DESIGN AND IMPLEMENTATION OF

WINDOWED OPERATORS

AND SCHEDULER FOR

STREAM DATA

The members of the Committee approve the master's thesis of Satyajeet Sonune

Sharma Chakravarthy
Supervising Professor _____

Larry Holder _____

Lynn Peterson _____

DESIGN AND IMPLEMENTATION OF WINDOWED OPERATORS AND SCHEDULER FOR STREAM DATA

by

SATYAJEET SONUNE

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

DEC 2003

ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. Sharma Chakravarthy, for giving me an opportunity to work on this challenging topic and providing me ample guidance and support through the course of this research.

I would like to thank Dr. Larry Holder and Dr. Lynn Peterson for serving on my committee.

I am grateful to Altaf Gilani and Dustin for their invaluable help and advice during the implementation of this work. 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, the SPAWAR System Center-San Diego & by the Rome Laboratory (grant F30602-01-0543), and the NSF (grants IIS-012370 and IIS-0097517) for this research work.

I would also like to thank my family members for their endless love and constant support throughout my academic career.

Dec 08, 2003

ABSTRACT

DESIGN AND IMPLEMENTATION OF

WINDOWED OPERATORS

AND SCHEDULER FOR

STREAM DATA

Publication No. _____

Satyajeet Sonune, MS

The University of Texas at Arlington, 2003

Supervising Professor: Dr. Sharma Chakravarthy

The new processing requirements of streaming applications like financial tickers, network monitoring, traffic management and sensor monitoring are forcing a re-examination of approaches and techniques used in traditional DBMS due to its inability to operate on streaming data as they would require potentially unlimited resources for collecting, storing and processing real time unbounded streamed data in timely manner. Hence the need of a system is realized whose computation can keep up with the data flow to provide real time response to streamed queries by processing endless data streams on the fly.

This thesis addresses the design and implementation of a Query Processing Architecture for stream data, modeled as a client server architecture comprising of various modules such as Instantiator, Stream Operators and Scheduler.. A data-flow operator/queue graph is used for representing a query plan. Instantiator has the responsibility of initializing and instantiating stream operators on accepting user queries from the client over a predefined set of protocols. Aggregates and Nested Join operators have been designed to operate on continuous streams that provide continuous output using the window concept. A new operator called Split has been introduced to divide single heterogeneous stream into multiple homogeneous streams based on application logic. A scheduler has been included so that different scheduling approaches (e.g., round robin, dataflow, weighted round robin) can be tried to understand their effect on response time, memory usage etc. Experiments have been performed by to measure average tuple latency, total query time and memory usage (main and secondary) for different data rates and input stream sizes.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

CHAPTER 1

INTRODUCTION

Traditional database management systems (DBMSs) are repositories, in which all data to be managed is stored on secondary storage and updated as appropriate. . They utilize a request-response paradigm wherein the user poses a logical query, which is evaluated by the query engine. Traditional DBMSs are not suited for most of streaming applications, which are needed for many newer applications. Examples of streaming applications are: financial tickers, network monitoring and traffic management, network security, click stream processing, and sensor monitoring. There are many reasons why traditional DBMSs cannot be used to support streaming applications. Some of them are listed below:

1. It is not practically feasible to store continuous data streams in a traditional DBMS, as they are not designed for rapid and continuous storage of data.

2. Operators of traditional DBMSs are not designed to handle continuous queries on unbounded streams. Operators such as "Join" and "Aggregate" may block forever as the input streams arrive continuously.

3. Traditional DBMSs are always expected to produce precise answers. In streaming, where data stream may arrive asynchronously at a rapid rate, answer may be computed with incomplete information and hence may not be exact.

4. Traditional DBMSs do not support real-time processing of tuples. They cannot be used for network monitoring and network security applications, which have a low tolerance for stale data.

5. Streaming data may be lost, garbled or arrive asynchronously. Traditional DBMSs are not designed to handle such variations in input data and may produce incorrect results.

On the other hand, a Data Stream Management System (DSMS) is designed and developed keeping stream characteristics in mind and attempt to address the problems mentioned above. Some of the important characteristics of DSMSs are as follows:

1. DSMSs can handle continuous streams of data. Data is processed on the fly and results are generated. It does not have to store raw data on the disk. Once data is processed it is either discarded or archived. Thus the resource limitation problem of storing each and every piece of information, as is done in traditional DBMS is solved. Important data may be archived.

2. Monitoring applications are easily supported by DSMSs. In fact they are targeted for trigger-oriented applications. Sensor networks are being widely deployed for measurement, detection and surveillance applications. In a factory warehouse, one may want to trigger an alarm if the sensor reading goes beyond some threshold value. Every application can potentially monitor multiple streams of data.

3. DSMSs provide a new set of operators, which can operate on continuous streams without blocking. Traditional “Join” and other “aggregate” operators, which are

difficult to use for streams are modified to efficiently handle streaming data. They operate on windows that define the boundaries for input data sets. Continuous operations are supported by "sliding" these windows and changing their size. Results are evaluated on the unit of a window of data and the processing repeats for further evaluation.

4. The data arrival rate of streams may sometimes exceed the data processing rate. Thus input queues may start losing data. In such situations, sampling and histogram techniques may be used in order to produce approximate results. DSMS also computes results even when data is lost, garbled or arrives asynchronously.

5. They provide real--time response to streamed queries. A query submitted to the system is run continuously against streaming data. Thus, output is produced continuously and incrementally at the end of every window. Updates of routing tables, network security and monitoring traffic are some of the applications served by the real time response of DSMSs.

It is important to understand the characteristics of *streaming data and streaming queries* in order to understand and justify the need for a Data Stream Management System. Streaming data display following characteristics:

1. Streaming data to be operated on are not available from disk or main memory; rather they arrive continuously and online.

2. Streaming data are potentially unbounded in size. They are continuously generated by sensor class of devices.

3. Streaming data may be lost, stale, garbled or may be intentionally omitted for processing. When input rate is high, it is sometimes necessary to shed load by dropping less important data. Sampling [1] is a common technique used to handle heavy input rates.

4. 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.

Streaming queries can be broadly classified into:

- Predefined Queries, and
- Ad-Hoc Queries

Predefined queries are queries, which are available to the system before any relevant data has arrived.

Ad-Hoc Queries are submitted to the system when the data stream has already started. Hence query referring to past information is difficult to evaluate unless the system supports storage of past information. Since Ad-Hoc [2] queries are not known beforehand, query optimization, finding common sub-expressions, etc., adds complexity to the system.

Predefined and Ad-Hoc Queries are further classified into:

- One-Time queries or snap-shot queries
- Continuous queries

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.

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 output may itself be streamed.

The above summarizes data streams, their behavior and their characteristics. It also clearly explains why traditional DBMSs are not suitable for streaming applications. DSMS is specifically designed for rapid and continuous loading of individual data items, and directly support *continuous queries* that are typical of data stream applications.

The rest of the thesis is organized as follows. In Section 2, the architecture of DSMS is explained. Some of the important modules highlighted in this architecture are Instantiator, operators, buffer manager, scheduler, alternate plan generator, and run time optimizer. Design of the system architecture gives a broader picture of the entire system without going into implementation details. In Section 3 we review recent projects on data stream processing, as well as a plethora of past research in areas related to data streams, such as Aurora, Psoup, Fjords, Eddies and CACQ. This section also attempts to explain new problems the proposed system has addressed in realizing a complete architecture. Chapter 4 discusses the design issues of the entire system. For every

module, it explains the design issues, the alternatives considered and the proposed solution. The functionalities of each module and their inter-relationships are described.

Chapter 5 describes implementation details and emphasizes problems encountered while implementing the system. Experimental results and performance evaluation to validate the system and measure its performance are provided.

We conclude in Chapter 6 by giving an overview of our contributions and a summary of directions for future work.

CHAPTER 2

DSMS ARCHITECTURE

DSMS is modeled as a client-server architecture in which client accepts input from the user, maps it into a form understood by a server and sends the processed input along with other necessary information to the server over a predefined set of protocols. Server, on fetching a request, instantiates its various components, such as operators, buffers and scheduler, executes it and sends the result back to the client. The various components are shown in Figure 2.1.

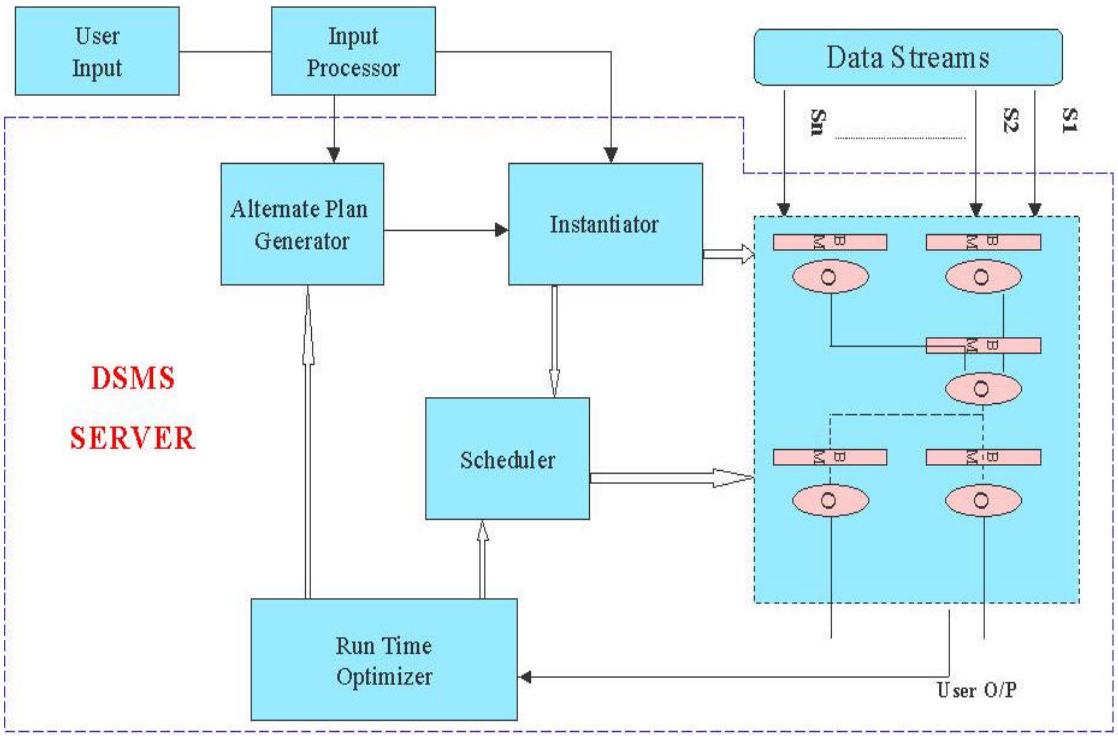

Figure 2.1 DSMS Architecture

This chapter provides a brief overview of various modules constituting the system:

2.1 DSMS Client:

Client provides graphical user interface to pose queries to the system. It not only checks correctness of queries with respect to syntax and semantics but also modifies them into a form acceptable by the server for processing. It constructs a plan object that represents a single complete query from user specifications. It also generates intermediate schema when base schema is altered by operators such as project or join which shrinks and expands schema, respectively. Resolving same stream attribute names into unique names to resolve any conflicts is an added functionality supported by the client. Once the input is processed completely with all needed information generated, it is sent to the server over a defined set of protocols. Communication between client and server is command driven and protocol oriented. A generic model of communication is established in which the client sends a command followed by a request. In order to identify a request, client sends a unique command before sending the actual request. Once the server receives the command, it expects specific request corresponding to the previously received command from a client. Client then sends the request to be processed by the server. Server, based on the command received, processes the request and generates the output. Following are the types of services that are offered.

- Query Input
- Schema Input

- Read Schema
- Stop Query
- Execute Query

Client can be of following types:

- Non-Web based client: Since DSMS is developed in java; user interface may be designed in Swings or AWT components to allow users to construct queries.
- Web-based client: In order to have worldwide accessibility, web-based interface is provided to the user for defining queries. All the client functionality explained above are incorporated in a web server. DSMS client is a web-based client.

2.2 Instantiator:

Instantiator has the responsibility of initializing and instantiating streaming operators and their associated buffers on accepting user queries from the client. It is a sequence of operator nodes. 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 root. Instantiator traverses the query tree in a bottom-up fashion and does the following for each operator node.

1. Creates an instance of the Operator and initialize it on reading operator node data.

2. Associate input and output queues (or buffers) with desired parameters to operators for consuming and producing tuples.

3. Inherit window specifications for window-based operators, such as Aggregate and Join.

4. Every operator is an independent entity and expects predicate condition in a predefined form. Instantiator extracts the information from the operator node and brings it into the form required by each operator.

5. Associate a scheduler with the operator to facilitate communication for scheduling.

The plan object is traversed in post order to ensure that child operators are instantiated prior to parent operator that is required to respect query semantics as data flows from leaves to root. Instantiator does not start the operator, rather it does all the necessary initialization (step1 to step 5) and places it in the ready queue during post order traversal to be scheduled by a scheduler.

2.3 DSMS Server:

This is responsible for executing user requests, and producing desired output. It provides integration and interaction of various modules such as Instantiator, operators, buffer manager and scheduler for efficiently producing correct output. It performs the following functions:

1. Accepts command and request from a client that describes the task to be carried out.

2. 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.

3. 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.

4. Associate input streams with base buffers to start data flow in the system. It also associates buffers with operators as defined in operator node. Inter-operator queues are used to buffer the output of one operator, which acts as an input to one or more operators at the next level of query tree. All these operations are performed by the Instantiator module of the server.

5. Start scheduler to schedule operators for doing necessary computation and produce the result to the client.

2.4 Alternate Plan Generator:

Once the user submits a query, a plan object is constructed. A plan object is nothing but a partially ordered tree that indicates the order in which operators need to be instantiated. If a single plan object is traversed in post-order, it generates only one possible instantiation order. If the operators are instantiated in that order, it might result in un-optimized output, as the order of instantiation may not be most efficient. Consider a plan object in which 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 produced better result. Thus the need for Alternate plan generator is realized which can generate all possible equivalent alternate plans that ensure the same output. The gain may be magnified when it comes to optimizing a global plan by selecting one of the alternate plans in which most number of operators in an alternate

plan merges with the existing operators in the global plan to share memory and computation.

Merging enhances computation sharing and hence facilitates faster response time. Optimizer can make use of 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. The best alternate plan (local optimal) of a query tree may not be the most 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. Devising efficient heuristics that generate good plans is a rich area for future research.

2.5 Operators:

Streaming operators are specially designed to handle streaming data. They operate on continuous streams using the window concept (to avoid blocking) providing continuous and incremental output. There is a close association between buffers (queues) and operators. Every operator has at the most two input queues and one or more output queues. An operator reads from its input queues, performs needed operation based on its semantics and produces result in its output queue. Buffers are shared among operators. The output queue of one operator may become the input queue of one or more operators at the next level of the query execution plan. It is important to design an operator as a controllable entity so that its priorities and various states can be controlled by different entities (such as user, buffer, scheduler, optimizer, etc.,) in a system. Operators can either be in Ready, Run, Suspend or Stop state during the course

of their execution. Transition from one state to another is controlled either by an operator itself or by a scheduler. An operator's priority can be controlled by a scheduler, user or optimizer to satisfy quality of service requirements.

Operators are classified as windowed and non-windowed operators depending on whether they have a window associated with their computation. Split, Select and Project are non-windowed operators. Aggregate and Join are windowed operators as they need a window to define their input boundaries; otherwise they may block forever because streaming data is potentially unbounded in size. Windows are further classified into two types: Disjoint and Overlap. In the disjoint case, end time of current window coincides with the start time of next window. Thus two successive windows do not overlap. In a overlap window, start time of the next window falls prior to the end time of the current window. Thus two successive windows always have some common region (or overlap area) and hence the name.

2.5.1 Non-windowed operators:

1. Select: It has one input queue and one output queue. If the incoming tuple satisfies the given condition, it outputs the tuple to its output queue else it ignores the tuple. The condition to be checked is given by the user as part of a query.

2. Split: It is similar to select with the only difference that select evaluates only one condition while split evaluates multiple condition. Split has one input queue and multiple output queues, one for each condition. An incoming tuple is checked against all the conditions. If it satisfies the condition, it is placed in the corresponding output queue else next condition is evaluated.

3. Project: This operator is analogous to the Project operator of traditional DBMS. It projects only the desired attributes at the output.

2.5.2 Windowed Operators:

1. Aggregate: These operators need a window for their computation. They operate on a window worth of data, performs the needed aggregate operation and produces output at the end of every window. Currently supported aggregate operations are: Min, Max, Average, Sum and Count.

2. Join: This is a blocking operator and hence operates on a window. Two shades of nested loop join are being supported, with reuse and without reuse. It performs join on timestamp ordered tuples collected at its left and right input queues to produce timestamp ordered joined tuples without duplicates. In the case of without reuse, every window is computed independently without making use of the result of the previous window. In reuse, the overlapped region of the current window and the next window is reused (overlapped region is not computed again) for the computation of next window.

2.6 Buffer Manager:

Buffers are the intermediate storage structures used by the operators. All operators in a query tree are connected using buffers. Buffers are implemented as queues. Buffers are of two types: *Bounded and Unbounded.* A Bounded buffer has an upper limit on the number of elements it can store, which can be specified while instantiating a buffer. When the specified limit is reached, successive elements are stored on disk preventing any loss of data. This is in contrast to load shedding [3]

techniques in other systems in which the accuracy of the result is reduced by shedding load at peak time. Buffers are shared among operators. Every buffer internally maintains a common pointer for all operators which points to latest elements read by all operators. All elements including and prior to that are safely discarded which creates main memory buffer space. These buffer spaces can be filled by reading elements from the disk in the order in which they were stored. To accomplish this, buffer incorporates minimal persistent logic to store and retrieve elements to and from the secondary storage as and when needed.

An unbounded buffer has no limit and continue to grow until main memory is exhausted. This makes it a main memory steam processing system. Buffers support two useful operations: dequeue and enqueue.

Dequeue: It removes the top element from the buffer. If the top element is the last element, buffer becomes empty. It makes sense to suspend all operators waiting on the buffer to save CPU cycles. Hence operators attempting to read from an empty buffer are suspended. An element is dequeued only if it is read by all operators sharing it.

Enqueue: A new tuple can be added to the buffer using enqueue operation. If buffer was empty before, it sends resumption signal to all operators upon which they are placed in a ready queue of the scheduler. If the buffer is bounded and the upper limit has already reached, new tuples are added to the secondary memory.

Enqueue and Dequeue operations must be synchronized to ensure correctness as the same location may be accessed by both operations at the same time.

2.7 Scheduler:

Scheduling algorithms developed for real time systems attempt to execute the tasks with the maximum expected utility in order to meet QoS constraints. Each tuple entering a system represents a task but it is not workable as the total number of tasks would be too large for a scheduler. Similarly a query can be considered a task but scheduler is bound to lose the flexibility of scheduling, as the granularity offered by a query may not be acceptable. Thus the most effective way is to perform scheduling at the operator level. Aurora [4] also implements operator scheduling. It schedule operators based on its state and priority. Scheduler maintains a ready queue, which decides the order in which operators are scheduled. This queue is initially populated by the Instantiator while traversing query tree in post order. It chooses an operator for execution, ascertain what processing is required and process them. Operators must be in a ready state in order to be scheduled. One of the following conditions may occur during the running state of the operator.

1. Operator may finish its execution completely upon which it immediately informs scheduler so that the operator waiting next in the ready queue can be scheduled as early as possible. Its reference is deleted from the scheduler.

2. During the execution, operator itself may be suspended because of unavailability of resources. This would remove the operator reference from the ready queue. When all resources become available, then it is again placed at the end of the ready queue.

3. Time quantum of the operator has expired but operator has not yet completed its operation. Scheduler still suspends the execution of the operator and puts the operator at the end of the ready queue to provide fair chance to all operators ensuring starvation avoidance.

