshold: It takes an operator as input and gives out the number of tuples present in the input buffer of the operator. If the operator is join which has two input buffers it gives the sum of all the tuples present in the input queues.
- run: This method takes the segment which has maximum memory release capacity and process all the operators in the path and so on.

5.2 Query Instantiator

5.2.1 Instantiator Implementation

MavStream server gives the handle of the query tree to the Instantiator when the query start time occurs. Instantiator traverses the query tree in post order and instantiates operators. Whenever an operator is instantiated, corresponding operator field of the operator node is populated. In the following subsections, we discuss the implementation of StreamBufferList and the process of instantiation.

5.2.1.1 StreamBufferList

It is a list of the streams and its corresponding buffers which is implemented as a Hashtable with Stream Name as the key and object of Buffer as its value. This is done to provide a quick mapping between Streams and its buffers. These hashtable is used for base buffers and not the intermediate buffers.

5.2.1.2 Query Tree instantiation

Upon receiving the query tree from the server the Instantiator gets the handle to the root of query tree which has access to the entire tree. The process of instantiating operators from the query tree and linking them with buffers is implemented as a recursive

algorithm. It takes the operator node and a Boolean value, which indicates whether it is a left child or not, and returns the output buffer. Initially the value of the root node is passed and it is declared to be a left child even if it does not have any parents. The query tree is traversed in post order. This is done so as to get a bottom up instantiation of operators to construct a query processing tree that supports data flow based computation and scheduling.

Once an operator node is extracted from the tree, it is checked to see if it is a leaf node. If so, then using the stream names from the operator data, it will pick up the corresponding buffer from the StreamBufferList and passes it on to instantiate the operator and get the output buffer from the operator. If the node is a left child to its parent, we assign the output buffer to the leftInputQueue else to the rightInputQueue.

For non-leaf nodes, the leftInputQueue and rightInputQueue buffers are set while instantiating the operator. Based on whether the instantiated operator is a left or right child, the corresponding output buffer of that operator is set to the leftInputQueue or the rightInputQueue of the parent operator in the tree. This is done in a recursive manner from the bottom to the root node.

All of the extracted information from the tree and its buffer mappings are passed to the *extractOperatorNodeInfo* function, which calls the appropriate operator instantiation routine based on the operator type to instantiate the operator and returns the output buffer of the operator.

5.2.1.3 Initializations done by Instantiator

Every operator is an independent entity and needs input in its specific form for which the input specification given by the user is modified to fit the format of the operator.

For example, join operator expects following inputs:

- position of left join attribute in its input stream
- position of right join attribute in its input stream
- data type of attribute
- relational operators constituting the condition

An operator such as Project shrinks the schema while Join expands the schema. Hence prior to higher operator instantiation, their input buffers must be associated with new streams generated by preceding operators.

5.3 MavStream Client-Server Model

As discussed in the design section 4.3 MavStream Client-Server Model, MavStream is a client-server architecture where client requests for the services provided by the server. The request from the user can be the creation of query or schema definition with the QoS requirements included in it. Before sending the user request by the client to server, it needs to do some pre-processing so that, server understands the user request. Server initializes some of the parameters needed for other components like buffer and scheduler. The socket-based connection allows client and server to exchange request-response objects based on a pre-defined communication protocols. In the coming subsection we discuss the implementation details of the MavStream client and the server.

5.3.1 Client Implementation

The core functionality of MavStream client lies in construction of query plan object where it contains the query tree which is a data flow operator-buffer graph formed from the user query and a list of all the operator paths, segments and simplified segments.

A data structure is needed which contains the complete information of the operators specifying the type of operator, name of the input streams and output stream, its window specification and selectivity known as operator data.

Once the information about all the operators is populated then a tree is constructed with each node having the operator data, pointers to left, right and parent node and a pointer to the real operator which is null at client side and populated after the instantiation of the operator known as operator node. The root node has access to all the operator nodes.

After the query tree is constructed, there is a wrapper called query plan object (Figure 5.4) which contains the root node of the query tree and the list containing all the paths in the query tree. These paths are empty at the client side and are populated by the server because it needs the selectivity and processing capacity of the operator.

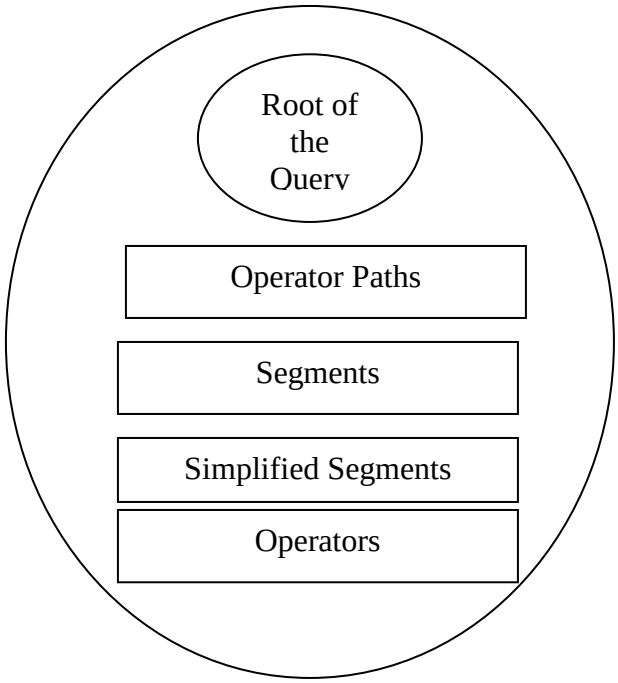


Figure 5.4 Query Plan Object

In the following subsection we will discuss the data structure of operator data, operator node and query plan object.

5.3.1.1 Operator Data

We discuss some of the important parameters of the operator data data structure:

- Operator Type: This specifies the type of operator like SELECT, PROJECT, JOIN etc.
- Stream One: Name of the input stream. For operators like SELECT, PROJECT and AGGREGATE only one input stream is required.
- Stream Two: For operators like JOIN, which executes on two input buffers, Stream Two provides the schema definition on the other buffer. For operators which do not have two streams, stream two will be null.

- Selectivity: This specifies the initial selectivity of the operator. This can later be updated by monitoring the output from the operator and overwriting the selectivity. For operators like JOIN, selectivity can be more than one.
- Processing Capacity: It is the number of the tuples an operator can process in a time unit, which is dependent on the system characteristics of the server. This field is empty on the client side and populated when it comes to the server.
- Memory Release Capacity: It is the number of bytes of memory an operator can release in a time unit. It depends on the processing capacity of the operator and the input and the output tuple size. For operators like SELECT, the input and output tuple size is equal. This field is empty on the client side and populated when it comes to the server.
- Windowed Parameters: This parameter becomes useful for operators that are window-based. Some of the parameters are begin window, end window, hop size and end query.

5.3.1.2 Operator Node data structure consists of

- OperatorData: This is operator data that stores the required information in order to create and instantiate an operator object.
- ParentNode: This node points to the parent object of the operator node. If the node is the root node then the parent field is empty.
- NodeLeftChild: This points to an object of operator node class and represents the left child of the current object.
- NodeRightChild: This points to an object of operator node class and represents the right child of the current object.

- CorrespondingOperator: This is of the operator type. It points to the operator instance created from this operator node. This field is empty at the client and populated when the operator is instantiated.

5.3.1.3 Query Plan Object

- OperatorNode: This is the root node of the query. It contains the entire query tree.
- RoundRobin: This contains the list of operator nodes in bottom-up order.
- OperatorPaths: This contains the list of operator paths present in the query plan.
- Segments: This contains the list of segments present in the query plan.
- SimplifiedSegments: This contains the list of simplified segments present in the query plan.

5.3.2 Server Implementation

Client signals the server to receive the query plan object. Upon receiving the query plan object from the client, server populates the operator paths, segments and simplified segments for the query. In the sections below, we will discuss the operator path and segment classes and commands supported by the server.

5.3.2.1 Operator path

Operator path class implements java Comparator class using which we can compare one operator path with another operator path based on the processing capacity of the operator path. The properties of an operator path are:

- ProcessingCapacity: Number of tuples that can be processed by this operator path in one time unit.
- Operators: List of the operators in the operator path in the bottom-up order.

The following methods are associated with the operator path class.

- ComputeOperatorPathProcessingCapacity: This method computes the processing capacity of the operator path by processing all the operators present in the operator path.
- Size: This method gives the number of operators present in the operator path.
- CompareTo: This method takes another operator path as input and compares the processing capacities of both the operator paths. This method is implemented from the Comparator class.

5.3.2.2 Segment

Segment class implements the java Comparator class by which we can compare one segment with another segment based on the memory release capacity of the segment. These are the properties of the segment.

- MemoryReleaseCapacity: The amount of memory that can be released by this segment in one time unit.
- Operators: List of the operators in the segment in bottom-up order.

These are the methods of the segment.

- ComputeMemoryReleaseCapacity: This method computes the memory release capacity of the segment.
- Size: This method gives the number of operators present in the segment.
- CompareTo: This method takes another segment as input and compares the memory release capacities of both the segments. This method is implemented from Comparator class.
- SubsetOrSuperset: This method takes another segment as input and returns a string indicating whether this segment is subset or superset of the other segment.

5.3.2.3 Commands supported

These are the some of the commands supported by the server.

- New Stream: With this command user is able to create a new schema with the server. Client takes the schema input from the user and converts it into a form given in the schema definition. This is sent as an input for the command. Server adds this schema to the MavStream schema table.
- New query: With this command user will submit the query definition. Client takes the query given by the user and transforms it to query plan object and sends that to the MavStream server. Upon accepting the query plan object, server will create the paths associated with the plan and populates the query plan object data structures.
- Stop query: User should be able to stop a long running query as desired. MavStream expects the client to send the handle of the query that needs to be stopped. From the Query Data Structure, the server picks up the operators running for that query and stops each of the running operators. System, then updates the state of the query in the query data structure to be terminated.