Following are the scheduling policies implemented in DSMS based on time quantum:

1. Round-Robin: When all queries and all operators are assigned the same 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 round robin but few operators may get more time-share over others. This policy could be useful to improve the response time and overall performance by assigning higher priorities to deserving operators. 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.

2.8 Run-Time Optimizer:

It is needed in a DSMS for run time optimization based on the quality of service observed. It aims at maximizing the output rate of query evaluation plans. QoS may be end-to-end delay (this delay is the difference in time when the tuple entered the system and when it is seen at the output), number of tuples produced per unit time, or strict

query deadline (user deadline indicating that query must output result before the specified time to be meaningful) for output. To ensure QoS, optimizer may take the following steps:

1. It may ask the scheduler to increase the priority of a query, which needs immediate service. It may also ask scheduler to assign more time quantum to specific operators and/or specific queries.

2. It may ask alternate plan generator to provide a suitable plan from the set of plans available that is better with respect to a global plan running in the system. The alternate plan may not be the best plan if considered alone, but may be the best for the global plan. The idea is to minimize the estimated cost of evaluating a query execution plan.

3. Run time optimizer can identify performance bottlenecks of an already executing plan and ways to overcome them. Either it can maximize the performance estimate for the entire plan or it can locally maximize the output rate at operator level.

Run time optimizer is expected to use all these parameters intelligently to improve QoS. It is supposed to continuously monitor the output and compare with the QoS requirement. If the QoS is respected, it may reduce resource utilization to achieve the same QoS. If the QoS requirement is not met then above heuristics are applied to accomplish the desired goal. This module is a good candidate for future research.

CHAPTER 3

RELATED WORK

This chapter presents an overview of the work done in data streams that addresses various issues in streaming applications and focus on the overall design and characteristics of various systems to handle the challenging problems in data streams.

3.1 Monitoring Streams

Aurora [5] supports continuous query processing, as opposed to a traditional DBMS, in which queries are evaluated continuously over the incoming data stream. They have implemented monitoring applications, which are difficult to implement in traditional DBMS which was developed primarily for business applications. Aurora support trigger oriented monitoring applications that require a large number of triggers and hence support active technology very well. They extend their work from Stream Group [6] that addresses many issues in stream processing. It can support continuous queries, ad-hoc query and views (a path defined with no connected application) at the same time. Aurora can handle a variety of stream data that could be lost, stale or garbled. The emphasis is on quality of service requirements, which are crucial to real-time applications, such as sensor-based monitoring and financial data analysis.

Aurora is fundamentally a data-flow system and uses the popular boxes and arrows paradigm found in most process flow and workflow systems. Tuples flow through a loop-free, directed, graph of processing operations (i.e., boxes). Ultimately,

output streams are presented to applications, which must be programmed to deal with the asynchronous nature of tuples in an output stream. It has a connection point that supports dynamic modification to the network and it also has potential for persistent storage. New boxes (operators) can be added to or deleted from a connection point and it provides access to the recent past, which is beneficial to a new application that connects to the network.

There are few optimization techniques proposed in Aurora. It allows them to insert/move map (project) operations to the earliest possible points in the network, thereby shrinking the size of the tuples that must be subsequently processed. Filter operations can sometimes be pushed down the query tree through joins. Combining Boxes is another optimization technique. For example, two filtering operations can be combined into a single, more complex filter that can be more efficiently executed than the two boxes it replaces.

Aurora also supports run-time network to process data flows through a potentially large workflow diagram. The scheduler picks a box for execution, ascertains what processing is required, and passes a pointer to the box description (together with a pointer to the box state) to the multi-threaded box processor. The QoS evaluator continually monitors system performance and activates the load shedder, which sheds load till the performance of the system, reaches an acceptable level.

The job of the Aurora Storage Manager (ASM) is to store all tuples required by an Aurora network. ASM must manage storage for the tuples that are being passed

through an Aurora network, and it must also maintain extra tuple storage that may be required at connection points.

Aurora exploits the benefits of non-linearity in both intra-box and inter-box tuple processing primarily through train scheduling, which attempts to queue as many tuples as possible without processing, to process complete trains at once, and to pass them to the subsequent boxes without having to go to disk.

This architecture can be extended to support distributed processing. There are various issues in distributed processing such as load shedding, distribution of query plans and collection of results from distributed nodes. They should increase scalability, energy use and bandwidth efficiency. They make the assumption that all tasks are assumed to be present in the main memory and are scheduled and executed in their entirety. Thus they should find the techniques, which allow disk swapping when the amount of information is too large to be accommodated in the main memory.

3.2 PSoup: A system for Streaming Queries over Streaming Data

Psoup [7] is a system that combines the processing of ad hoc and continuous queries by treating data and queries symmetrically, allowing new queries to be applied to old data and new data to be applied to old queries. PSoup also supports intermittent connectivity by separating the computation of query results from the delivery of those results and materializing them, thereby improving throughput and query response times. PSoup is flexible enough to make use of other architectures like eddies to adapt dynamically to the processing of input streams. They make use of efficient data structure called RB-trees, which reduces the time required for indexing at the desired

location for making a search. PSoup efficiently supports other complex operations such as processing of composite tuples and aggregate operations. PSoup allows the processing of data as well as queries on the fly still producing the correct output. Earlier approaches supported arrival of either of the two but not both and hence it is a considerable improvement over the existing mechanisms. PSoup is intelligent enough to share computation thereby conserving resources and improving efficiency. They are generally used in maintaining incremental results. They have also optimized multi-query evaluation by using appropriate algorithms to join the data and query streams. They have also developed techniques to share both the computation and storage of different query results.

This system stores the queries and data in structures called State Modules (SteMs). There is one Query SteM for all the query specifications in the system, and there is one Data SteM for each data stream. The results are materialized in a Results Structure. They defined three different systems based on their storage requirement: NoMaterialization (NoMat): the storage cost is equal to the space taken to store the base data streams plus the size of the structures used to store the queries themselves. PSoup-Partial: in addition to costs incurred by NoMat, PSoup-P also includes the cost of the Results Structure. PSoup-complete (PSoup-C): like PSoup-P, PSoup-C includes the cost of storing the results in addition to the costs included by NoMat systems. PSoup-C always stores the current results of standing queries at a given time. Lazy evaluation (as used in NoMat) suffers from poor response time while having no maintenance costs.

Eager evaluation (as done in PSoup-C) offers excellent response time but has increased maintenance costs.

The system can be improved by having multiple query stems as we have for data streams. This will increase the complexity of the system but it will also increase the query throughput as multiple data streams can be handled simultaneously by different query stems. Psoup is currently implemented as a main memory system. But the system can be improved to archive data streams to disk and support queries over them. Both queries and data can be stored onto the disk. They should come up with a scheduling mechanism to support de-scheduling of queries to disk, which are not frequent. Similarly queries that are invoked often should be given higher priority by the scheduler. This system is suitable for data recharging and monitoring applications that intermittently connect to a server to retrieve the results of a query.

3.3 Continuously Adaptive Continuous Queries Over Streams

The CACQ [8] system is presented on the basis of eddy, a continuously adaptive query-processing operator, which continuously reorders operators in a query plan as it runs; and the Telegraph adaptive dataflow engine as a platform to be used for the continuous query engine. Since earlier approaches used only static query plans, this architecture offers significant performance and robustness gains relative to existing continuous query system and is more aggressive in its ability to share computation and storage across queries over streams.

CACQ is developed from the Telegraph project [9] design and incorporates many significant innovations that make it better suited to continuous query processing

over streams than other continuous query systems. Their work is slightly different from Continuously queries over data streams [10] They use the eddy operator to adapt continuously to the changing query workload, data delivery rates, and overall system performance. They explicitly encode the work, which has been performed on a tuple, its lineage, within the tuple, allowing operators from many queries to be applied to a single tuple. They use an efficient predicate index for applying different selections to a single tuple. They also split joins into unary operators called SteMs (State Modules) that allow pipelined join computation and sharing of state between joins in different queries.

It makes use of Eddies [11] which route tuple through operators (lineage) in a query dynamically and hence it is possible to modify the order of operations in a query plan while the query is in flight. The eddy determines the order in which to apply operators by observing their recent cost and selectivity and routing tuples accordingly.

This contrasts with systems based on static query plans, in which the state of intermediate tuples is implicit in the query plan. Query operators in a static plan operate on tuples of a single lineage. In CACQ this ability is extended to multiple overlapping queries, maximizing the sharing of work and state across queries.

Users may issue queries that join data from distinct but overlapping subsets of sources. They use a space-efficient generalization of doubly pipelined joins within eddy framework. Eddy encapsulates the logic for computing joins over the incoming sources using SteMs. This allows them to incrementally compute a join over any subset of the sources and stream the results to the user.

They have also implemented a variant of the eddy ticket scheme. In their variant, a grouped-filter or SteM is given a number of tickets equal to the number of predicates it applies, and penalized a number of tickets equal to the number of predicates it applies when it returns a tuple back to the eddy. In this way, they favor low-selectivity via tickets and quick work via backpressure.

They present the first continuous query implementation based on a continuously adaptive query processing scheme. Their eddy-based design provides significant performance benefits, not only because of its ability to adapt, but also because of the aggressive cross-query sharing of work and space that it enables. By breaking the abstraction of shared relational algebra expressions, their CACQ implementation is able to share physical operators – both selections and join – at a very fine grain. These features are augmented with a grouped-filter index to simultaneously evaluate multiple selection predicates.

3.4 Fjord

They introduced Fjord [12] which is a hybrid approach for push and pull architecture. It combines push based sensor sources with traditional sources that produce data via blocking, pull based iterator interface. They introduce the concept of sensor proxies, which is responsible for communication between query processors and the physical sensors by doing simple aggregation over sensor data and relaying tuples to appropriate query operators and conserving sensor power by not transmitting sensor data that falls beyond certain threshold values. Fjord architecture does multiple sensor queries on sensors, still conserving sensor resources [13] and maintaining high query

throughput. This architecture can be easily adapted to support different types of query languages for querying streaming data. Fjords have been implemented in real life applications like Traffic Analysis and it can scale to a large number of queries. Sensors are efficiently utilized by sending control messages to adjust their sample rates and their power consumption is also controlled effectively. Operators need not have to worry about the push or pull based architecture. They are thus comparatively less complex. Single Fjord can support multiple queries and it allows allocating streaming tuple only once, which is shared by multiple queries by query folding thereby conserving resources. They have introduced the concept of transition model, which sometimes needs to be scheduled more frequently as compared to other modules as the queries on those modules may be more frequent as compared to others. But operating system has coarse control over thread scheduling and is not useful when scheduling needs some prioritization. They have their own scheduler that handles thread scheduling with prioritization.

Fjords provide support for integrating streaming data that is pushed into the system with disk-based data, which is pulled by traditional operators. Fjords also allow combining multiple queries into a single plan and explicitly handle operators with multiple inputs and outputs.

The key advantage of Fjords is that they allow distributed query plans to use a mixture of push and pull connections between operators. Push or pull is implemented by the queue. By integrating non-blocking operators into Fjords, they take full advantage of Fjords' ability to mix push and pull semantics within a query plan. Another major

component of sensor query solution is the sensor proxy, which acts as an interface between a single sensor and the Fjords querying that sensor.

The history of the stream is not relevant. This means that streaming tuples need only be placed in the query processor's memory once, and that selection operators over the same source can apply multiple predicates at once. Fjords explicitly enable this sharing by instantiating streaming scan operators with multiple outputs that allocate only a single copy of every streaming tuple; new queries over the same streaming source are folded into an existing Fjord rather than being placed in a separate Fjord.

These solutions are an important part of the Telegraph Query Processing System, which seeks to extend traditional query processing capabilities to a variety of nontraditional data sources. Telegraph, when enhanced with Fjords, enables query processing over networks of wireless and battery powered devices that cannot be queried via traditional means.

3.5 Eddies

Eddies [11] support dynamic reordering of a query plan in which they identify "moments of symmetry" during which operators can be easily reordered when they are subjected to changes in cost, selectivity and the arrival rate of tuples. Moments of symmetry allow reordering of inputs not only to a single binary operator but it generalizes the problem to solve any number of binary joins by using the commutative property of a join. They provide runtime adaptavity and a reduction in code complexity, which is not possible with traditional plans. They can be used as an optimizer, which does not need a traditional query optimizer with a complex code. They are also used

with traditional optimizers to improve adaptability within pipelines. It allows the system to adapt dynamically to fluctuations in computing resources, data characteristics and user preferences. This allows each tuple to have flexible ordering of query operators when eddies are combined with appropriate join algorithm. Eddies have a flexible prioritization scheme to process tuples from its priority queue. Their priority scheme is simple to implement and ensures that eddies are not clogged with new tuples. Eddies implements an intelligent lottery scheme for variable selectivity, which is simple to implement and produces effective results. Eddies solves the problem of limiting concurrency due to barriers by using the concept of Rivers and allows I/O and computation to perform simultaneously. It is developed to work efficiently in large-scale system with unpredictable and fluctuating environment. It takes into account the problems caused by hardware, data and user interface complexity in large-scale systems.

An 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 the eddy determines the efficiency of the system. An 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 the eddy with low priority, and when they are returned to the eddy from an operator they are given high priority which ensures that tuples flow completely through

the eddy before new tuples are consumed from the inputs, ensuring that the eddy does not become "clogged" with new tuples.

Eddies have the limitation that they can be used efficiently when we favor join algorithm with frequent moments of symmetry, adaptive or non-existent barriers and minimal ordering constraints that are generally needed in various join algorithms such as Merge Join and Nested Loop Join. Thus they are effective only in Ripple Join. Their implementation is not fully dynamic. They still make use of some static mechanisms like "pre-optimization" "phase, choices of join algorithm and access methods. Resources are often not utilized properly in not so promising alternatives like implementing sort-merge join or other joins which do not satisfy the requirements for the eddies to be most effective. They should further use parallelism and adaptavity available in Rivers. Reoptimizing queries with intra-operator parallelism requires repartitioning data. But there is no efficient technique so far for adaptively adjusting the degree of partitioning for each operator in the query plan.

They want to apply their work to the generic space of dataflow programming. These include applications such as multimedia analysis and transcoding, and the composition of scalable, reliable Internet services. They want to use eddies as the main scheduling mechanism and rivers to serve as a generic parallel dataflow engine in that environment.

3.6 Dynamic Regrouping of Continuous Queries

They have proposed an approach for incremental grouping to efficiently group new continuous queries without having to regroup existing queries thus significantly

reducing the cost of execution. They have also proposed another approach called dynamic regrouping to increase the overall quality of regrouping which otherwise would have deteriorated by continuously adding and removing queries from the group statically. This dynamic approach increases the overall performance of the system. Their regrouping method, when applied in conjunction with the incremental grouping, obtains a reasonable improvement over the incremental grouping method at a low extra overhead in regrouping time. They consider multiple query optimizations as opposed to single query optimization and their regrouping mechanism [14] can handle newly arrived queries. The incremental grouping in conjunction with dynamic regrouping results in a high quality grouping at a fairly low cost. It can optimize large continuous workload and hence can be applied to a large-scale system. Regrouping is done very efficiently and does not impose significant burden on the system.

It maintains intermediate files incrementally by materializing the results, which avoids re-computation of the entire plan when any failure occurs. This model is quite simple and introduces simple metric to evaluate cost estimation called update frequency, which at any node is the sum of the update frequencies of all its children. Delete operation in global query optimization is not at all complex. The node count is simply reduced by 1 and the rest of the tree is automatically rearranged.

Incremental group optimization attempts to find the optimal solution to the new query submitted from all possible solutions. The overall cost for the new query is the sum of the costs of all new nodes added. They run top-down local exhaustive search, to find an optimal incremental plan for a new query. In dynamic regrouping algorithm,

they construct links between existing nodes and nodes that were added since the last re-grouping and then a minimal weighted solution is found from the current solution by removing redundant nodes.

This algorithm assumes that the amount of physical memory available is infinite and all nodes can fit in the physical memory. But this is not true when the number of installed continuous queries becomes very large. It makes use of update frequency for estimating the cost of that node which is an approximate method as the accurate method for computing cost is very difficult. The algorithm developed for an incremental plan is not efficient as it tries to find all possible sub query plan in an exhaustive top down manner to check whether a sub query node exists.

3.7 Niagara CQ

They have developed an Internet-scale continuous query system, which supports millions of queries using group optimization on the assumption that many continuous queries on the Internet will have some similarities. Previous group optimizations were not highly scalable as they could group only a small number of queries at the same time. A new “incremental grouping” methodology that makes group optimization more scalable than the previous approaches was proposed which can be applied to very general group optimization methods. NiagaraCQ [15] groups continuous queries based on the observation that many web queries share similar structure. In this system, both timer-based and change-based continuous queries can be grouped together for event detection and group execution, a capability not found in other systems. Incremental evaluation of continuous queries, use of both pull and push models for detecting

heterogeneous data source changes and a caching mechanism assist in making the system scalable.

Grouped queries can share common computation, tend to fit in memory and can reduce the I/O cost significantly. Grouping on selection predicates can eliminate a large number of unnecessary query invocations. They use an incremental group optimization strategy with dynamic re-grouping. New queries are added to existing query groups, without having to regroup already installed queries. They also use a query-split scheme that requires minimal changes to a general-purpose query engine.

NiagaraCQ caches query plans, system data structures, and data files as all information required by continuous queries and intermediate results will not fit in memory by considering the scalability of the system. Grouped query plans tend to be memory resident since we assume that the number of query groups is relatively small and saves lots of disk I/Os.

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

A prototype version of NaigaraCQ includes a Group Optimizer, Continuous Query Manager, Event Detector, and Data Manager. Incremental group optimization support queries containing only selection and join. They should share computation for expensive operators, such as aggregation. “Dynamic regrouping” is another interesting future direction they may explore.

CHAPTER 4

DESIGN ISSUES FOR A DSMS

4.1 DSMS SCHEMA

In traditional DBMSs, “schema” refers to the organization of data in relational databases, where data is contained in tables. Schemas are used for describing a database in terms of names and the characteristics of the data items. Although the definition of schema in streams is similar to that of a conventional DBMS, it describes continuous, unbounded and time varying streams instead of describing fixed tables. Stream schema consists of various attributes (or fields) and each attribute of the stream is described by its name, data type and position within the stream. The following section explains how such schemas are defined:

A DSMS’s schema stores complete information about all streams supported by the system. A new stream will not be recognized until it is registered with the system. This involves storing a new stream definition in DSMS schema. All streams have their schema information maintained in persistent storage so that it can be recovered in the event of a system crash. The data structures used for storing schema information can grow and shrink dynamically, which provides complete flexibility for addition and deletion of schema.

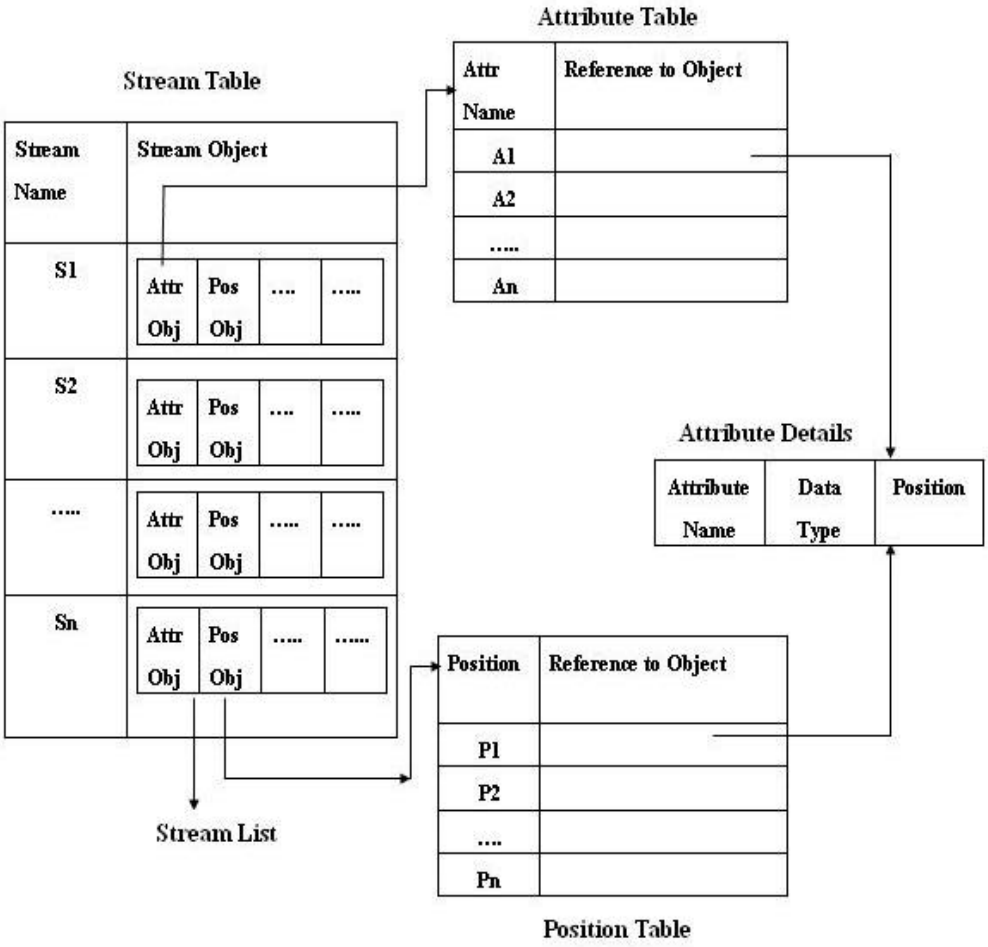

Figure 4.1 DSMS Schema

From the Figure 4.1, it is observed that stream names are stored as keys with their corresponding values as lists which in turn contain references to attribute and position tables. The attribute table describes attributes of a stream by its name, data type and position in schema while the position table provides the same information based on positions and hence attribute details can be accessed by specifying either attribute name or its position in the corresponding stream.

4.2 Buffer

Buffers are the intermediate storage structures used by operators. Buffers connect all operators in a query tree. An operator reads a tuple from an input buffer, processes it and passes output to the output buffer. Buffers are implemented as a queue. They support two basic operations for queue management viz. enqueue and dequeue.

4.2.1 Buffer Types:

Buffers are of two types:

- Bounded, and
- Unbounded

Bounded Buffer:

A bounded buffer has an upper limit on the number of elements it can store, which can be specified while instantiating it. It can be modified as and when the need arises. When the specified limit is reached, successive elements are stored in disk preventing any loss of data. Since limited memory resources are available for use by the system, this feature is useful in controlling the buffer size as a part of certain buffer management policies.

Unbounded Buffer:

In contrast to bounded buffers, an unbounded buffer continues to grow until main memory is exhausted. There is virtually no limit on the number of elements that can be stored. For initial implementation, experimentation and testing of DSMS, unbounded buffers were used heavily.

4.2.2 Buffer Access by Operators:

Buffers are shared among multiple operators as shown in Figure 4.2. Since operators can read elements from shared buffers independently, each operator maintains its own *reading pointer* that points to next element to be read.

In addition, each buffer internally maintains a *common pointer* for all operators that points to the *latest* element read by all operators. All elements including and prior to the common read element are safely discarded thus creating main memory buffer space. These buffer spaces can be filled by reading elements from the disk in the order in which they were stored. To accomplish this, the buffer incorporates minimal persistent logic to store and retrieve elements to and from the secondary memory as and when needed.

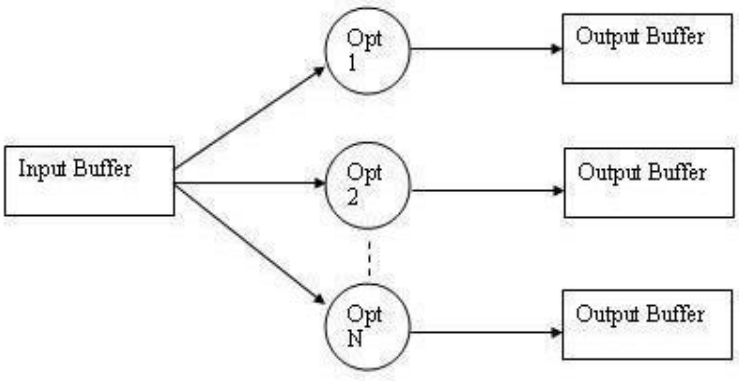

Figure 4.2 Buffers and Operators

4.2.3 Buffer Operations

Dequeue: Dequeue removes the top element from the buffer. If the top element is the last element, the buffer becomes empty. It would be preferable to suspend all

operators waiting on the buffer to save CPU cycles. Hence, operators attempting to read from an empty buffer are suspended. An element is dequeued only if all sharing operators read it. Once an element is removed, elements can be brought from secondary storage to fill the main memory buffer space.

Enqueue: A new tuple can be added to the buffer using an enqueue operation. If the buffer was empty before, it sends a resume signal to all operators upon which they are placed in the ready queue of the scheduler. If the buffer is bounded and the upper limit has already been reached, new tuple are added to secondary memory.

An element can be read by multiple operators simultaneously but can be written by only one operator at a time to maintain data consistency. Enqueue and Dequeue operations must be synchronized to ensure correctness as the same location may be accessed by both operations at the same time.

4.2.4 Persistence Logic:

Bounded buffers can store a limited number of elements as set by its upper bound. If the data arrival rate increases the data consumption rate, a bounded buffer would soon be exhausted. To prevent loss of data, incoming elements must be stored in secondary memory. This data can be read into main memory as and when buffer space is released. This functionality is provided in the Enqueue operation that writes elements in a file sequentially [16] to ensure that they are read in the main memory in the order of their arrival. This ordering is essential for windowed operators as they expect tuples to be timestamp ordered in order work correctly.

Since I/O operations are expensive, it is not recommended to fetch tuples from secondary memory each time a tuple is dequeued. The dequeue operation starts a separate thread to read elements from secondary memory only if n% or more tuples are removed from main memory buffers. For all the experiments, n is set to 50.

4.3 Stream Operators

Operators of traditional DBMSs are not designed to 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 and demand a re-examination of the design of conventional operators for handling 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. Thus we realize the need to design and develop *Stream Operators* by considering stream characteristics, which can accommodate the dynamic aspect of query plan generation and scheduling for processing streamed queries.

A query-processing graph is comprised of operators connected via queues as shown below:

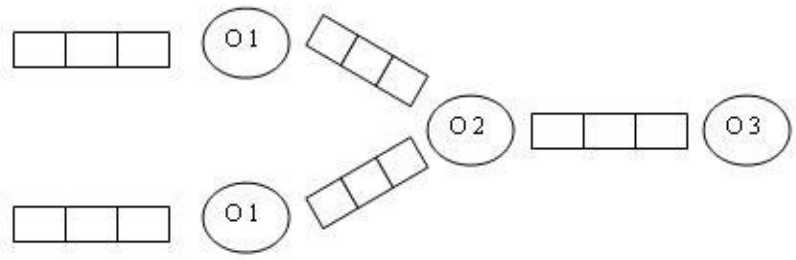

Figure 4.3 Operator Buffer Query Tree

Every operator has at the most two input queues but can have any number of output queues. Streaming data are buffered in these input queues. An operator reads data from these input queues takes the appropriate actions and generates results that are buffered in the output queue. The visualization is provided in Figure 4.3. The output queue of one operator becomes a shared input queue for other operators waiting at the next higher level of the query tree (a tree generated based on the query input which decides the order in which operators are instantiated) for consuming input.

4.3.1 Operator Design:

It is absolutely essential to design an operator as a manageable unit so that it can be controlled by different entities in a system such as the user, the buffer manager and the scheduler. An operator is instantiated dynamically along with its input and output queues. Every operator is implemented as a separate thread, which is scheduled by a scheduler, placed at the head of scheduler's ready queue. Scheduling decisions are purely based purely on two important properties possessed by an operator, which are as follows:

4.3.1.1 Priority:

The majority of streaming applications demand real time output for different kinds of queries with varying requirements. To suit these requirements, priorities are associated with queries indicating their urgency. During query execution, priorities can be changed either by a scheduler, a run-time optimizer or a user to increase the overall performance of the system. The following are the entities, which are likely to change the priorities.

User:

Queries with strict deadline must be completed before the specified time to be meaningful. These queries and all operators constituting these queries are given higher priority by the user. Consider a snapshot query (one time query) that needs to be evaluated immediately. For example, *"Retrieve the highest temperature recorded between 10p.m and 10:05 p.m."*. Such snapshot queries generally enjoy higher priority over long running queries. User can also change priorities of queries dynamically at run time.

Scheduler:

The scheduler plays an important role in improving the overall efficiency of a system with regard to memory utilization, tuple latency, run time resource utilization, query throughput and quality of service requirements. All these parameters are controlled by adjusting the operator's priority. For example, an operator at the bottom of a query tree is assigned higher priority as compared to an operator at the top, since base operators are flooded with input streams (Leaf nodes are expected to handle huge

streams of data since they are closer to the source. As we traverse the tree from leaf to root, the amount of data to be handled reduces drastically due to the selectivity of intermediate operators). It may also change the priority based upon the resources allocated to the operator. For example, if an operator is given higher priority but all the resources needed for its operation are not available then its priority may be reduced. Also it is meaningful to assign higher priority to an operator with higher fan-out over one that feeds its output to a few nodes or none. Priorities can also be assigned based on operator complexity and functionality. *Join* may need more time quantum than *Select*, as its operation is more complex and time consuming.

Run time Optimizer:

Run time Optimizer may not change the priority of an operator itself but directs the scheduler to change the priority of an operator. It verifies whether the desired QoS is met and accordingly asks scheduler to change the query plan and/or priorities associated with the query (which in turn affects priorities of corresponding operators). For example, if a response time (end--to--end query processing time) is x and QoS is y, any ideal system would expect x < y. The run time optimizer continuously monitors the output and compares it with the defined QoS. If the desired QoS is not met, it tunes the system to achieve the desired QoS requirements. One of the tuning parameters is the priority.

4.3.1.2 State:

Operators are schedulable entities. They have different states of execution during their lifetime. Operators can be in one of the following four states during their course of execution.

- Ready
- Run
- Suspend
- Stop

Transition from one state to another is determined by the operator's priority, availability of resources and scheduling schemes used. The state transitions are shown in Figure 4.4 and described below:

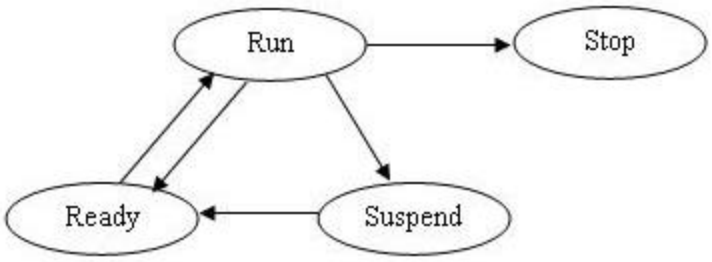

Figure 4.4 Operator State Diagram

Ready:

When the user submits a query, constituent operators and their input and output queues are initialized and instantiated upon which they are placed at the end of

scheduler's ready queue. An operator previously suspended transitions into this state when all resources needed for its execution are available.

Run:

An operator goes into this state when it is selected by the scheduler for execution. This is the state in which operator performs its actual operation. Operator can switch to the *ready* state if its assigned time quantum has not elapsed or it may be *suspended* if all resources needed for its execution are not available.

Suspend (Wait):

A running operator can be suspended for the following reasons:

- It may be pre-empted by a higher priority operator.
- All resources needed for its operation are not available. (Input queues are empty).

Stop:

This state indicates that all queries requiring this operator are completely processed. The operator is removed from the system when it is stopped.

It is essential to provide APIs for defining operator state, operator priority, scheduler instance and output queues. These commonalities have been identified for all operators and have led to the design of an operator hierarchy consisting of a generic parent operator and specialized child operators. The parent operator provides APIs to support the functionality mentioned above while the children possess additional functionality besides those inherited from the generalized parent operator. This design avoids code replication to multiple operators and provides easy development and

maintenance of code. The output queue is also defined in the base operator but input queue is defined in specialized operators. This is because all operators except "Join" needs only one input queue. The output queue is defined in base operator because any operator can have any number of output queues associated with it.

The operator hierarchy can be represented as shown in Figure 4.5:

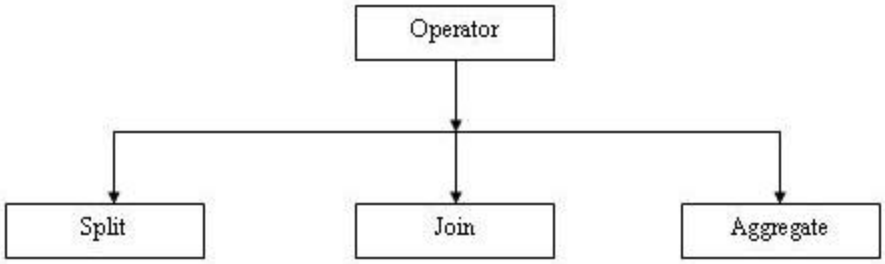

Figure 4.5 Operator Hierarchy

4.3.2 Operator Types:

Based on whether operators need window bounds for their computation or not, they are further classified as:

- Non Windowed Operators
- Windowed Operators

Non windowed operators:

They do not depend on windows for their computation. These operators work on one tuple at a time and generate the required output. They are non-blocking operators

(operators are said blocking when they cannot produce output unless a complete set of input is available). Split is a non-windowed operator.

Windowed Operators:

As streaming data is potentially unbounded in size, blocking operators, such as Aggregate and Join may block forever if their input bounds are not defined. Hence the concept of a window is introduced which produces a bounded set of tuples from unbounded streams. Once a window is processed, the window slides so that the operation can be performed for the next set of data and is repeated until the query is ended. The following are the windowed operators supported by our DSMS:

- Aggregate
- Nested Loop Join

4.3.3 Non Windowed Operators

4.3.3.1 Split Operator

Select evaluates only one condition while split evaluates multiple conditions. Split has one input queue and multiple output queues -- one for each condition. The need for a Split operator was identified to logically divide the streams based on application logic. One application of the Split operator would be to divide a single composite (heterogeneous) stream into multiple homogeneous streams (all elements in each stream are of the same type). A list of conditions is maintained and the incoming tuples are subjected to condition evaluation sequentially. If the tuple satisfies the condition, it is sent to the corresponding output queue else the same tuple is evaluated for the next condition. This is repeated until all elements in the condition list are

checked or the tuple satisfies one of the conditions after which the next tuple is considered for evaluation. If a tuple doesn't satisfy any of the conditions, it is put in the default output buffer. The Split operator is better understood using Figure 4.6.

4.3.3.1.1 Design Alternatives:

The complexity of this operator lies in the condition evaluation and its efficiency is proportional to the efficiency of the tool used for condition evaluation. The conditions have to be interpreted at run time and cannot be compiled into code. We have tried to implement our own condition evaluator but encountered several problem relating to cost, complexity and efficiency. Moreover we realizes that it was not as powerful and as efficient as FESI (Free Ecma Script Interpreter) which has its own condition evaluator that supports virtually all any Java expression consisting of relational, logical and many other operators. FESI has been chosen because it has already been tested for correctness and efficiency. FESI reduces code complexity and provides higher level of abstraction for evaluating a condition. FESI APIs for condition evaluation alleviates us from building our own expression evaluator.

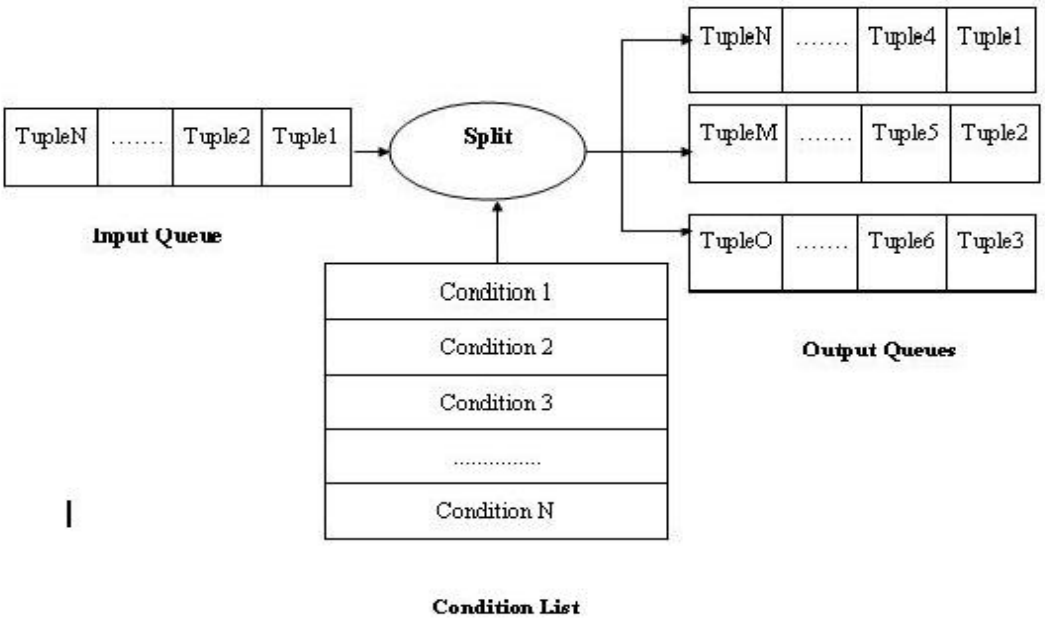

Figure 4.6 Split Operator

The Split operator has multiple conditions to evaluate. Every time a condition is evaluated, a schema needs to be accessed in order to replace the attribute name mentioned in the condition string with the corresponding attribute position to read field values at that position from the incoming tuple. There are two alternatives for handling this situation. One of the alternatives is to access the schema every time a new condition is evaluated. If there are 'N' conditions, the schema needs to be accessed 'N' times. The other alternative is to access schema just once and subsequently sets all the attributes of the input stream with the corresponding attribute positions. Accessing a schema is a time consuming operation and hence the latter is preferred wherein all the attributes of input stream are set regardless of whether they are needed in the condition list or not.

4.3.3.1.2 Interaction between Buffer and Scheduler:

Split operator communicates with two other important modules, namely, the buffer manager and the scheduler. It registers itself with its input and output queues (buffers), and with the scheduler to facilitate communication among them. The scheduler starts or resumes the operator if it was not already started and runs it for the assigned time slice. Its interaction with the buffer is important and introduces the interesting issue of operator suspension and resumption. CPU cycles are wasted when an operator attempts to read from an empty buffer. It is appropriate if the operator is suspended when the resources are not available. Either the buffer can suspend the operator or the operator can suspend itself under these circumstances. Buffers are shared by many operators and if the buffer takes the responsibility of invoking operators then all operators are simultaneously suspended and awakened (placed in the ready queue). This approach is efficient but entails extra processing responsibilities for the buffer manager, which already performs some complex tasks. The other alternative is to provide this control to the operator itself thus causing every operator to suspend independently. While not as efficient as the first approach, it does help to reduce the load on buffers. This trade-off was deemed necessary and control was assigned to operators rather than buffers.

The algorithm for *Split* is as follows:

```
While (end time of query is not reached) {
        If (input queue is not empty)
                Read tuple from input queue.
```

While (all conditions are not evaluated)

Read the next condition string.

Set operands of the condition string by the corresponding tuple field values to generate modified condition string.

Evaluate the modified condition string using Fesi Interpreter.

If (condition is satisfied)

Send tuple to the output queue associated with the present condition.

Break.

If (none of the conditions are satisfied)

Send tuple to default output queue.