5.4 Feeder

As discussed in the design section 4.4 MavStream Feeder, Feeder is implemented as a single thread. Server provides the reference of the buffers to the feeder and tuples are fed into these buffers. Server should specify the name of the file (given by the user) that contains the tuples for each buffer and also the input rate at which the tuples are to be fed. Once the feeder has the necessary information the tuples are fed into the buffers at the input rate specified by the server. All these information (buffer, filename, input rate) is

given to the Feeder by wrapping them into a FeederComparable object from the server. Whenever this object is added to the Feeder object it generates the poisson value corresponding to that buffer and makes it relative to the sleeping thread of Feeder and adds it to the poisson values list. Feeder thread picks the least value from the poisson values list and finds the time feeder thread needs to sleep. Thread sleeps for the time calculated and enqueue's the tuple into that buffer. After we enqueue, we need to generate another poisson value for that buffer and update that value. In the following sections we will look at the definitions of various classes that are needed to implement the Feeder.

5.4.1 Classes

5.4.1.1 BufferPoisson

This class implements the java Comparable class, so objects of this class can be compared in order to be sorted. The following are the properties of the BufferPoisson class:

- poissonValue: This is a double value specifying the time to sleep.
- bufid: This is the id of the buffer to which the Feeder should sleep.

The method of the BufferPoisson class is

- compareTo: It takes other BufferPoisson object as input and compares this object with the input object.

5.4.1.2 Poisson

There will be a separate Poisson object associated with each of the buffer that needs to be fed by the Feeder. The following are the properties of the Poisson class:

- lambda: This is the input rate specified by the server.

- bufid: This is the id of the buffer to which the Feeder generates poisson values.

These are the methods of the Poisson class.

- getNext: This method gives the BufferPoisson object, which has the next poisson value that the thread has to sleep along with the bufid.
- setBufID: This method sets the bufid property of this class.

5.4.1.3 FeederComparable

Server creates the object of this class (for every buffer it needs to feed) and inputs this object to the Feeder object. The following are the properties of the FeederComparable class:

- buffer: This is a reference to the buffer to which the feeder thread needs to feed.
- filename: The file from which it needs to read those tuples to feed to buffer.
- inputrate: The rate at which the tuples are given to the buffer.
- Poisson: Object of the Poisson class.
- Bufid: The id of the buffer to which the Feeder should feed.

5.4.1.4 Feeder

Feeder is the thread which feeds the tuples to the buffers specified by the server. This class implements the Runnable class. The following are the properties of the Feeder class:

- association: This is a Hashtable which has the key as the id of the buffer and value as the reference to the buffer.
- poissonValues: This contains the list of the BufferPoisson objects (one for each buffer) .
- totalTimeThreadSlept: This value corresponds to the total time the Feeder thread has slept so far.

The following are the methods of the Feeder class.

- add: This method is used to add the FeederComparable objects to the feeder that specifies the buffers to which the Feeder should feed.
- run: This method takes care of the algorithm given in the design section.
- readFileToMainmemory: This method is used to bring all the tuples from the file into the main memory data structures.

5.5 Chapter Summary

In this chapter we discussed the implementation details and implementation issues we faced in integrating the scheduling strategies such as path capacity scheduling strategy, segment scheduling strategy and simplified segment scheduling strategy. We have explained the pseudo code for the scheduler and the algorithm to handle subsets and supersets in segment construction.

Implementation of streamBufferList, query tree instantiation and initializations done by the instantiator are explained. We discussed the data structures of operator data, operator node and query plan object. The commands supported by the server are given. The implementation of feeder module is explained.

We have elaborated the entire API for all the classes implemented for the scheduler, instantiator, server and feeder.

CHAPTER 6

EXPERIMENTAL EVALUATION

All the experiments were run on an unloaded machine with 2 Xeon processors, 2.4GHz each, 2GB RAM and Red Hat Linux 8.0 as the operating system. The data set for performance evaluation is obtained from the MavHome (A smart Home being developed at UTA for predicting the behavior of inhabitants) [6] live feed collected over a period of time. The live feed is stored in our database that is modified to generate synthetic data stream. This synthetic data stream is fed to this system using the feeder module, where the delays between tuples follow Poisson distribution.

6.1 Calculating Performance Metrics

The performance metrics which we consider for our experimental evaluation are: average tuple latency, memory utilization and throughput.

To calculate the average tuple latency we timestamp each tuple when entering and leaving the system and we find the difference in timestamps which gives the tuple latency. To calculate memory utilization we monitor each buffer for every second and determine the number of tuples present in it. Multiplying number of tuples by the average size of tuple in that buffer gives memory utilization. To calculate throughput we monitor the output buffer for every second and determine the number of tuples increased.

6.2 Effect on varying data rate on Average Tuple Latency

In this experiment, the effect of varying data rate on average tuple latency is observed for various scheduling strategies (simple round robin, path capacity scheduling, segment scheduling and simplified segment scheduling). It is run using a single query with six operators in the system. This experiment is done in the main memory. The data rate is varied from 100 tuples/sec to 900 tuples/sec. The data set is fixed with 1000 tuples/window with 5 windows.

It is observed from the Figure 6.1 that as the data rate increases the "Average Tuple Latency" increases. Higher the data rate, the more is the buffer utilization. This increases waiting time in the buffer that is proportional to the data rate.

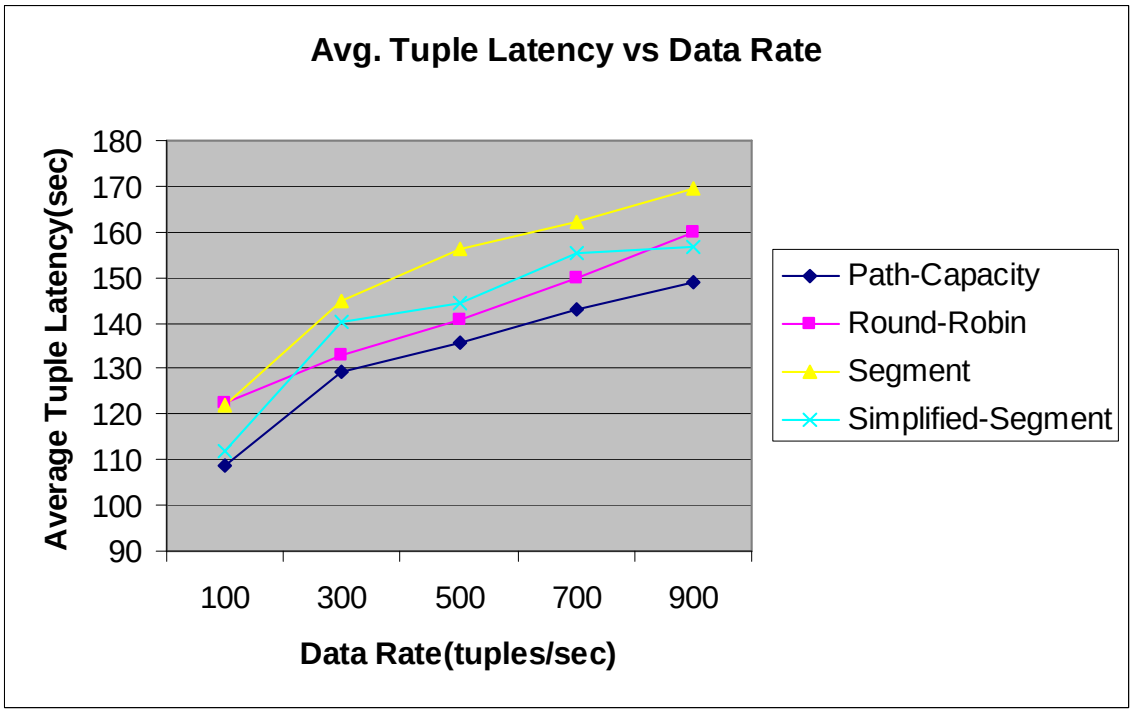


Figure 6.1 Effect on Average Tuple Latency on varying Data Rate

It is important to understand the effect of various scheduling strategies. As expected, the performance of path capacity scheduling is better than other scheduling strategies. In the path capacity scheduling strategy, tuples are processed without buffering in the intermediate queues, where as in the other scheduling strategies, tuples are buffered in the intermediate queues and the waiting time increases as well as the tuple latency. Simplified segment strategy is better than the segment strategy as the number of times a tuple gets buffered is at most twice. The operators of two operator paths are scheduled in an interleaving manner [13] in round-robin, segment and simplified scheduling strategies, increasing the tuple latency. The operators of two operator paths under the path capacity strategy are not scheduled in an interleaving manner and therefore, the path capacity strategy has lesser tuple latency than any non-path based scheduling strategy such as round-robin. On an average path capacity scheduling strategy shows 13% improvement over segment scheduling strategy.

6.3 Effect on varying data rate on Maximum Memory

In this experiment, the effect of varying data rate on Maximum Memory Utilization is observed in various scheduling strategies (simple round robin, path capacity scheduling, segment scheduling and simplified segment scheduling). It is run using a single query with six operators in the system. This experiment is done in main memory. The data rate is varied from 100 tuples/sec to 900 tuples/sec. The data set is fixed with 1000 tuples/window with 5 windows.

It is observed from the Figure 6.2 that as the data rate increases the “Maximum Memory Utilization” increases. Higher the data rate, the more is the buffer utilization.