Dequeue read tuple

Else

Wait on the input queue (operator suspended)

}

4.3.4 Windowed Operators

Operators such as Select, Split and Project work on a single tuple at a time and do not need complete set of input tuples to be available. However the Join and Aggregate operations such as Average, Sum, Min and Max need a complete set of input before they can produce any output. In streams, data arrives continuously and blocking

operators may block forever waiting for an unbounded stream to terminate. The solution to this problem is to define a window that marks the beginning and end of input bounds. All tuples falling within the window becomes the input set for blocking operators. The results are produced incrementally at the end of every window. The window itself can be defined in a number of ways. The following are the possible combinations of windows [17] which can be defined for a query. They are as follows:

4.3.4.1 Window Types

There following are the different types of physical window viz.

- Snapshot window
- Landmark window
- Sliding window
- Reverse landmark
- Reverse sliding

A sliding window has two types: overlap and disjoint sliding windows.

Snapshot Window:

This is a single fixed window. Its beginning and end time are fixed, as shown in Figure 4.7. Queries using a snapshot window produce output only once at the end of window. These are also called one-time queries. Once the output is produced, the query is removed from the system. Example of such a query: *Select all devices that were turned on between 5 P.M. and 6 P.M on Jun5 2003.*

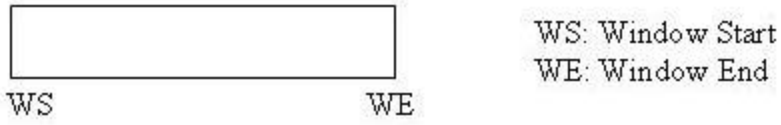

Figure 4.7 Snapshot Window

Landmark window

This window has a fixed begin time and a variable end time. Windows are continuously formed until either the query end time is reached or the query is terminated explicitly, as shown in Figure 4.8. An example of such a query is: *Continuously select all the passengers that entered at the airport from 5 P.M. on June 5, 2003 every hour.*

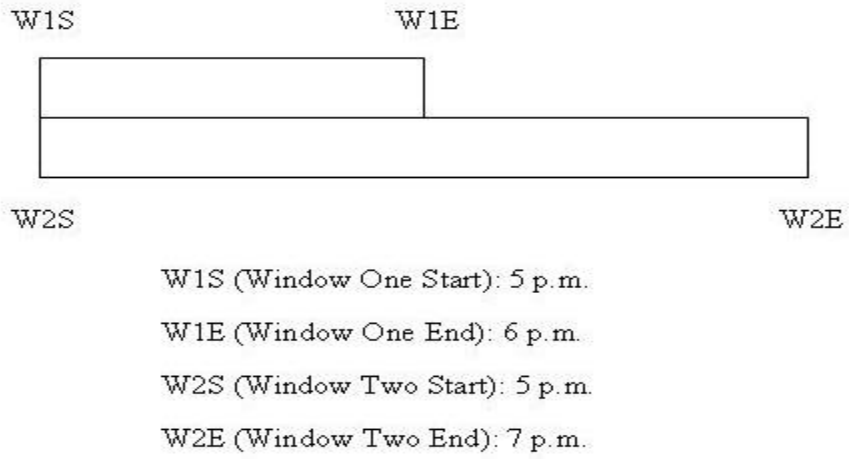

Figure 4.8 Landmark Window

It can be observed that window is continuously expanding in the forward direction as the start time is fixed. Initial size of the window is 1 hr, between 5 P.M and

6 P.M. Next window size is 2 hrs, between 5 P.M and 7 P.M and this runs indefinitely. These queries are also called long running queries. We can also give the time to end the query. For example: *Continuously select all the passengers entered at the airport from 5 P.M. on June 5, 2003 every hour until 6 A.M on June 6 2003.* Thus *6 A.M on June 6 2003* is the terminating time for the query.

Reverse Landmark Window:

This classification is a mirror image of Landmark Window explained above. This window has its start time fixed but end time moving in the reverse direction. See Figure 4.9. Example of such a query is: Continuously select all passengers entered at the airport starting from 6 p.m. on June 6 2003 to 3 p.m. on June 6 2003 every hour.

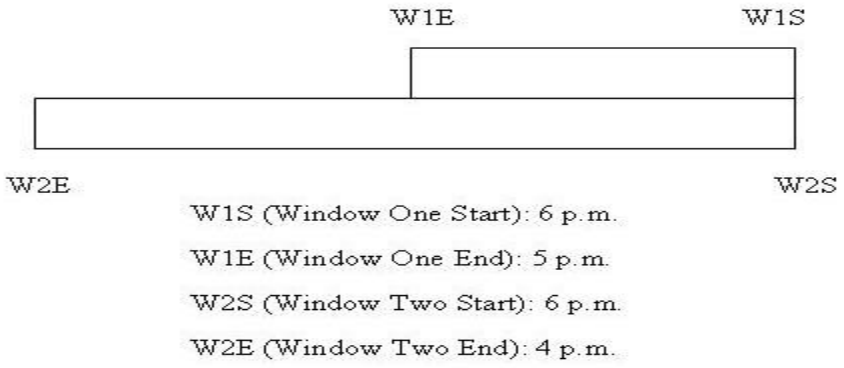

Figure 4.9 Reverse Landmark

Sliding Window:

This window has both its end points moving and hence the name sliding window. This is also a long running query.

Sliding windows are again divided into 2 types:

- Disjoint sliding window and Reverse Disjoint
- Overlap Sliding window and Reverse Overlap

Disjoint sliding window: In this, successive windows never overlap. The endTime of the current window becomes the beginTime of the next window, as shown in Figure 4.10. Thus two successive windows never overlap and hence the name disjoint sliding window. An example of disjoint sliding window is: Show me the common items purchased in 2 departmental stores every hour starting from 5 P.M onwards.

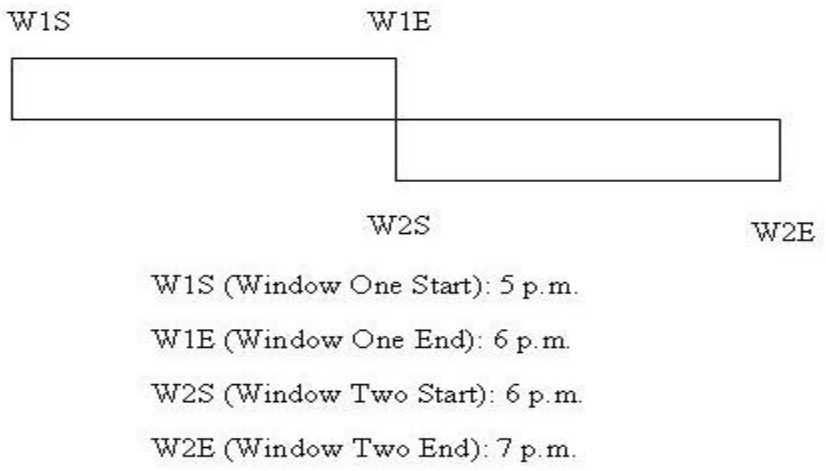

Figure 4.10 Disjoint Sliding Window

Reverse Disjoint: In reverse disjoint, window shifts in the reverse direction while respecting the disjoint constraint, as shown in Figure 4.11. It is the mirror image of a Disjoint Window. An example of a disjoint sliding window is: Give me the common items purchased in 2 departmental stores every hour starting from now (say now is 6 p.m. on June 6, 2003) to 1 p.m. on June 6, 2003.

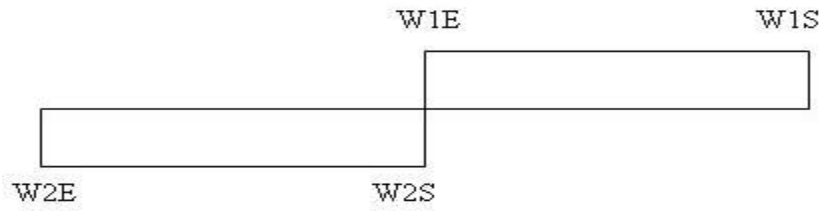

Figure 4.11 Reverse Disjoint

Overlap sliding window: In this variation, two adjacent windows may overlap, as shown in Figure 4.12. The start time of next window is always lower than the end time of current window. An example of an Overlap windowed query is: *Give me the average temperature recorded for every one hour by a thermostat every 10 minutes from now.*

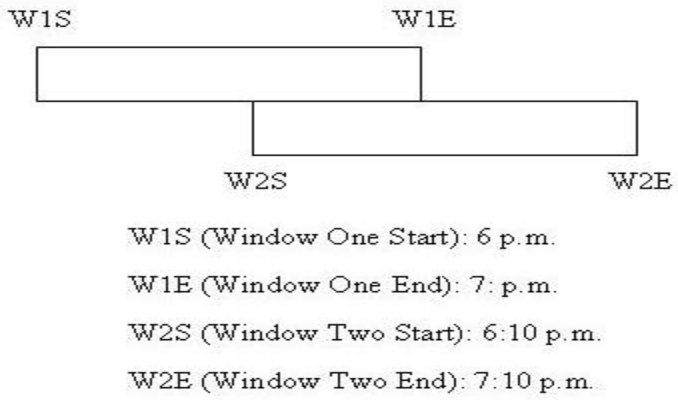

Figure 4.12 Overlap Sliding Window

Reverse Overlap Sliding: It slides in the reverse direction, as shown in Figure 4.13. It is the mirror image of the overlap sliding window. Example of Overlap

windowed query is: Give me the average temperature recorded for every one hour by a thermostat every 10 minutes from now (6 p.m. on June 6, 2003) to 4 p.m. on June 6, 2003.

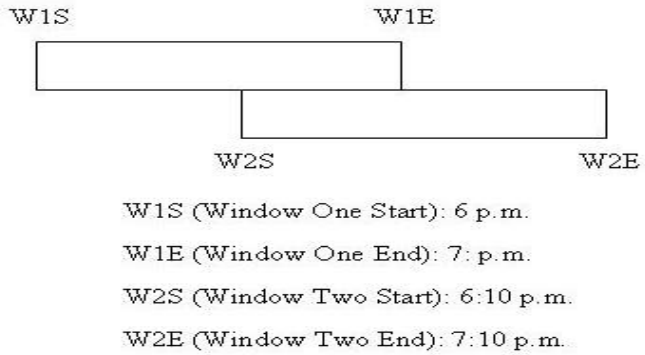

Figure 4.13 Reverse Overlap Sliding

4.3.4.2 Representation of windows

All types of windows except Snapshot can move in both directions. They can expand in forward as well as in reverse direction. Thus we realize the need for windowed representation such that all types of windows can be uniquely represented and identified. We have proposed the following representation, which takes care of all possible combination of windows.

```
if (windows == Physical) {
        beginWindow
        endWindow
        hopSize (startTime, endTime)
        endQuery
```

}

where beginWindow, endWindow, startTime, endTime and endQuery of Physical Window are absolute or relative time. Relative time can be given by using the keyword Now () where Now () returns the current system time.

Consider the query example for disjoint sliding window. *Give me the common items purchased in 2 departmental stores every hour starting from 5 P.M until 9p.m.*

The windowed representation for the same is as follows:

Window = = Physical {

Begin window = 5 p.m. (June 5, 2003)

End Window = 6 p.m. (June 5, 2003)

Hop Size (1 hr, 1 hr)

End query = 9 p.m. (June 5, 2003)

}

Consider the query example for reverse landmark window. *Continuously select all passengers entered at the airport starting from 6 p.m. on June 6 2003 to 3 p.m. on June 6 2003 every hour.*

The windowed representation for the same is as follows:

Window = = Physical {

Begin window = 6 p.m. (June 6, 2003)

End Window = 5 p.m. (June 6, 2003)

Hop Size (0 hr, -1 hr) (-1 indicates backward moving window)

End query = 3 p.m. (June 6, 2003)

}

Reverse (or backward) windows can be used for historical queries (when one wants to query on past data). Currently only forward queries are supported. However this design has the flexibility to support backward (or reverse) queries as well.

4.3.4.3 Nested Join

This join algorithm can be compared with classic nested join of RDBMS in which for two joining relations, every element from one relation is compared with all elements in the other relation to check whether the join condition is satisfied. Whenever a match is found, tuples are joined and produced at the output. This algorithm does the same but operate on *Streams*, as shown in Figure 4.14. Since join is a blocking operator (blocking operators cannot produce output until entire set of input is available) it needs a window for its computation, which defines its input boundaries. It produces results continuously which are consumed by higher operators. It does not wait for the entire window to elapse to produce the output. This operator registers itself with query window class, which defines window bounds for input streams. It also has APIs for generating sliding and disjoint windows based on window specifications and controls their movements accordingly. It detects query termination and declares the end of computation. These windows are not defined at the query level and hence different operators of the same query may be working on different windows at the same time. Nested join is a binary join with two input queues associated with it. These queues (buffers) are populated by streams, which may be same or different to feed input to the join operator. When a new tuple arrives at one input queue, it is joined with all the

tuples falling in the current window bound of another queue that satisfy the join condition. This action is atomic which ensures duplicate avoidance at output. It is not only essential for input tuples to be timestamp ordered but also the output produced by joining input tuples must be timestamp ordered to ensure that higher windowed operators which are continuously consuming inputs (which are the output from lower windowed operators) also produce correct results. Implementation section explains how timestamp ordering is respected for join output.

Design Alternatives:

Two threads instead of single thread:

This operator could have been implemented using two threads viz. left thread and right thread. Left thread reads tuples from left input queue and scans all tuples in the right input queue to find the matched tuples and the right thread behaves analogously. The idea of using two threads was to achieve some degree of parallelism. But if two threads are not synchronized and join computation is not done atomically, duplicate tuples will be produced. Atomic action for left thread involves reading tuple from the left external buffer provided it has a lower timestamp from its corresponding right tuple, computing join on the tuples residing in the right internal buffers (every join operator has two internal buffers, one corresponding to each external buffer. Join computation is done on these internal buffers) and eventually placing itself in the left internal buffer. To ensure the output to be timestamp ordered, threads may have to block at the input unless it finds a corresponding tuple with a higher timestamp. Also left thread had access to right external buffers and right thread had access to left

external buffers, something that can be achieved using a single thread. Thus the entire purpose of having two threads for achieving some degree of parallelism is defeated. Hence the final version of join operator is implemented using a single thread.

If a tuple with timestamp 't1' arrives at left input queue prior to tuple with timestamp 't2' at right input queue, such that 't1' > 't2', 't1' should block at its input queue (input queue is external queue) and allow 't2' to perform join to ensure timestamp ordering for output tuples. Another alternative is to compute join without blocking at input. This would produce correct results but the output may not be timestamp ordered. In order to get output sorted by timestamp, they may be subjected to sorting algorithm making the overall join computation expensive. Hence this approach is ruled out.

Determining window bounds:

In order to work correctly, join expects tuples to be timestamp ordered. Whenever it reads a tuple whose timestamp is greater than the current window bound, it marks the end of window boundary assuming that tuples following it will also fall beyond the current window as tuples are timestamp ordered. Another alternative is to compute the difference between the timestamp of current tuple and the timestamp of current start window. If the difference is less than the window width, tuple falls in the window else it is outside the current window. This approach does not need tuples to be timestamp ordered. We are following the first approach and expect tuples to be timestamp ordered.

Types of Nested Join:

There are two versions of nested join: *with reuse* and *without reuse.* They are meaningful only for overlapped windows.

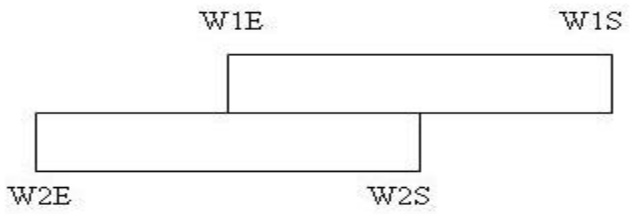

Figure 4.14 Nested Join

4.3.4.3.1 Nested Join with reuse:

Here the startTime (W2S) of next window is lower than the endTime (W1E) of current window. The first window is processed completely but the result of common time slice between current and next window (W2S and W1E) is materialized for the computation of next window. Prior to next window processing, results materialized by current window are re-copied to the output queue for the next window to ensure that next window result is timestamp ordered. In the next window computation, left window joins tuples residing between W1E and W2E in left input queue with tuples falling in W2S and W2E of right input queue and right window behave analogously. This computation is appended to previous materialized result to produce complete and correct next window output. Since common computation is shared in two successive

windows, this shade is called Nested Join with Reuse. The effect of this join on memory and processing cost is explained as follows:

Memory: In this algorithm, tuples are discarded from external buffer as soon as they are consumed since they are stored internally in the operator itself for the computation of next window and duplicates are avoided, as the common time slice is not recomputed. External buffers are flooded with stream data, which should be consumed and discarded at rapid rate to avoid disk operations. This shade does exactly the same at the expense of operator's internal memory.

Processing cost: Processing logic is efficient and simple to implement. The beauty of this shade is that it does not have to remember and revert back to the past on external buffers for the computation of next window. External buffer pointers always move forward as elements are read only once. Overlapped region is not recomputed which increases throughput enormously when shared portion is significant in a large window.

4.3.4.3.2 Nested Join without reuse:

This is another shade of nested loop join, which does not make use of overlapped region of two successive windows and computes every window independently of each other. Hence current window does not store the result of common computation for the next window. It is not efficient with respect to memory and computation, which can be explained as follows:

Memory: Since the windows are computed independently, tuples, which are already seen by current window, cannot be discarded. Only tuples prior to W2S can be

safely removed but tuples falling between W2S and W1E cannot be removed, as they are needed for the computation of next window. In without reuse, internal memory is cleared on processing current window, since keeping elements in internal buffer would result in duplicates as tuples falling in common time slice are recomputed. Thus without reuse leads to memory wastage and to make matter worse external buffers continue to grow eventually leading to disk operations if the overlap region is significant in a large window for processing bursty streaming data arriving at rapid rate.

Processing cost: Processing cost is significant as the overlapped region is recomputed. Join is an expensive operation and re-computing significant portion over large and multiple windows decreases response time significantly. Also computation logic is more complex as it has to remember and access past information on external queues for next window (to the past of external queues) computation once the end of current window is reached.

The algorithm for Nested Join (with and without reuse) is as follows:

```
while (operator is alive) {
        if (either left or right input buffer is empty)
                suspend join operator.
        else {
                fetch left tuple from left buffer.
                fetch right tuple from right buffer.
                if (timestamp of left tuple < timestamp of right tuple) {
                        if left tuple falls in the current window {
```

```
                computeLeftJoin
            }
            else {
                leftWindow Processed
            }
        }// if (timestamp of left tuple < timestamp of right tuple)
    else { // start processing right tuple
            if right tuple falls in the current window {
                computeRightJoin
            }
            else {
                rightWindowProcessed
            }
        }// else (timestamp of left tuple < timestamp of right tuple)
        if (leftWindowProcessed && rightWindowProcessed) {
            purge input buffers for discarding old tuples.
            set window buffer pointers (depending on with or without
            reuse).
            generate next window.
            if (next window > end query)
                stop the operator.
        } // if (leftWindowProcessed && rightWindowProcessed) {
```

}// else

} // while (operator is alive)

computeLeftJoin and computeRightJoin performs join operation on left and right tuple by removing common joining attribute from the right tuple and assigning lower timestamp followed by higher timestamp as the last two fields to the resultant tuple to ensure that output is timestamp ordered.

4.3.4.4 Aggregate Operators

Aggregate operators are blocking operators and hence need window for their computation. They operate on a window worth of data and produce output at the end of every window. Aggregate operations supported by this system are sum, min, max, average and count. Aggregate operators register themselves with a query-window class. This class is responsible for creating and manipulating windows. It defines the window boundaries and controls the forward and backward movement of windows based on hop-size by creating next and previous windows. Aggregate operator can see windows independently of other operators in a system since query-window APIs can be called at instance level. Its interaction with scheduler and buffer is similar to Split.

Design Alternative:

As explained above, there is a generic operator class and all other operators are derived from this generic class. Earlier it was thought to have another class, which resides between generic parent operator class and specialized children operator classes. This intermediate class was termed as ‘Aggregate’ class, which could have supported generalized functionality for all aggregate operators. This included, providing methods

for associating input and output queues, defining a query window to work on, calling purging logic to discard unwanted tuples from an input queue and setting a field for aggregation. But this option was ruled out for the following reasons:

1. It was different from the general operator hierarchy consisting of two levels. Thus with three levels of operator hierarchy, natural flow of computation and program logic would have lost.
2. It does not significantly reduce code complexity and hardly adds to efficiency.

The algorithm for Aggregate is as follows:

```
While (query is alive) {Read tuple from input buffer.
If (tuple falls in the current window)
        Perform necessary aggregation on the specified field.
Else {
        Output aggregation result (current window has elapsed).
        Purge elements, which can be safely discarded.
        Compute the next window.
        If (the end time of next window is greater than end query time)
                Stop the operator.
}
}
```

All aggregate operators are reusing the common computation (overlapped region) since it saves processing time. For average operation, two variables are used to keep track of

sum and count of aggregate fields as and when tuples are added. At the end of every window, average is computed.

4.4 DSMS Client-Server Model

DSMS is modeled as two tier client-server architecture with client defined as a requestor of services and server as the provider of services. Client provides a graphical user interface to allow users to request services from the server. Some of the offered services include generation of schema for new streams, processing a plan object (a data flow graph consisting of operators and their associated queues) and requesting definitions of already existing schemas. These requests need some processing at the client to make it protocol specific and the processed request is eventually sent to the server by following the protocol defined for client-server communication. Server is a powerful and complex program, which integrates and controls its various modules such as Instantiator, operators, buffer and scheduler to execute client requests in a timely manner based on quality of service specifications. Server response can either be used to display results to clients or they can be used to trigger events providing active support. The communications between Client and Server can be better visualized from Figure 4.15.

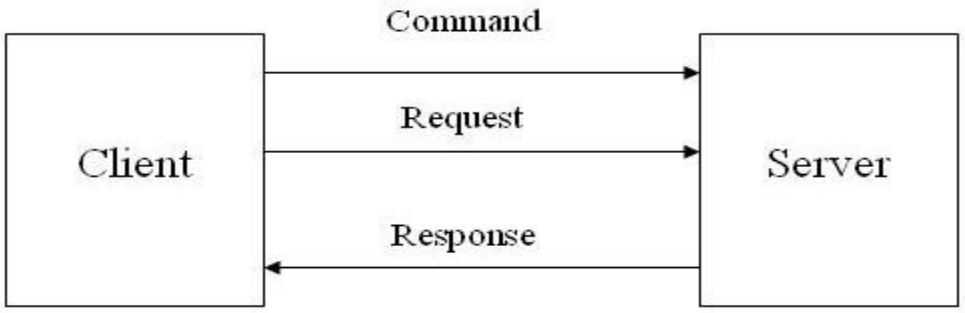

Figure 4.15 Client-Server Communication Model

Client-Server application communicates over the network using sockets. CORBA is another alternative that is generally used object components written by different vendors want to interoperate across networks and operating systems. Since our client and server are implemented in java, socket based communication is proposed as a less complex and straightforward alternative RMI allows programmers to distribute computing across networked environment. It defines a set of remote interfaces to create remote objects which client can invoke with the same syntax that it uses to invoke methods on local objects. As our server is not distributed, this no longer remains a suitable mode of establishing client-server communication.

4.4.1 DSMS Client:

Client is a simple program used for collecting user requests and presenting them to server for processing. Client can be of following types:

- Non-web based client: This client does not make use of web features and hence lack worldwide accessibility. Since DSMS is developed in java, user interface can be designed in Swings or AWT components to allow users to construct queries. It

may allow users to perform basic operations such as defining new streams, instantiating or stopping a query, deleting a schema, etc.

- Web based client: Web based interface is provided for constructing queries and submitting other requests. User constructs queries by moving across the web pages. Client may need some processing for requests, which are incorporated in a web server. DSMS client is a web-based client.

Operators need requests in a specific form to be instantiated. They also demand availability of schema definitions of their input streams to work correctly. Request modification and schema generation can be done at the client side or at the server side. Since server is more complex, these functionalities are provided at the client. This distribution of processing allows the client to offer a user-friendy environment and allows the server to be relatively less complex.

Client has following responsibilities:

1. It constructs a plan object (a data flow operator-queue graph) from the user input, which defines the order of operator instantiation depending on the direction of data flow. It is a sequence of operator nodes where each node completely describes the corresponding operator.

2. It provides user interface to accept request from clients. User may request server to stop a query, delete an existing schema, add a new schema and besides submitting a query. Quality of service specifications and priorities associated with queries can also be specified.

3. It participates with server in command driven communication protocol. Requests are sent and response collected from the server to be presented to users.

4. Client generates intermediate schema when operators in a query tree produces new streams. Join and Project always present new stream as former expands and the latter shrinks the base schema. Hence a new schema definition needs to be registered with the server to support the new streams being generated. Client has the responsibility of creating and registering new schema with the server.

5. An operator may take input from two different streams with one or more same attribute names. Client resolves these attribute names so that next operator uniquely identifies attributes in the resultant stream.

6. Client may also do some validation checks on the syntax and semantics of query submitted.

4.4.2 DSMS Server:

This program is responsible for executing user requests, and producing desired output. All stream management services are handled by the server. Server is mainly responsible for collecting, storing and processing unbounded streamed data in timely manner producing real time response to user queries. It integrates and instantiates various modules including operators, buffers and scheduler to ensure that computation keeps up with the data flow rate and quality of service requirements are respected as delayed response may be totally unacceptable. Some of the services offered by DSMS Server are as follows:

1. Addition and Deletion of Schema

2. Query Instantiation and termination

3. Operator initialization and instantiation

4. Accepting priorities and quality of service specifications for queries.

5. Associate input and output queues with the operators.

It is important to understand the protocol followed for client-server communication. In order to identify a request, client sends a unique command before sending the actual request. Once the command is received by the server, it expects specific request corresponding to the command from a client. Client then sends the request to be processed by a server. Server, based on the command received, processes the request and dispatches the response to the client. It performs the following functions:

1. Accepts command and request from a client, which describes the task to be carried out.

2. Retrieve all stream names and their schema definition so that user can pose relevant queries. Server response provides details about all available streams so that user can formulate queries accordingly.

3. Client can request the server to either start or stop a query. Client sends a plan object for query instantiation. Server initializes and instantiates operators constituting a query and schedules them for execution. Client may also want to explicitly terminate a running query in a system. The query is stopped which in turn stops all the operators associated with the query. It is then removed from the system.

4. It allows deletion and addition of schemas. It registers new stream and its schema definition. This may be used to support other stream producing sources. It deletes existing schema when application no longer needs it. This provides complete flexibility of changing base schema.

5. Associate input buffers with base streams. In a query tree, output queue of one operator becomes an input queue for the next operator. Hence buffer association of all intermediate operators in a query tree is defined. But buffer linkages with their corresponding streams must be explicitly given for base operators. This information is passed by a client to be stored in a server.

6. It initializes operator by reading operator data node which contains all initialization specifications and defines window boundaries for them. Initialization also involves associating input and output queues and binding to defined scheduler.

7. Start scheduler to schedule operators for doing necessary computation and generates result.

4.5 Scheduler

Scheduler plays an important role in improving the overall efficiency of system with regards to memory utilization, tuple latency, run time resource utilization, query throughput and quality of service requirements. All these parameters cannot be satisfied by a single scheduling scheme. For example chain scheduling [18] defined in the literature is superior to FIFO scheduling with respect to memory consumption while FIFO outperforms chain scheduling in terms of overall tuple latency. It is difficult to design an optimal scheduling strategy, which can dynamically change scheduling

algorithm to improve overall efficiency of the system, as continuous streams are unpredictable and bursty. Scheduling schemes of traditional DBMS are not used as they are designed for predefined and periodic task, which are completely different from the stream characteristics in which processing cycle varies continuously.

Tuple latency and query throughput are the most important criteria in processing streamed queries for providing real–time response. The scheduling scheme proposed in DSMS are *FIFO* in which tuples are processed in the order of their arrival and *weight based round robin* in which operators are scheduled round robin but with different weights (time slice) based on priority. Thus the only parameter, which can affect the overall performance of the system, is priority, which must be intelligently assigned to operators so as to create a perfect balance between memory space and processing time, still ensuring real time response to streamed queries. The scheduler is developed for real time systems, which attempt to execute the higher priority task with the maximum expected utility in order to meet quality of service requirements.

4.5.1 Parameters for Priority Assignment:

The criteria for assigning priorities to operators depend on the critical and sensitive parameters of the system which are enumerated below:

Memory: The arrival rate of data stream may exceed the data processing rate due to high volume and bursty traffic. These variations in data rate may buffer tuples in memory and it may even exceed total main memory causing the system to swap pages from the disk. This can also increase the overall response time since the waiting time of tuples in buffer increases as buffer size increases according to queuing analysis [19].

Situation become worse when join operator expect current window worth of data in main memory to be effective. Hence it makes sense to assign more time quantum to join, as they are more complex and time consuming than select which discards input tuples as soon as they are consumed. This non-uniform distribution of weights would cause join to be scheduled for longer period so that more tuples are processed and consumed from main memory buffers.

Query-Throughput: When memory is not a critical factor, we must emphasize on improving the tuple latency to maximize query throughput. Operators at leaves are closer to data sources, which are flooded with continuous and rapid data streams. Hence they should be scheduled more frequently to create buffer space for incoming tuples and to produce output to be fed to higher operators in a tree. Selectivity reduces input tuples for processing as we progress higher up in a tree. To improve the overall performance of the system, utilization of leaf operators must be higher than non-leaf operators and hence they should be scheduled more frequently.

CPU Utilization: As we know, an operator can feed its output to multiple operators. In order to improve the response time, such operators must be scheduled more frequently as operators with higher fan out ensure that other operators waiting on it are not blocked waiting for input, which improves overall system utilization.

Priorities can be assigned either by users as a part of query to indicate whether they need immediate service or they may be assigned by a scheduler depending on resource availability and quality of service requirements. Once the priority is assigned, scheduler ensures that the priority of an operator never falls below the initial assigned

priority. It may increase the priority to improve system performance and query execution throughput but it cannot decrease the priority from the initial assigned value. Scheduler is the highest priority thread running in a system that picks an operator for execution from the head of its ready queue, ascertain what processing is required and process them for the assigned time. Operators must be in a ready state in order to be scheduled. One of the following conditions may occur during the running state of the operator.

1. Operate may finish its execution prior to its assigned time quantum. It informs scheduler who removes the operator from ready queue and schedule next operator at the head of ready queue.

2. Operator may block waiting for the availability of resources. For example, its input queue may be empty upon which operator is suspended. It informs scheduler about its suspension, which in turn releases its execution and removes the operator from the ready queue. Whenever resources become available, operators are brought into ready state and placed at the end of ready queue.

3. An operator may not have finished its operation but its assigned time quantum is expired. The Scheduler suspends the execution of the operator and places its at the end of ready queue to facilitate fair scheduling.

4.5.2 Design Alternatives:

It is important to decide the granularity of scheduling entities. It may be scheduled at tuple levels but would be practically infeasible as the number of tuples is huge in data streams. There is always some cost incurred in switching from one entity to

another, which at tuple level is intolerable. Another interesting possibility is to schedule queries. This is acceptable but often reduces flexibility when it comes to optimization of global execution plan. At query level, optimizations are difficult since entire query overlap is difficult to achieve. Hence the best solution lies in scheduling operators, which has granularity that is in between tuples and queries. Different operators can be assigned different priorities which helps in achieving best results. Join can be provided a higher priority than select which is better as compared to assigning same priorities to both at query level. Similarly operators having greater fan out can be assigned higher priorities than operators having low fan-out. Thus considering granularity at operators is the best possible alternative.

4.5.3 Scheduling Policies:

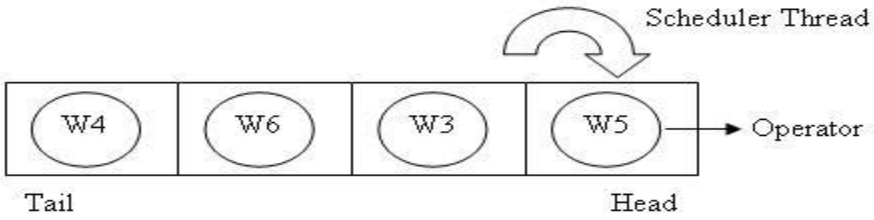

Figure 4.16 Scheduler

DSMS scheduler supports the following scheduling policies based on time quantum assigned to operator:

1. Round robin scheduling (Bottom-Up): In this scheduling scheme, equal weights are assigned to all operators. They are scheduled in the order in which they are instantiated. Figure 4.16 illustrates a ready queue in which operators are placed, as they

are instantiated. This queue is traversed sequentially to schedule operators. Every operator is scheduled for the same time quantum. This is not effective for queries having strict deadlines if system is overloaded, as it virtually does no optimization.

2. Weighted Round robin scheduling: It assigns different weights to different operators based on priorities. Higher the priority of operator, higher is the weight (time quantum) assigned to it. This is more effective than its counterpart as various parameters like tuple latency, query throughput and quality of service requirements can be controlled by assigning appropriate weights to different operators. For example a system with higher weights assigned to join than select would definitely perform better with respect to memory utilization than a system with both operators sharing equal weights under stress. Similarly assigning higher weights to leaf operators than non-leaf operators would increase the system utilization.

CHAPTER 5

IMPLEMENTATION

5.1 DSMS SCHEMA

DSMS schema stores complete information about all streams registered with the system. A new stream will be recognized only when its definition is stored in the DSMS schema. Schema information of all the streams is maintained in a persistent storage for recovery in case the system crashes. Hashtables and vectors are the data structures used to store schema information in memory. Hashtables store information as a key-value pair. The stream-name is stored as a key and a vector, containing the complete information about that stream as a value. The Hash table and vector is termed as StreamHashtable and StreamVector respectively.

The first element in a stream vector is a pointer to another hash table (termed AttributeHashtable), containing the complete description about the attributes of the corresponding stream. AttributeHashtable contains attribute-name as the key and a vector (termed attribute position vector) containing details about that attribute as the value. Attribute details include its name, data-type (which may be varchar, number or boolean) and its position in the stream. Since vectors can grow dynamically, additional details about the attributes can be added if necessary.

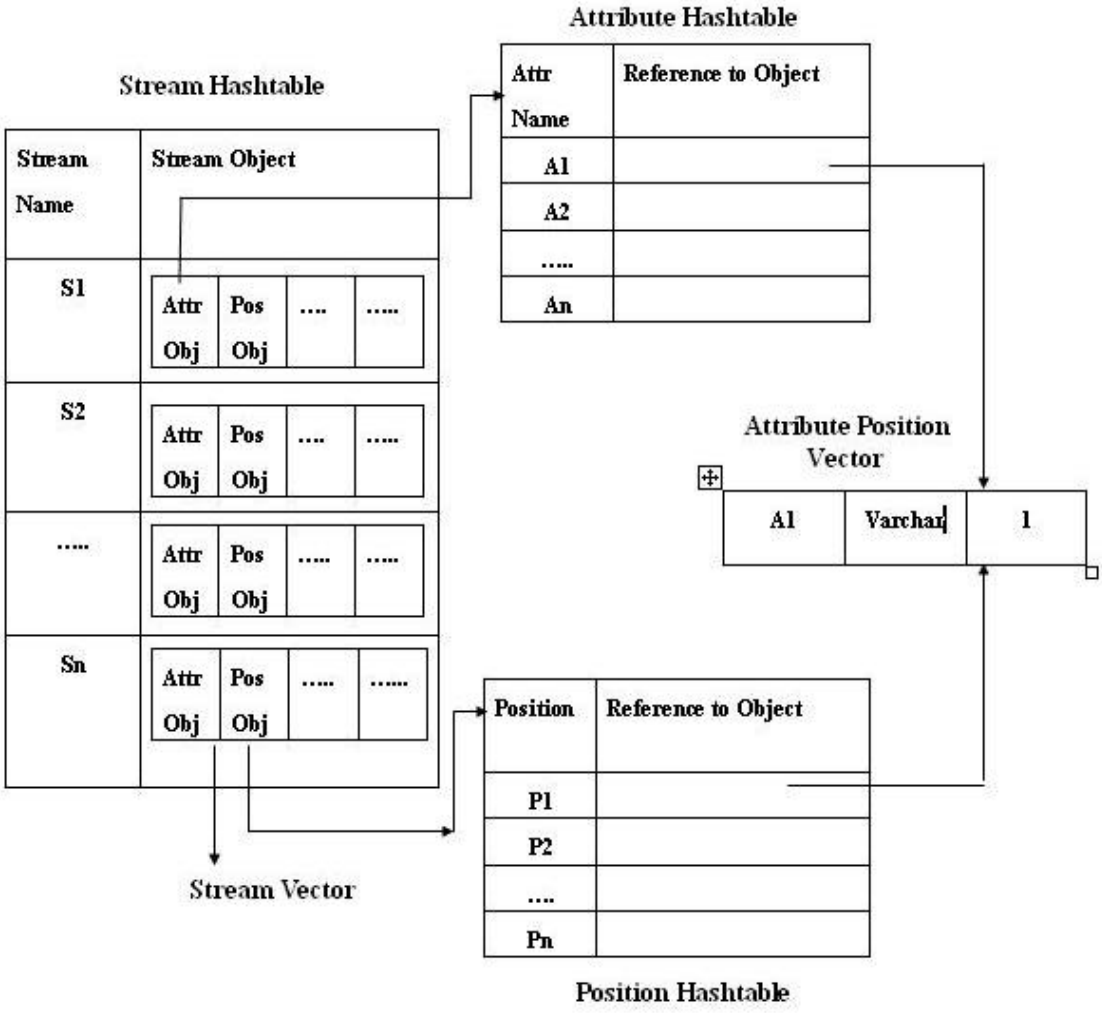

Figure 5.1 DSMS Schema Data Structure

The second element of the Stream Vector is also a pointer to a hash table (termed PositionHashtable) containing attribute description based on positions. In Position Hash table, positions of attributes are stored as a key and its value is a pointer

to the same AttributePositionVector described above. Essentially both AttributeHashtable and PositionHashtable point to the same vector (AttributePositionVector) that describes the attribute with respect to its name data-type and position in the stream. The complete setup is shown in Figure 5.1

All Streams are registered as key-value pairs in a Hash table to improve searching time. Given a StreamName, its complete information can be accessed quickly as all links are maintained through Hashtables, which needs O (1) time for searching.

5.2 Buffer

High-speed streaming data are buffered in queues, which are consumed by operators connected to it. The output of one operator is buffered in its output queue, which may be the input queue of the next operator in a query tree. Buffer decides the input –output relationship among operators. A single buffer is implemented as a queue using a vector. This data structure can grow or shrink dynamically as elements are added or consumed. Buffer may be bounded or unbounded depending upon its upper limit specification. An unbounded vector grows until the main memory is exhausted since its upper limit is unspecified.

Buffer supports two operations: enqueue and dequeue. Operator consumes data elements by calling dequeue which returns corresponding data object to operator and removes its reference from the queue. Operators attempting to read from an empty

queue are suspended to save CPU cycles. Enqueue operation adds elements in the queue and sends notify signals to resume all operators waiting on it. These two APIs are synchronized to maintain data consistency. Synchronization acquires lock on the entire object but allows non-synchronized APIs to execute simultaneously with synchronized APIs.

Minimal persistence logic is also supported to handle bursty, asynchronous and high-speed data streams. Elements are added in a file in sequential order when main memory buffer size is reached. Two files are maintained for each buffer of which only one will be used for storing elements at any point of time. When the maximum file size is reached for the currently active file, the other file is opened to continue storing elements. Dequeue removes element references from queues when consumed by operators creating main memory buffer space. File operations are not recommended for each dequeue and hence this operation starts a separate thread to read elements from corresponding files only if fifty percent or more tuples are removed from main memory buffers.