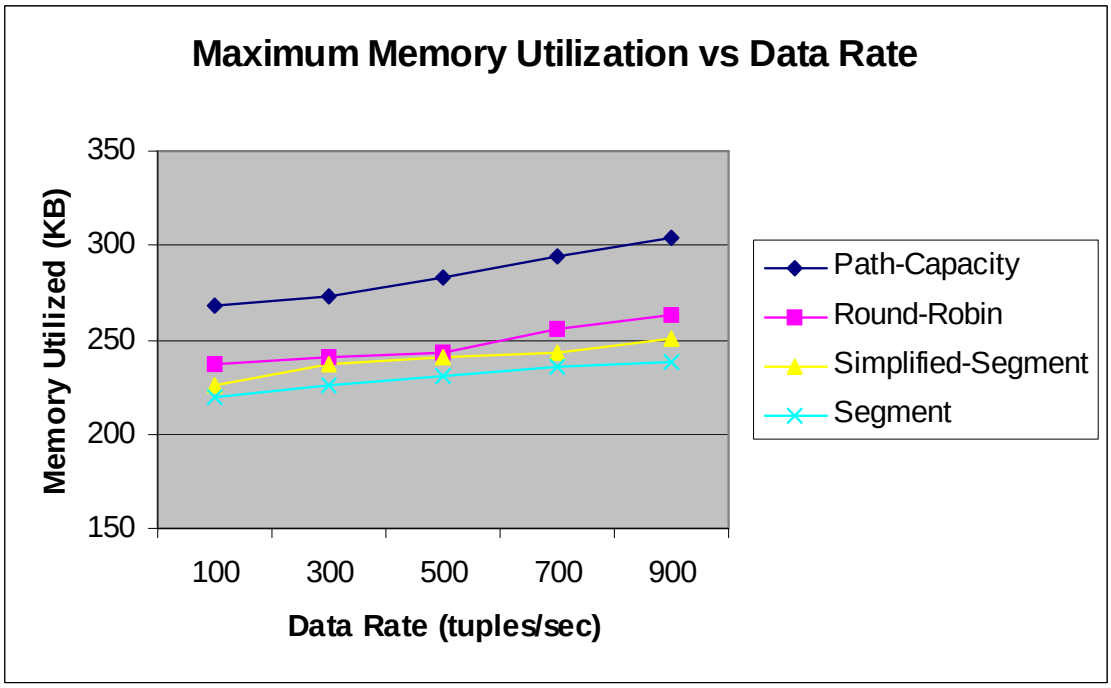


Figure 6.2 Effect on Maximum Memory Utilization on varying Data Rate

The memory utilization of the segment scheduling strategy is better than the other scheduling strategies. In the path capacity scheduling strategy, unprocessed tuples are buffered in the input queues of base operators. Segment scheduling strategy takes advantage of lower selectivity and processing capacity of the bottom operators. Simplified segment strategy takes slightly more memory than the segment scheduling strategy and gives better tuple latency than the segment strategy as the tuple is buffered at most two times. Segment scheduling strategy picks out the particular segment that is most effective at reducing memory usage and schedules it repeatedly as long as there are tuples present in its input queue, reducing memory utilization. On the other hand, round-robin executes the best operator less frequently, increasing the memory usage.

6.4 Throughput of the system during the query execution

In this experiment, the number of tuples given out by the system is monitored during the course of execution of the query in various scheduling strategies (path capacity scheduling, segment scheduling, round-robin scheduling and simplified segment scheduling). It is run using a single query with six operators in the system. This experiment is done in the main memory. This is done so as to study the throughput. The data set is fixed at 1000 tuples/window with 5 windows.

It is observed from Figure 6.3, Figure 6.4 and Figure 6.5 (simplified scheduling strategy is not shown for improved legibility) that the output pattern of the path capacity scheduling strategy is much smoother than the segment scheduling strategy. The total number of tuples given out is the same for a query using different scheduling strategies.

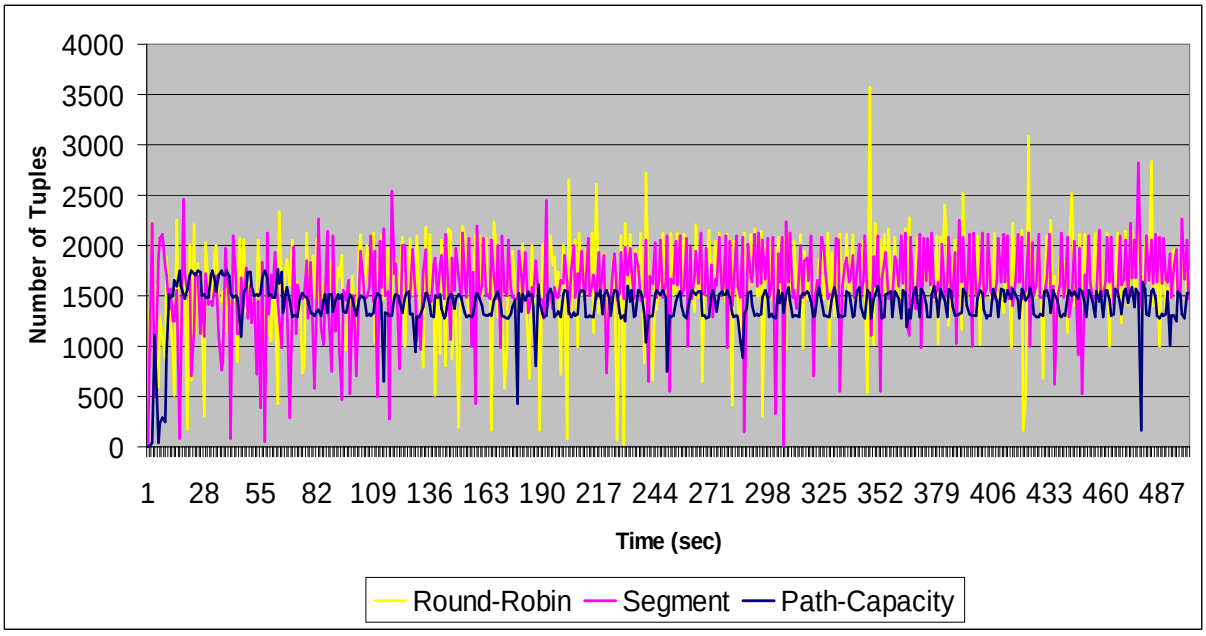


Figure 6.3 Throughput (Data Rate = 100 tuples/sec)