5.3 Streamed Operators

5.3.1 Split:

This operator expects a list of conditions that are evaluated against a stream of tuples. Incoming tuples are subjected to sequential condition evaluation. If the tuple satisfies the condition, it is sent to the output queue associated with that condition else

the same tuple is evaluated for the next condition in the list and this process is repeated until the list is exhausted. Condition string is typically a combination of attribute name, relational and/or logical operators and constants. For example, *"tbRoom = "b" and tbDeviceId > 5"*.

The complexity of *Split* lies in evaluating the above condition string, which involves replacing attribute names (tbRoom, tbDeviceId) with corresponding tuple field values. This modification is needed for it to be correctly interpreted by Free Ecma Script Interpreter (FESI), a tool that can evaluate any valid java expressions. It makes use of the following methods to generate the modified string.

5.3.1.1 Important APIs in Split:

findPositionofOperands: Split has multiple conditions to evaluate. Every time a condition is evaluated, a schema needs to be accessed in order to replace the attribute name mentioned in the condition string with the corresponding tuple field values from the incoming tuple. If there are 'N' conditions, schema needs to be accessed 'N' times. The other alternative is to access the schema just once and set all the attributes of the input streams with the corresponding tuple field positions. The latter is preferred because accessing schema is a time consuming operation. However sometimes it results in setting attribute names with their corresponding field positions even if it is not specified in any of the condition list.

Since the *Stream Name* is always known to operators (this is passed when operators are instantiated), this API accesses schema definition of the corresponding stream to find the position of operands. Once the positions of operands are obtained by executing the above API, these attributes are replaced dynamically by corresponding tuple field values, which results in a modified string comprising only of constants and operators. This modified string is ultimately subjected to condition evaluator (FESI), which returns true if the condition is satisfied.

Free Ecma Script Interpreter (Fesi):

FESI, used as a condition evaluator in this system is a powerful utility that evaluates any valid java expressions dynamically at run time. The setMember () method of FESI accepts two arguments. The first argument is a key and the second argument is a value. In the condition string, it replaces all occurrences of keys with their values. In this case, attribute names are provided as keys and actual field contents from tuples are provided as values. Thus it sets the attributes to their actual values from the tuples. The position to be substituted for attributes are obtained from the schema once. The value at the corresponding field from the tuple is fetched and set for the respective operand.

5.3.1.2 Split Example

The following example illustrates how a condition string is modified:

Let the condition string be "tbRoom = "b" and tbDeviceId > 5 ".

Let the schema name is tbDeviceRoom and the schema is as in Figure 5.2:

Attributes	DataType	Position
tbDeviceId	Number	1
tbDeviceDescription	Varchar	2
tbRoom	Varchar	3
tbRoomDescription	Varchar	4

Figure 5.2 tbDeviceRoom Schema

FunctionName	*Input to the Function*	*Output obtained*
findPositionofOperands	*tbDeviceId*	*1*
	tbDeviceDescription	*2*
	tbRoom	*3*
	tbRoomDescription	*4*

Figure 5.3 APIs Input and Output

The “setMember ()” method sets the operands to their actual values from the tuples. The values to be substituted for operands are obtained from their position in the schema. The value at the corresponding field from the tuple is fetched and set for the respective operand.

Let us suppose that the first tuple read from the input queue is as in Figure 5.4:

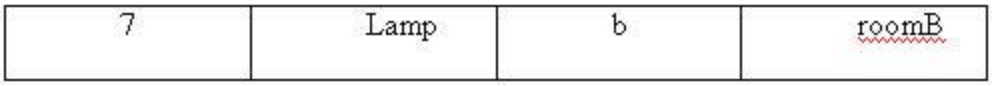

Figure 5.4 Stream Tuple

Original condition string was:

"tbRoom = "b" and tbDeviceId > 5 ".

The position of "*tbRoom*" is 3 and position of "*tbDeviceId*" is 1 in the schema.

Thus third field is fetched from the tuple for *tbRoom* that is "b" and first field is fetched from the tuple for *tbDeviceId* that is 7.

The modified condition string on setting the operands with their respective tuple values is:

"b" =="b" and 7 > 5. This string is input to the eval () method of FESI, which accepts a condition string as a parameter. It returns true as the condition is satisfied. The tuple is then sent to the output queue associated with this condition.

5.3.1.3 Design Issues:

findPositionofOperands method is called just once to access the schema information to find the position of attributes so that they can be set to their corresponding tuple field values. This avoids accessing schema for each and every tuple which otherwise would have been very expensive as tuples in stream arrive in bulk.

Once this method is called, *split* runs continuously and evaluates the condition string against data stream until the query is ended.

However calling the setMember () for every incoming tuple cannot be avoided because the string can be evaluated but not constructed dynamically. The string cannot be modified to a form that eliminates the need of setMember ().

Consider the same example string *"tbRoom = "b" and tbDeviceId > 5 "*. If this is replaced by v [3] + " = 'b' and " + v [1] + ">5", it prompts an error since the values of v [3] and v [1] are still not available as the tuples are yet to be read. If the same string is replaced as *"v [3] = "b" and v [1] > 5 "*, then it takes v [3] and v [1] as string constants. In either case, string cannot be constructed dynamically and hence setMember () needs to be called for every tuple.

5.3.2 Join operator

It is a non-blocking operator working on windows to produce results incrementally and continuously for continuous queries without waiting for the window to elapse. It is implemented as a single thread with two internal buffers, one corresponding to each of its external queues. Every new tuple from one stream is joined with all the tuples satisfying the join condition and falling in the current window from the opposite stream and then stored in the corresponding internal vector. Both join and insertion phases for one tuple must be executed prior to processing next tuple to produce correct results. If two new tuples are read, one from each of its external

queues, the one with lower timestamp is considered for join to produce output in the timestamp order.

Query Window: It is important to understand the window concept prior to implementation specific details of the join operator. It was necessary to define windows at query level, which requires every operator to register with the query window class. This approach was however not used since making a procedure call to query window and changing window bounds every time a window is altered was expensive. A better solution is to fetch the initial window specifications from query window only once, and manipulate them locally for modifying the windows rather than making procedure calls to them.

A *Query Window* class provides window specifications such as window start time, window end time, hop size and end query time. It generates APIs to set and retrieve the same. Operators can use APIs of this class to fetch initial window specifications and manipulate windows of other operators independently allowing different operators of the same query to see different windows at the same time.

5.3.2.1 Nested Join Implementation without reuse:

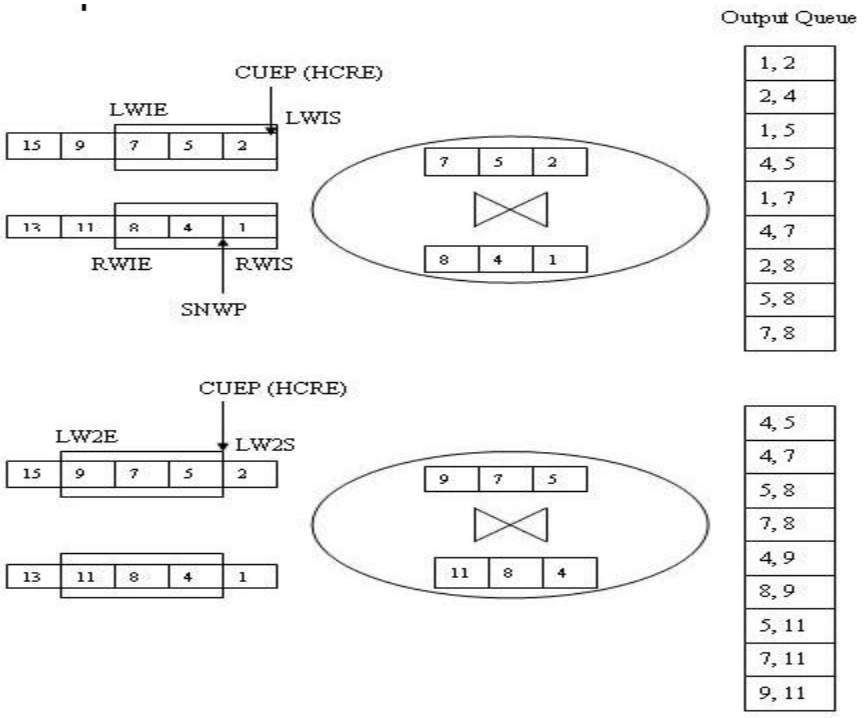

Figure 5.5 Nested Join without Reuse

Figure 5.5 indicates that join has two external buffers (left input queue and right input queue) and two internal buffers. The need for internal buffers is explained shortly. Windows are defined on these buffers with an assumption that these buffers always have the same window bound. A new window is generated only when all tuples falling in left and right window are processed completely. To start the operation, first the window bound is obtained from *Query Window* class along with other window

information. In the left buffer it is represented as LW1S and LW1E. In the right buffer window bounds are represented as RW1S and RW1E. Tuples are read from these queues and checked whether they fall in the current window. In this system, tuples are timestamp ordered. Buffer position of the first tuple that falls in the current window is marked. Hence all elements prior to this position are irrelevant in the current window. This position is called HighestCommonReadElement (HCRE). Purging logic is used on the buffer in which lowest value of HCRE among all the operators is calculated and all elements are purged up to that position. Lowest value of HCRE is considered instead of highest, as it is not correct to remove tuples, which are not yet processed by other operators sharing the buffer. Lowest value of HCRE guarantees that none of the purged elements would be needed by any operator~~s~~ sharing the buffer. Thus first window is defined for providing input bounds and purging logic is implemented to remove stale tuples before starting the actual join computation.

Every buffer has a CurrentUnreadElementPointer (CUEP) which points to the current element to be read from the buffer. Every operator sharing the buffer has its own copy of CUEP maintained by the buffer. The join sequence is:

1. If any of the buffers are empty, suspend the operator.

2. Assume that the tuple is read from left buffer first. From the figure, its timestamp is '2'. Read the corresponding tuple from right buffer with timestamp as '1'.

3. Compare the two timestamp and pick the tuple for join having smaller timestamp. In this case, it is '1'. Output produced must essentially be timestamp ordered. Since this tuple belongs to the right external buffer, it is checked against all the tuples in the left internal vector that satisfy the join predicate. If a match is found, it joins the tuple and produces the result at the output queue. The joined tuple has the joining attribute removed from the right tuple (since it is same as the joining attribute of the left tuple) and the resultant tuple has lower timestamp followed by the higher timestamp in the last two fields. It is then added in the right internal buffer (since the tuple under consideration was read from right external buffer).

4. As the tuple with timestamp '1' considered for join belongs to right external buffer, its CUEP is incremented by 1. CUEP of left buffer remains unaltered.

5. LW1E and LW2S are the two timestamps of interest. Find the first tuple whose timestamp hits LW2S and mark that position as Start Next Window Pointer (SNWP). The computation of the current window is terminated once a tuple hits the LW1E timestamp.

Repeat steps 1 to 5 until tuples are found in each buffer with their timestamp exceeding the current end window bound. The steps are common for both the shades of nested Join. Prior to beginning the computation for the next window, following steps are taken which are different in two versions.

Nested Join without Reuse: Here the computations of the current window are not used for the computation of next window. Every window is computed independently of each other. Hence it does the following:

1. CUEP is set to SNWP. This is useful in case of overlap window. For disjoint window, CUEP is same as SNWP.

2. Since next window does not make use of current window, all internal buffers are cleared.

3. Purging logic is called which sets CUEP as the HCRE.

4. Next window is generated. If the end time of next window is greater than end query, operator is stopped else next window is computed.

5.3.2.2 Nested Join with reuse:

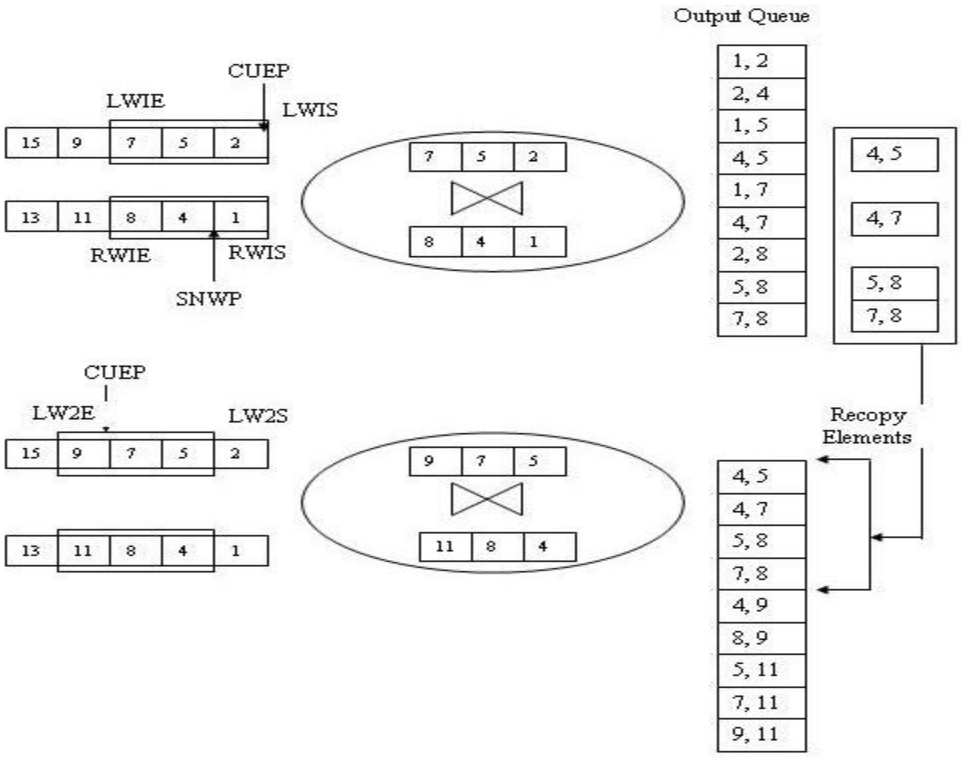

Figure 5.6 Nested Join with Reuse

As shown in above Figure 5.6, it makes use of window computation of the current window for the computation of next window. It is true only for overlap windows as disjoint windows do not have any computation to share. It does the following:

1. Computation of overlapped region is stored in a temporary data structure within the operator for the current window. As soon as the current window is

processed, all resultant tuples of overlapped area are copied as it is in the output queue for next window computation and hence the name reuse.

2. Internal buffers are not completely cleared. Only those elements are cleared whose timestamps are less than LW2S. Internal buffers cannot be completely cleared because they represent elements falling in overlapped area. These elements from internal buffers avoid reading from external buffers. They are needed so that they can be joined with new elements falling in next window.

3. CUEP is not set to point to SNWP rather it points to LW1E. Elements falling between SNWP and LW1E are present in internal buffers to be considered for join.

4. Purging logic is called which sets HCRE to LW1E.

5. Next window is generated. If the end time of next window is greater than end query, operator is stopped else next window is computed.

5.3.2.3 Important issues in Join:

Internal Buffers:

Internal buffers are used to avoid multiple scan on external buffers for every join computation and hence reduces load on them. In the absence of internal buffers, every new tuple read from one external buffer would have been scanned with all the tuples in the other external buffer falling in current window, significantly increasing the load on them. Internal buffers also increase the memory space utilization. Purging logic

would not have been able to purge tuples from external queues if internal buffers were absent, as they may be needed by the next overlap window. Streaming data, which is unpredictable and bursty, would grow the queue size to a point at which disk operations could not have been avoided. The only role of internal buffers is to buffer stream data to be read by operators. Once tuples are read into internal buffers, purging logic can safely remove elements from external queues. This would reduce disk swapping significantly thus improving the response time. This is helpful especially with *reuse* join computation.

Boundary tuples:

This is a boundary condition, which needs to be considered for ensuring correctness. In disjoint windows, for reuse and without reuse, when the first tuple is encountered, which falls beyond the current window bound, it is not considered for join for the current window. But it should be the first element to be considered for next window computation. But CUEP has already shifted by 1 and points to the tuple next to it. In order to ensure that the boundary tuple is considered for the next window, it is stored in a temporary variable. As we begin computing next window, this tuple is read from the temporary variable and computed prior to computation of other tuples.

Timestamp Ordering:

Join is a windowed operator and expects tuples to be timestamp ordered in order to determine end of current window computation. As soon as it encounters a tuple with

timestamp greater than end time of current window, it declares end of current window computation as following tuples are guaranteed to fall beyond the current window due to timestamp ordering. Hence join operator should also produce tuples which are ordered by timestamp to ensure that higher windowed operators also execute correctly.

This algorithm produces joined tuples, which are ordered by higher timestamps. A tuple is blocked at the input and is not considered for join until it finds a corresponding tuple with higher timestamp from the opposite stream. When it is joined with tuples present in internal buffers, resultant tuples are generated with lower timestamp followed by higher timestamp as the last two fields. Higher windowed operators consider last field (higher timestamp), which is guaranteed to be timestamp ordered.

Duplicate tuple avoidance:

Tuples read from left external buffer are not placed in the left internal buffer until they are joined with all tuples satisfying join predicate in the right internal buffer. This is done as an atomic action. If two tuples arrive with the same timestamp in left and right external buffers, they will be processed in sequence. Atomic action ensures that two tuples are not processed simultaneously which otherwise would have resulted in duplicates. Since tuples read from external buffers are considered one at a time for performing join with tuples in an internal buffer, duplicates are never produced.

5.3.3 Methodology for Experimental Evaluation

All the experiments were run on an unloaded machine with dual 2.4 GHz Xeon processors, 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) [20] 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. Delay between two consecutive tuples follows Poisson distribution. In order to evaluate the performance differences of two variation of "Join" explained above with respect to memory utilization, average tuple latency and query lifetime, several experiments were performed by varying the window sizes and their overlap. Prior to "Join" experiments, it is important to understand the behavior of varying data rate (Poisson distribution) on tuple latency and processing time. The following experiment is performed with a single query having a single operator (Nested Join Re-compute), to avoid any false reporting of time due to system overload by running multiple queries and operators or by any other factor. Sliding window of size 1000 tuples is chosen with the number of windows as five. The data set is uniform (all windows have same number of tuples). The data rate is varied from 5 tuples/sec to unbounded where unbounded represent no delay between two consecutive tuples or flooding the data into the system.

5.3.3.1 Analysis of Total processing time and Average tuple latency by varying data rate:

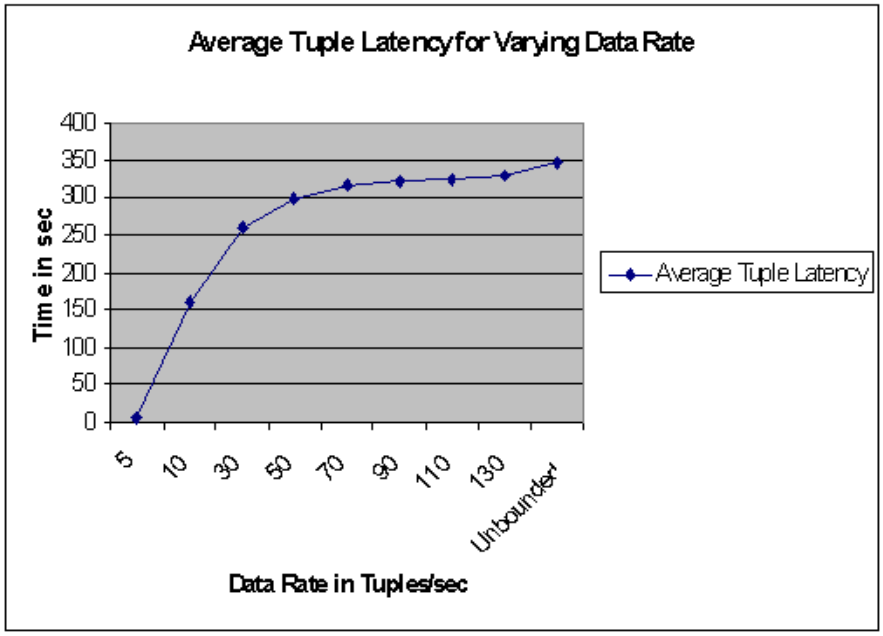

Figure 5.7 Average Tuple Latency for Varying Data Rate

Average Tuple Latency: Tuple latency is defined as the difference of the timestamp at which tuple is produced at the output and the timestamp at which it entered the system. Average tuple latency is then calculated as the average of tuple latencies of all output tuples. In Figure 5.7, it is observed that as data rate increases, average tuple latency also increases. This is because, initially when the data rate is low, operator processes tuples immediately and there are virtually no tuples waiting in the

buffer. Tuples are processed as soon as they enter the system. But as the arrival rate increases, more tuples are produced within the same interval. If it exceeds the processing time, tuples are buffered in the queue increasing the waiting time. Hence we can say that average tuple latency is proportional to data rate (arrival rate) of streams.

Total Processing Time:

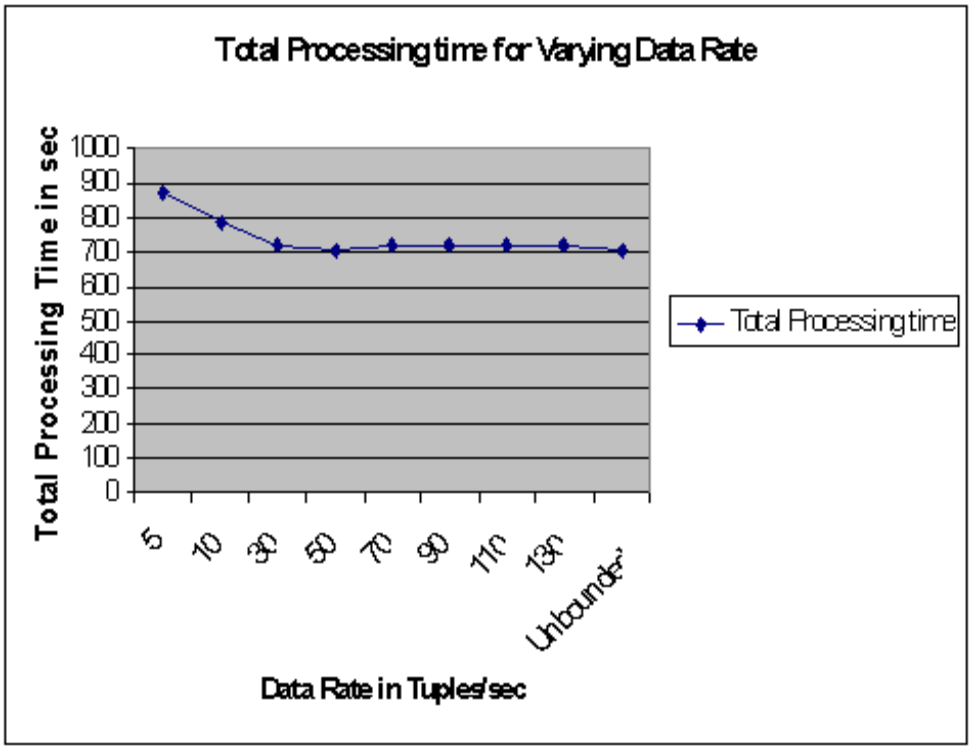

Figure 5.8 Total Processing Time for Varying Data Rate

The total processing time is defined as the difference of time at which the query is terminated and the time at which it is started. From the Figure 5.8, it can be observed that the behavior is totally different from average tuple latency. As the data rate

increases, total processing time decreases and eventually it becomes constant regardless of the data rate. Initially when data rate is low, operator utilization is less as quite often operator is suspended due to unavailability of tuples because of low data rate. As data rate increases, operator utilization increases and hence the throughput and total processing time decrease. At one point, the arrival rate becomes equal to the processing rate upon which the total tuple processing time becomes constant because operator utilization cannot be increased beyond its maximum processing capability and the total number of input tuples for processing is also fixed.

Maximum input buffer count reached: Join operator has two input buffers that collect stream data either from base streams or from its child operators. Since "join" is the only operator present, this parameter indicates the maximum number of elements present in the buffer at any point in time during the entire query processing. This parameter is again dependent on data rate. Initially when the data rate is low, processing rate is higher than the arrival rate and hence there was no or less accumulation of tuples in the buffer. As data rate increases, more tuples accumulate in the buffer and the maximum count of input buffer increases proportionally.

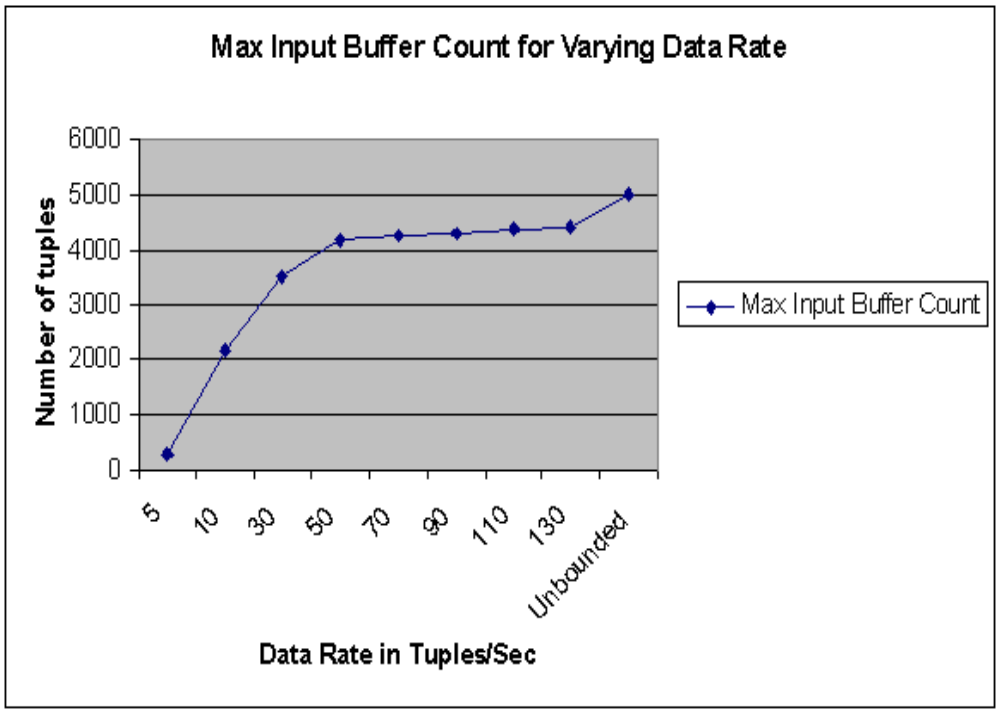

Figure 5.9 Max Input Buffer Reached for Varying Data Rate

5.3.3.2 Analysis of Average Tuple Latency, Total processing Time and Internal memory used for Nested Join Re-compute and Reuse

To compare the performance of Nested Loop Re-compute Vs Reuse, the data rate is 70 tuples/sec and the percentage overlap is increased from 10% to 75%. The percentage overlap is the most critical factor in performance evaluation. It is expected that as the percentage overlap increases, the performance of Reuse over Re-compute

also increases provided the data set is uniform. This is because the higher the window size and higher the percentage overlap, more is the common computation exploitation.

Effect on Total processing Time by varying percentage overlap:

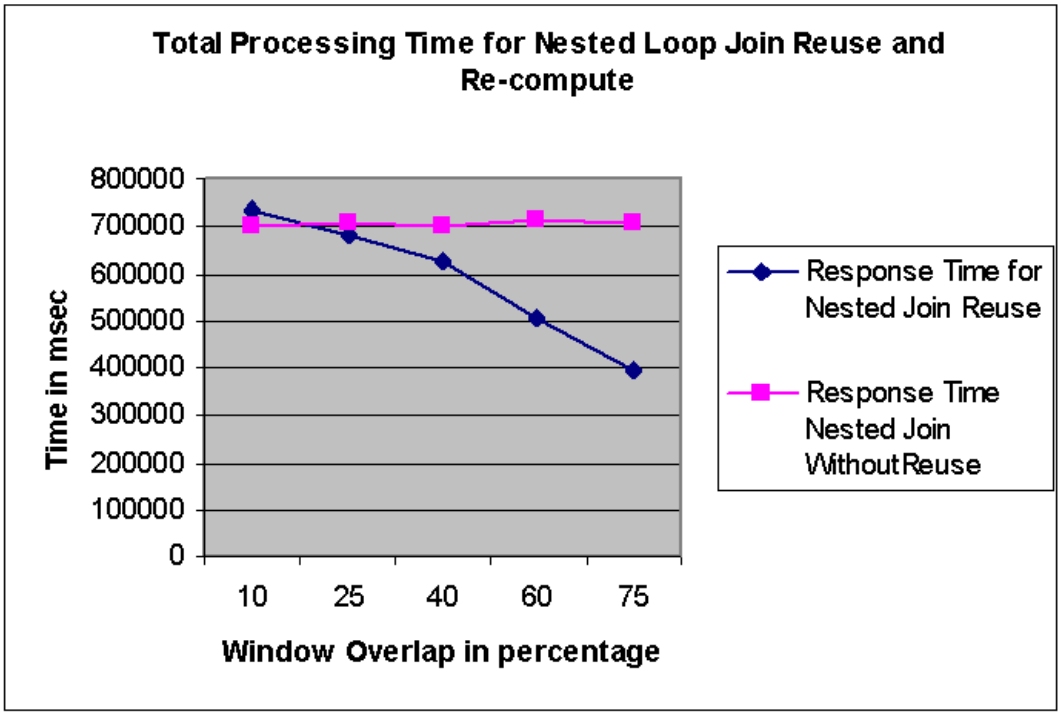

Figure 5.10 Total Processing Time by Varying Percentage Overlap

This parameter is calculated as the difference of timestamp of first operator instantiation from the timestamp of last operator instantiation for the respective query. Since Nested Join is the only operator running it is the difference of the time it is terminated and the time at which it is started. Since “Reuse” avoids re-computation of

the overlap region, it is expected that higher the window size and window overlap, the higher is the saving on processing provided the number of tuples falling in the window are proportional to window size. The same effect is observed in Figure 5.10. Initially when the window overlap is small, the difference in the processing cost of both the shades is negligible, as the common computation is not exploited significantly. As the overlap increases, "Reuse" outperforms "Without Reuse" with a significant margin since the saving on common computation is considerable. Since the data set is uniform, it is possible to compare the two shades across the percentage overlap. The performance of Re-compute is almost constant as it processes the same number of tuples in every window each time since it does not exploit common computation.

Average tuple latency: This parameter is computed by averaging the tuple lifetime (difference of time at which a tuple entered the system from the time at which it exited the system) of all the tuples seen by the root operator. In Reuse, the result of common computation is immediately placed at the output queue of the next window by the current window and hence the tuple of common computation virtually has no latency added in the next window. It preserves the latency of the previous window and this saving is accumulated at each window computation that promotes reduction in the overall tuple latency in Reuse as shown in Figure 5.11. The latency of Re-compute is almost constant as it produces each window independently at the same rate. Data rate is

75 tuples/sec and data set is 1000 tuples per window that gives the same performance to Re-compute variation.

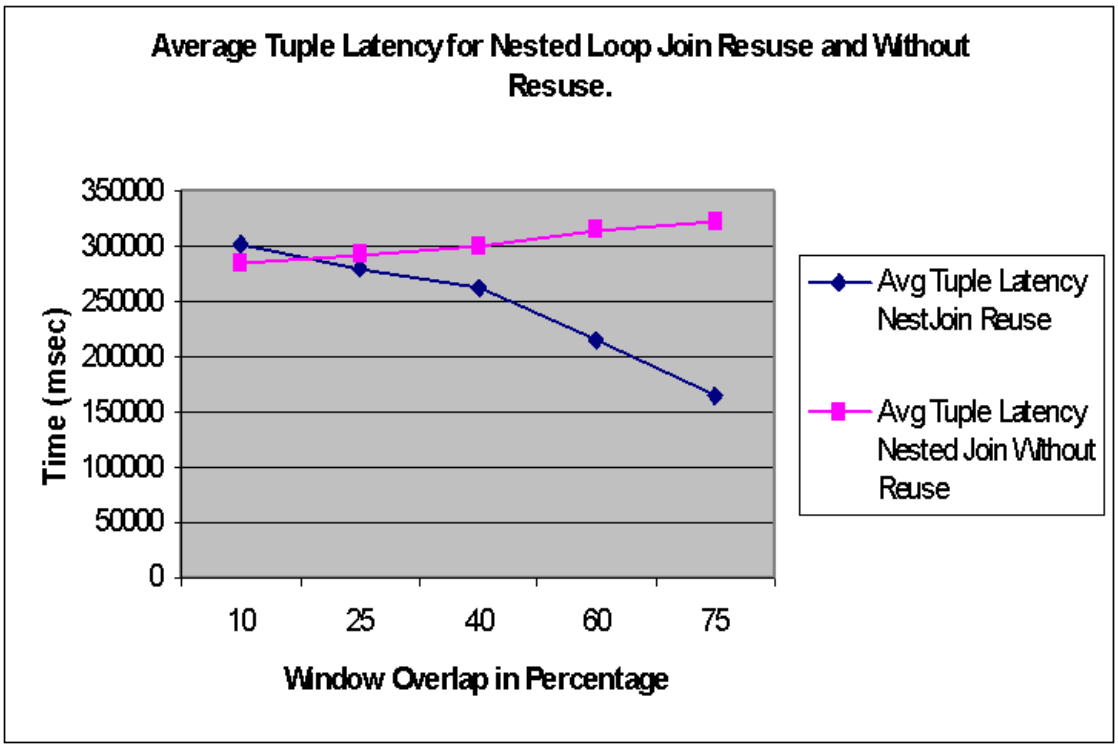

Figure 5.11 Average Tuple Latency by varying percentage overlap

Analysis of Memory Usage:

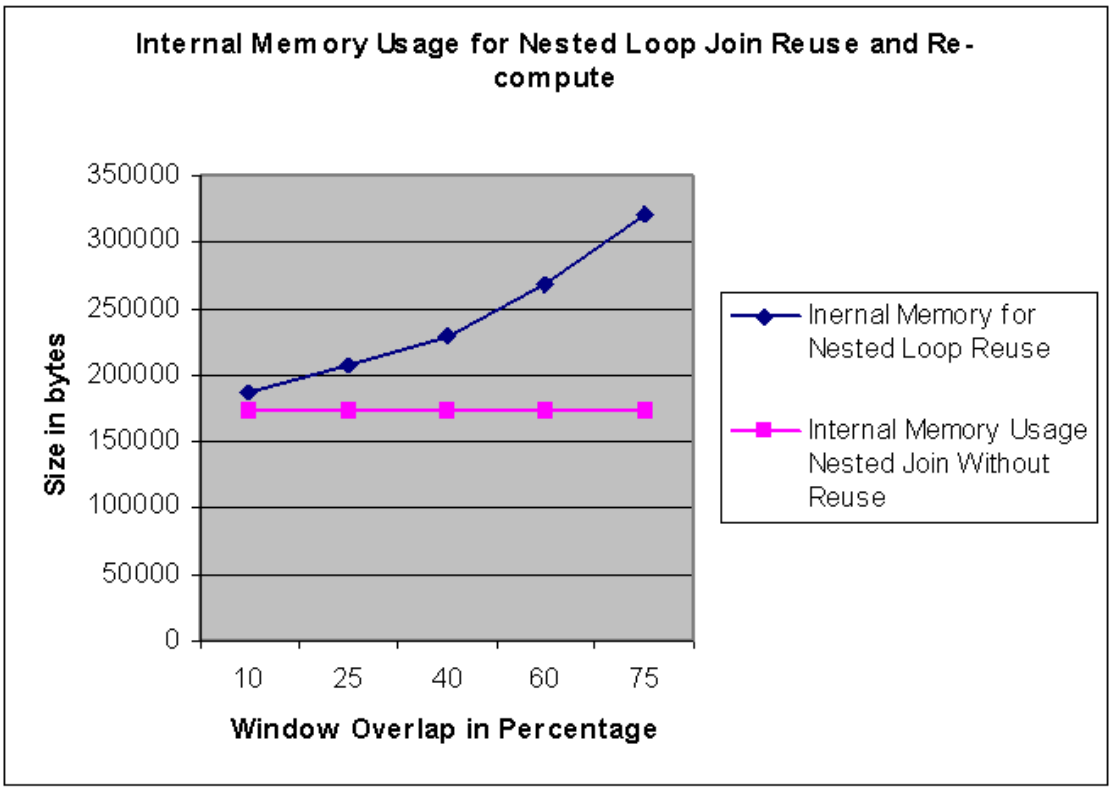

Figure 5.12 Internal Memory used by varying percentage overlap

The improvement on average tuple latency and query lifetime in “Reuse” comes at the cost of memory. The current window identifies the common computation that is copied in the temporary storage to be used for the next window computation. This temporary storage overhead is avoided in “Re-compute” as every window is computed independently. Moreover the internal memory in “Re-compute” is cleared for each window computation while in “Reuse” overlapping elements are preserved in an internal memory. The total memory cost in “Reuse” involves the cost of the internal

memory and temporary storage that are avoided in “Re-compute”. So internal memory usage in “Reuse” will always be greater than or equal to the usage of “Re-compute” as observed in Figure 5.12. The internal memory used by Re-compute is again constant as the amount of memory used depends on the number of tuples being processed per window which is constant since the data set is uniform and the window size is fixed.

5.4 Client-Server Model

This client server model is based on request-response paradigm in which clients submit requests, which are processed at the server, and the results of execution are sent back to client as response. As mentioned in the design chapter, DSMS client is designed as a web--enabled client that uses web server to provide useful functionalities that includes generation of new schema, processing user query input, submitting requests to server and dispatching results to users. DSMS Server is a program dedicated for stream processing which is continuously listening at a specific port to accept any number of client connections. This socket-based connection allows client and server to exchange request-response objects based on a pre-defined communication protocol. Once the client is connected, it sends a command object which indicates the type of service requested over the socket wrapped with object input and output streams. There is a unique command defined for each service. Once the command is received, server expects the corresponding request object. For each service, the protocol clearly defines how the request object is processed and how the response is sent to the client. Server,

upon accepting the request object, starts a new thread for processing the request and goes into listening mode to accept new client connections. Once the request is processed, response is dispatched to the client over the same socket connection.

5.4.1 Client Implementation:

The core functionality of DSMS client lies in constructing a data flow operator-buffer graph (plan object) from user query. Client must also preserve operator instantiation order to respect query semantics. Thus a data structure is needed which not only contains complete information about all operators but also maintains their instantiation order for the query to be meaningful. The solution to this problem is a query tree, which is dynamically constructed by adding operator nodes to existing tree as specified in a query. Thus a query tree is a sequence of operator nodes where every node completely describes an operator. These nodes are linked to describe parent-child relationship. If an operator node 'A 'at level n-1 is linked to an operator node 'B' at level n, 'A' is said to be a child of 'B'. A link is created from operator 'A' to operator 'B' when output queue of 'A' becomes the input queue of 'B'. This tree is constructed bottom up as and when the client adds the operators.