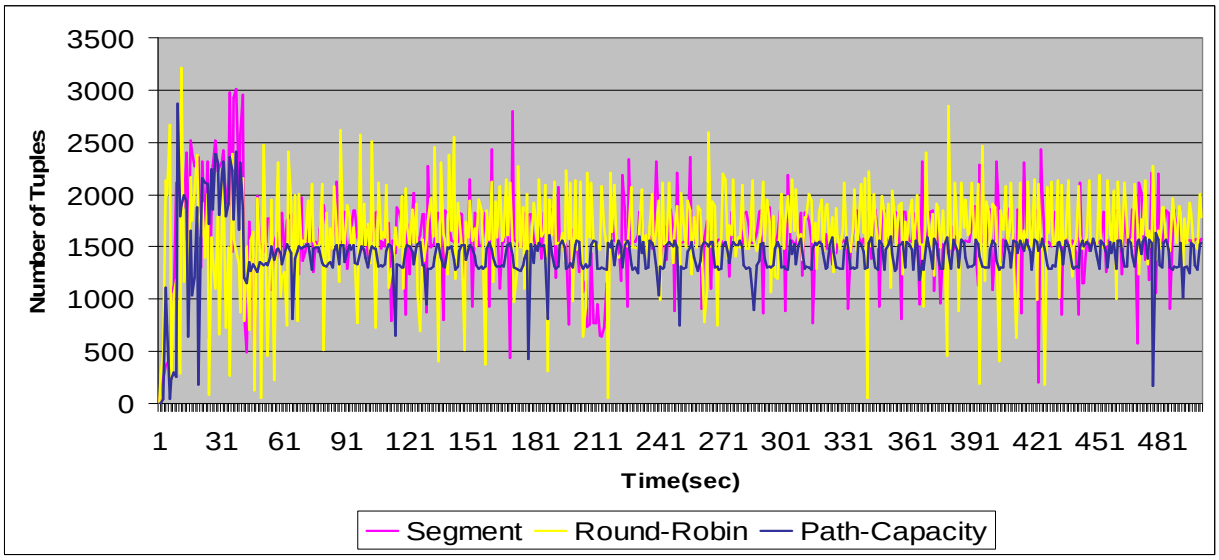


Figure 6.4 Throughput (Data Rate = 500 tuples/sec)

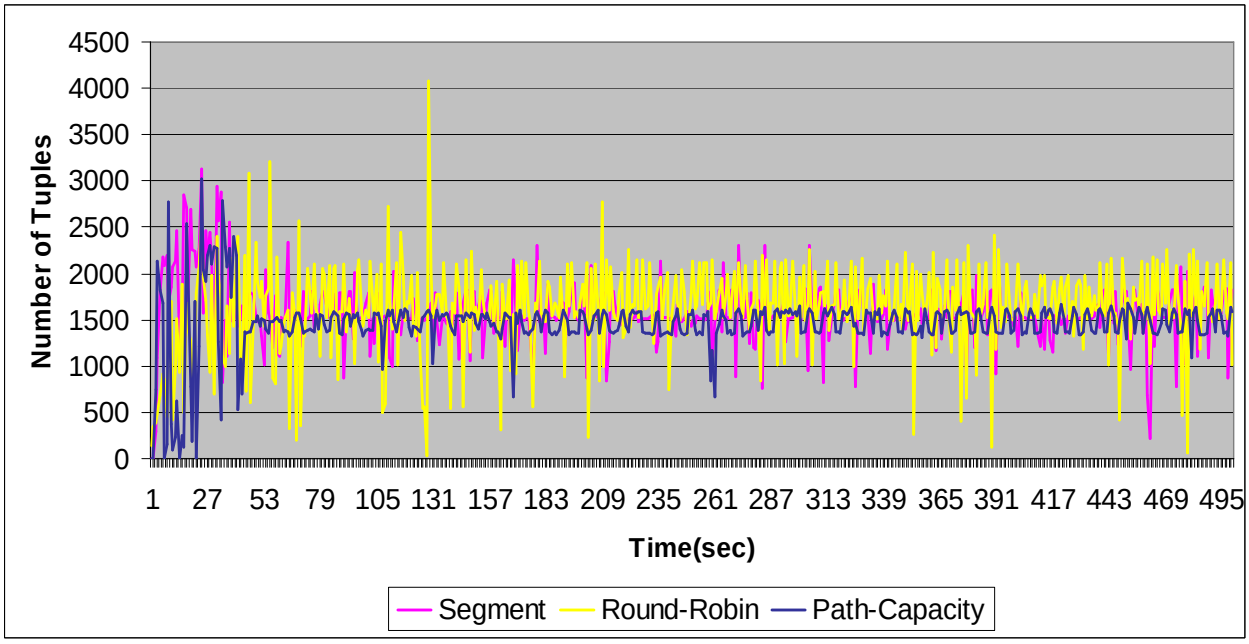


Figure 6.5 Throughput (Data Rate = 900 tuples/sec)

Data Rate	Path Capacity Strategy	Simplified Segment Strategy	Segment Strategy	Round-Robin Strategy
100	237	298	365	386
500	255	312	395	459
900	289	353	417	474

Figure 6.6 Standard Deviation Table

It is important to understand the effect of throughput in various scheduling strategies. The throughput of the path capacity strategy is smoother than the segment scheduling strategy. In the segment scheduling strategy most of the resources are allocated to the bottom operators which have lower selectivity. In round-robin, as all the operators are given equal priority, there will be output only when root operator is scheduled which accounts for the bursty nature of the graph. Figure 6.6, which is the standard deviation representation of the earlier graphs, compares the throughput of these scheduling strategies. As the data rate increases, so does the standard deviation. Path capacity scheduling strategy has less deviation when compared to the other scheduling strategies. Simplified segment strategy has better standard deviation than the segment scheduling strategy.

6.5 Memory utilization by the system during the query execution

In this experiment, the memory utilized by the system is monitored during the course of execution of the query in various scheduling strategies (path capacity

scheduling, segment scheduling and simplified segment scheduling). It is run using a single query with six operators in the system. This experiment is done in the main memory. The data rate is fixed at 40 tuples/sec for the first 150 seconds and increased to 80 tuples/sec till 200 seconds, again decreased to normal rate of 40 tuples/second till 300 seconds and increased to 80 tuples/sec till 350 seconds and brought down to 40 tuples/second and fixed till the end of execution. This is done so as to study the memory utilization in the bursty input rates. The data set is fixed at 2000 tuples/window with 10 windows.

It is observed from the Figure 6.7 that the memory utilized by the segment scheduling is less than the path capacity scheduling. As the tuples are fed to the base buffers the memory utilization increases, once feeding the tuples to base buffers is completed the memory utilization decreases.

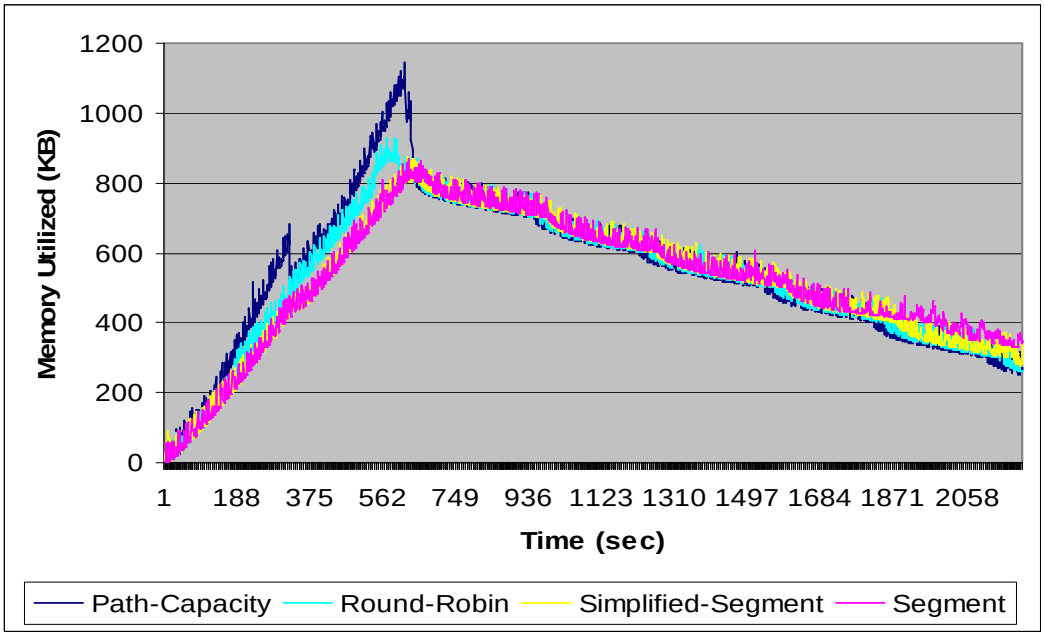


Figure 6.7 Memory utilized by the system

It is important to understand the effect of memory utilized in various scheduling strategies. The memory utilized by the path capacity scheduling strategy is more than the segment and simplified segment strategies. Path capacity scheduling buffers all the unprocessed tuples in the base buffer of the operator path. As the bottom operators have low selectivity they can release more memory, segment scheduling takes advantage of these operator and schedules these operators more often to release more memory. Simplified segment take slightly more memory then the segment strategy as it divides the operator path in to two segments. Path capacity strategy has some bursty nature in the memory consumed at the times when there is increase in the input rates.

6.6 Chapter Summary

Path capacity scheduling strategy has the low tuple latency with high memory utilization. Segment scheduling strategy has high tuple latency with low memory utilization. Simplified segment scheduling strategy has better tuple latency than the segment scheduling strategy but utilizes slightly more memory than the segment scheduling strategy. The throughput of the path capacity scheduling strategy and simplified segment scheduling strategy is steady when compared to bursty throughput of segment scheduling strategy. The experimental summary is given as a table in Figure 6.8.

	Tuple Latency	Memory Utilization	Throughput
Path Capacity Strategy	Low	High	Steady
Segment Strategy	High	Low	Bursty
Simplified Segment Strategy	Better than Segment Strategy	Slightly more than segment strategy	Steady

Figure 6.8 Experimental Summary

CHAPTER 7

CONCLUSION AND FUTURE WORK

In this thesis we have implemented many scheduling strategies for MavStream in order to meet the QoS requirements specified by the user. Path capacity scheduling strategy is implemented to minimize the average tuple latency of the stream processing system by prioritizing operator path which has maximum processing capacity. Segment scheduling is implemented to minimize the maximum memory required by the system by prioritizing the segment which can release more memory. Similarly, simplified segment scheduling takes slightly more memory than segment strategy and gives better tuple latency. We have evaluated these scheduling strategies for various performance metrics such as tuple latency, memory utilization and throughput.

"Feeder" module is designed to feed the buffers of leaf operator with tuples. It splits the incoming stream into multiple outgoing streams based on the split condition. Query plan object is designed which consists of the query tree representing the flow of data and all the paths associated with the query tree. Instantiator is designed to traverse this plan object in post order and instantiates operators respecting the query definition. Some of the protocols between the MavStream client and server are extended from previous work to support additional commands. DSMS server provides a set of services such as addition and deletion of schema, accepting and starting a query.

MavStream server is designed to construct the operator paths, segments and simplified segments from the query tree. It also helps in calculating the performance parameters used by the scheduler such as processing capacity of operator path, memory release capacity by the segments and simplified segments.

As far as future work is concerned, Load shedding [14] of tuples is one area where some of the tuples are shed at times of bursty input rates sacrificing accuracy [2] of the results. Run-Time Optimizer is one module which is not yet implemented in the present architecture. Optimizer monitors the output of the query and compares the QoS requirements and takes the appropriate actions. Alternate Plan Generator needs to be developed to produce alternate plans which can be used by the Run-Time Optimizer to merge an incoming plan with the global plan running in the system to share computation and memory.

REFERENCES

1. Altaf G., Design and Implementation of Stream Operators, Query Instantiator and Stream Buffer Manager, in CSE Department. 2002, University of Texas at Arlington.

2. Motwani R., W.J., Arasu A., Babu S., Datar M., Babcock B., Manku G., Rosenstein J., Olston C., & Varma R., Query Processing Resource Management and Approximation in a Data Stream Management System. IEEE Data Engineering Bulletin, 2003.

3. Babcock B., M.R., Datar M., Babu S., Widom J., Models and Issues in Data Stream Systems. In Proc. of the 2002 ACM Sigmod/Sigact Conference on Principles of Database Systems (PODS), 2002.

4. Satyajit S., Design and Implementation of Windowed Operators and Scheduler for Stream Data, in CSE Department. 2002, University of Texas at Arlington.

5. Carney D, C.U., Cherniack M., Convey C., Lee S., Seidman G., Stonebraker M., Tatbul N. & Zdonik S., Monitoring Streams- A new class of data management applications. In Proc of the 28th International Conference on VLDB, Hong Kong, China, 2002.

6. Group., M. http://mavhome.uta.edu.

7. Jiang Q., C.S., Scheduling strategies for Processing Continuous Queries over Streams. Technical Report, 2003.

8. Centintemel U., R.A., Zdonik S., Carney D. & Mitch C., Stonebraker M, Operator Scheduling in a Data Stream Manager. In Proc of the 28th International Conference on VLDB, 2002.

9. Babcock B., B.S., Datar M., Motwani R., Chain: Operators Scheduling for Memory Minimization in Stream Systems. In Proc. ACM Sigmod International Conference on Management of Data, 2003.

10. Viglas S., N.J., Rate-based Query Optimization for Streaming Information Sources. In proc. ACM Sigmod International Conference on Management of Data, 2002.

11. Avnur R., H., Eddies: Continuously adaptive query processing. In ACM SIGMOD, Dallas,TX, 2000: p. 261-272.

12. David, J.C.J., Dynamic Regrouping of continuous queries.

13. Jiang Q., C.S., Analysis and Validation of Continuous and Queries over Data Streams.

14. Cherniack M., Z.S., Cetintemel U.& Tatbul N & Stonebraker M., Load Shedding in a Data Stream Manager. In proc of 29th International conference on VLDB, September 2003.

BIOGRAPHICAL INFORMATION

Vamshi Krishna Pajjuri was born on 22nd Septemeber, 1980 in Warangal, India. He received his Bachelor of Engineering degree in Computer Science and Engineering from Kakatiya University, Andhra Pradesh, India in May 2002. In the Fall of 2002, he started his graduate studies in Computer Science and Engineering at The University of Texas, Arlington. He received his Master of Science in Computer Science and Engineering from The University of Texas at Arlington, in December 2004. His research interests include stream processing.