5.4.1.1 Data Flow Operator Buffer Query Tree

As evident from Figure 5.13, an operator node is a data structure consisting of following members:

1. OperatorData

2. Reference to left child

3. Reference to right child

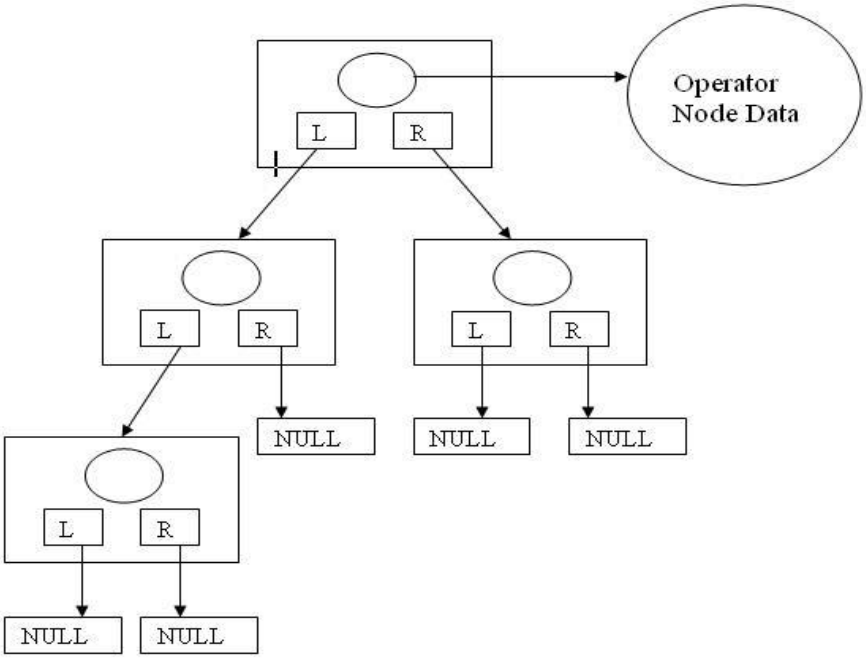

Figure 5.13 Plan Object (Query Tree)

OperatorData in turn is a data structure which completely describes an operator with respect to its operator type, input parameters (filtering condition for select operator, fields to be projected for project, etc;), input and output queues associated with an operator, input streams to be operated on, and query window specifications. References to left and right child are self-explanatory. For leaf nodes they are null. Any operator, which does not have its left or right child defined, has its corresponding reference set to null. These references are used to define links, which in turn constructs

the entire query tree. Client needs to generate following additional information prior to sending plan object to server for processing:

1. Client has complete knowledge about the operators and their instantiation order in a query. When they encounter project or join which changes base schema, they create new schema definitions to support new streams generated as the output of these operators. These schema definitions are registered with server prior to query instantiation so that they are available to next higher operators.

2. While generating new schema definitions, it resolves name conflicts, which may occur due to same attribute names of two different input streams by generating unique attribute names in the resulting stream.

5.4.2 Server Implementation

DSMS server implementation is explained as follows:

In order to facilitate socket based TCP communication, DSMS Server is extended from an abstract class called *TCPServer*, which implements generic functionality for client server communication. It allows server to mount on a specific port so that applications can connect to it. Once the client is connected, a *NetStream* object is created over the client socket to enable object-based communication over the network. This is accomplished by defining a class called *NetStream*, which extends serializable interface without which persistent object based communication is not possible. It wraps character input and output streams of the socket defined for client

server communication by object input and object output streams to write or read java objects to and from a byte stream. The key feature of serialization is that if an object refers to another object, referenced object is also serialized. This process is recursive and helps in serializing the entire query tree by just passing the root reference. It is important to understand client server communication protocol, which can be clearly explained by taking a simple example of specific service being requested from a client. Assume that client wants to register a new schema definition with the server. It goes through the following steps:

1. Since it is a command driven protocol, every request has a unique command, which is sent to the server prior to actual request.

2. Server on accepting the command is prepared to receive corresponding request object. In this case it a list consisting of two elements: stream name and its schema definition.

3. Server accepts the list and registers this stream by populating its stream data structure.

The protocol is concerned only about one to one correspondence between the services being requested and the command defined for that. It does not care whether command is a *string* specifying a service being requested or an *integer* value.

5.4.2.1 Instantiator:

Server has another important module called *Instantiator*, which deals with initialization and instantiation of operators, buffers and scheduler. Its main responsibility is to extract information from operator data node contained in plan object, initialize operators with obtained information, associate buffers among operators defined by data flow links, place operators in the ready queue of the associated scheduler and eventually schedule them for execution. Some of the initializations done by the Instantiator prior to operator instantiation are:

1. Operators at intermediate level operate on streams provided as output by their children. However it is not true for base operators, as they need to know on which stream to work on from the available streams. Instantiator has the responsibility of clearly defining the association between streams and input buffers for base operators.

2. All operators are designed as independent entities, which expect input in specific form. Thus input specifications submitted by a user needs to be modified to an acceptable form. 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

Consider a join condition set in OperatorData structure for input parameter by the client as:

tbDevice.Id > tbdeviceDescription.Id

Server already has the schema definition stored prior to operator instantiation, which is provided by the client. Instantiator accesses the schema in order to find the attribute positions of tbDevice.Id and tbdeviceDescription.Id. Once the attribute positions are found, join is instantiated with the desired parameters.

3. Operators like Project and Join generate new streams. Project shrinks and join expands their input streams. Hence prior to higher operator instantiation, their input buffers must be associated with new streams generated by preceding operators.

Server reads configuration file for initializing buffer, scheduler and operators. It defines initial main memory buffer size, initial secondary memory buffer size, and time slice to be assigned for scheduling and source and system timestamp fields of various operators. These initializations are followed by Instantiator initializations as explained above to start the complete execution process.

5.5 Scheduler

Operators are scheduled based on their state and priority. An operator is scheduled for execution only if it is in ready state. Scheduler is the highest priority thread, which picks operator reference from the head of the ready queue, starts the operator thread and schedules it for assigned time quantum. When the time quantum is

elapsed, control is returned to scheduler upon which next operator is scheduled. Operator execution may also be interrupted by scheduler prior to assigned time quantum because of unavailability of resources. Since every operator is implemented as a thread, cost incurred in context switching by scheduling algorithm may be high. This cost could have been avoided by making the entire query as a single thread. But it would have been difficult to achieve global optimization with respect to query throughput, optimization of global query plans, tuple response time, memory utilization and quality of service requirements as they can be best achieved by keeping the granularity at operator level which provides more control and flexibility for the system to adjust and adapt.

5.5.1 Implementation of scheduling policies:

Two policies have been implemented for scheduling which are as follows:

Plan object is traversed in post order to ensure that child operators are instantiated prior to parent operators respecting query semantics; they are simultaneously placed in a ready queue of the scheduler to maintain FIFO ordering. Scheduling algorithms is explained below.

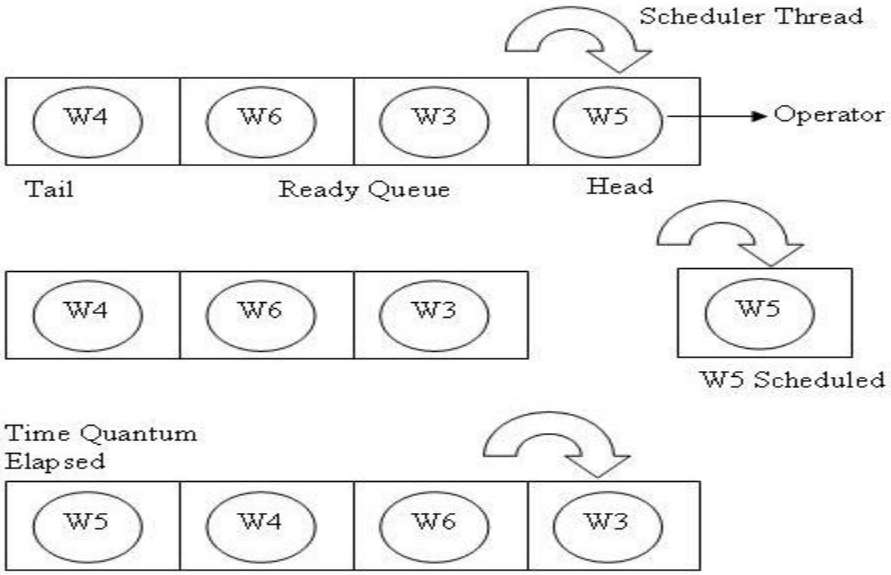

Figure 5.14 Weighted Round Robin Scheduling

5.5.1.1 Round robin scheduling:

It picks the first operator from the ready queue. If the operator is not alive (if it is not scheduled earlier), start the operator thread and execute it for assigned time quantum. If the time quantum is elapsed and the operator has not finished its operation completely, place the operator reference at the end of the ready queue. Thus all operators are guaranteed to be scheduled avoiding starvation. If the operator state indicates that it is alive (if it was scheduled earlier and is currently at ready state), resume the operator thread. Operator transitions from one state to another during the course of execution, which is explained as follows:

- If the operator is currently being scheduled and finds some of the resources unavailable, for example its input buffer is empty; it transitions from run state to suspended state.
- If all resources become available for the suspended operator, it transitions from suspended state to ready state and is placed at the end of scheduler's ready queue.
- When an operator is picked for execution by the scheduler, it transitions from ready state to run state.
- An operator is scheduled for the specified time quantum. If an operator finishes its operation and its reference is completely removed from the system.

5.5.1.2 Weighted Round Robin scheduling:

This is analogous to round robin but assigns different weights to different operators based on priority. Higher priority operators are assigned higher weights and hence scheduled for longer time. Starvation refers to a situation in which some operators or operator path are never served since there are always higher priority operators ahead of them. Starvation is avoided in this scheme as once the operator is scheduled and its time quantum has elapsed; it is always placed at the end of the ready queue, regardless of the priority of the operator. Thus every operator is guaranteed to be scheduled in time 't', where 't' is the sum of time quanta of all operators ahead of it in the scheduler's ready queue. Figure 5.14 shows the visualization of Weighted Round Robin scheduling. It provides more flexibility to the system to adjust and adapt to

satisfy quality of service requirements and attempts to provide optimal response to different optimization goals by assigning priorities accordingly. For example leaf operators can be assigned higher priority to improve system utilization, operators with higher fan-out and join operators can enjoy higher priority to avoid memory bottlenecks.

5.5.2 Implementation alternatives:

Another scheduling algorithm called priority based scheduling scheme was proposed according to which operators were scheduled strictly based on priorities. This was a pre-emptive scheduling in which a newly entered higher priority operator interrupts a lower priority operator. Since java threads has 10 priorities varying from 1-10 with 1 being the lowest and 10 being the highest, a list of ten entries is maintained with one entry per priority. Every entry in turn is a list containing operator references whose priority is same as the priority corresponding to the entry. This list was traversed in the highest priority order. It is common to have more than one operator~~s~~ with the same priority during which the operators are traversed and scheduled sequentially within the inner operator list. The obvious problem with this scheme is starvation. If there are long running queries with long running operators, scheduler will schedule operators in FIFO only at the highest priority level and lower priority operators may never get a chance to execute. This is not appropriate for data streams where real time

response and strict deadlines are absolutely essential for the response to be meaningful. Hence priority based scheduling scheme was ruled out.

5.5.3 Scheduling Experiments:

All the experiments were run on unloaded machine with 2 Xeon processor, 2.4GHz, 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) 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. Delay between two consecutive tuples follows Poisson distribution. In these experiments, the effect of varying data set on average tuple latency and total processing time is observed in various scheduling schemes (simple round robin, weighted and data flow). It is run using a single query with four operators in the system. The buffer assigned to each operator can contain at the most 1000 tuples. The data rate is fixed, 70 tuples/sec. The data set is varied from 500 tuples/window to 1500 tuples/window.

Effect on varying data set on Average Tuple Latency and Total processing time in various scheduling schemes:

It is observed from the Figure 5.15 that as the size of the data set increases the "Average Tuple Latency" and the "Total Processing Time" increases. Higher the number of tuples in a window, the more is the buffer utilization. This increases waiting

time in the buffer that is proportional to the number of tuples residing in the window. Also the query lifetime depends on the number of tuples to be processed.

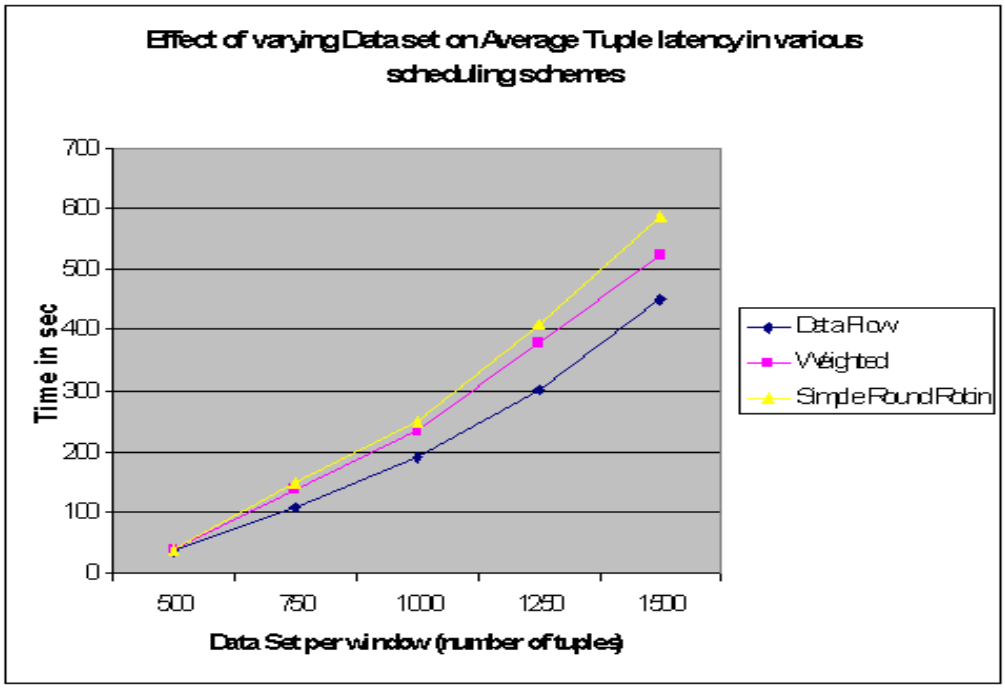

Figure 5.15 Effect on Average Tuple Latency by varying dataset

As data set increases, the number of tuples for processing increases which in turn increases the total processing time. It is important to understand the effect of various scheduling schemes. The performance of Data flow scheduling is better than the other scheduling schemes. It is a greedy approach in which operator is scheduled as soon as it has data to process else it is suspended. It makes use of operator system's scheduler. Weighted round robin is superior to simple round robin as the weights are assigned meaningfully to operators in the system. The "Select" operators that directly

consume data from leaves are given higher priority than “Project”. “Join” which is more complex and time consuming than “Select” is given still higher priority. In “Simple round robin” all operators have the same priority that affects the overall performance.

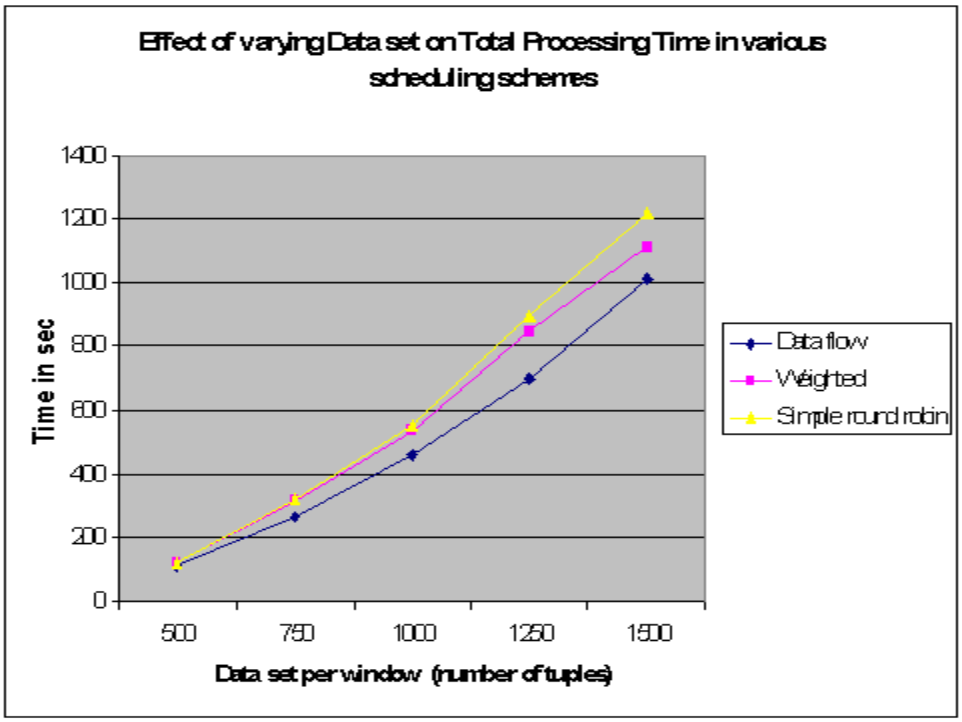

Figure 5.16 Effect Total Processing Time by varying dataset

Effect of Buffer Size on Average Tuple latency and Total Processing Time in various scheduling schemes:

The data set is fixed (1000 tuples/window). The data rate is fixed to 70 tuples/sec. This experiment involves single query with four operators. The main memory assigned to operators is increased from 500 tuples per buffer to infinite buffer. In each experiment their effect on Average Tuple latency and Total Processing Time is observed. Different scheduling algorithms (round robin and weighted round robin) are run to understand the behavior of "Average Tuple Latency" and "Total Processing Time" with respect to availability of main memory.

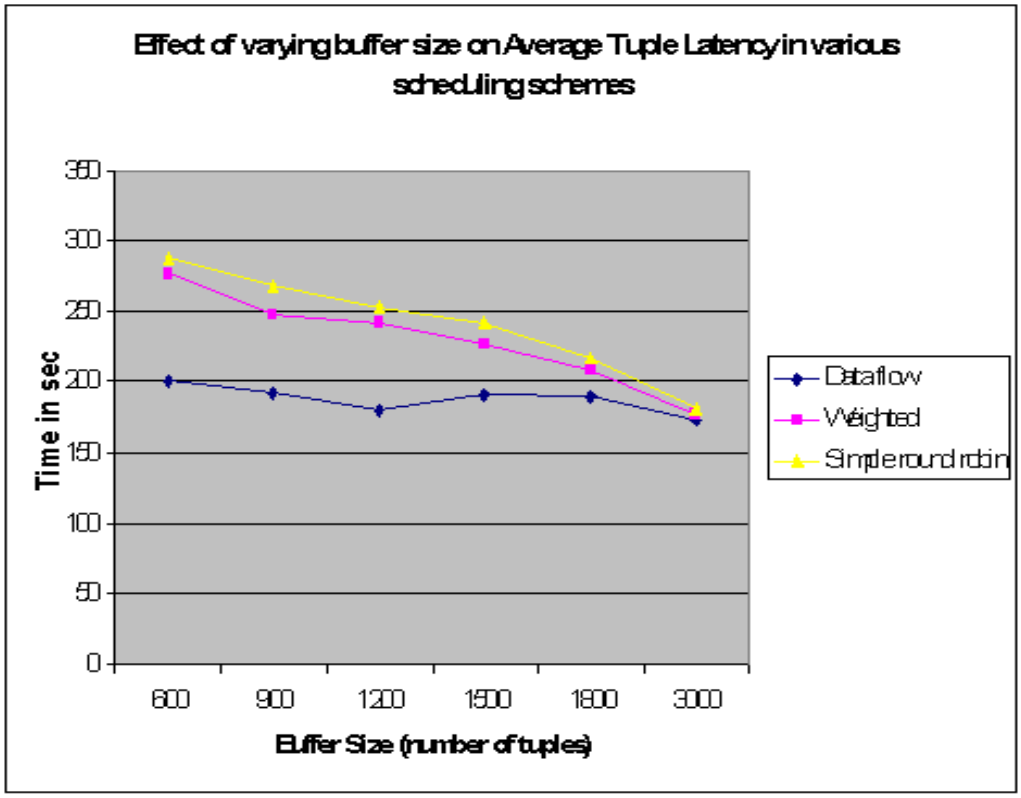

Figure 5.17 Effect on Average Tuple Latency by varying buffer size

As expected the average tuple latency and total processing time is inversely proportional to memory. Higher the memory available to operators, the lower is the average tuple latency and total processing time since no/few disk operations are involved. As the buffer sizes associated with operators are reduced, tuples that cannot be accommodated in main memory buffer are persisted on disk thus increasing the average tuple latency and total processing time, the trend observed in Figure 5.18.

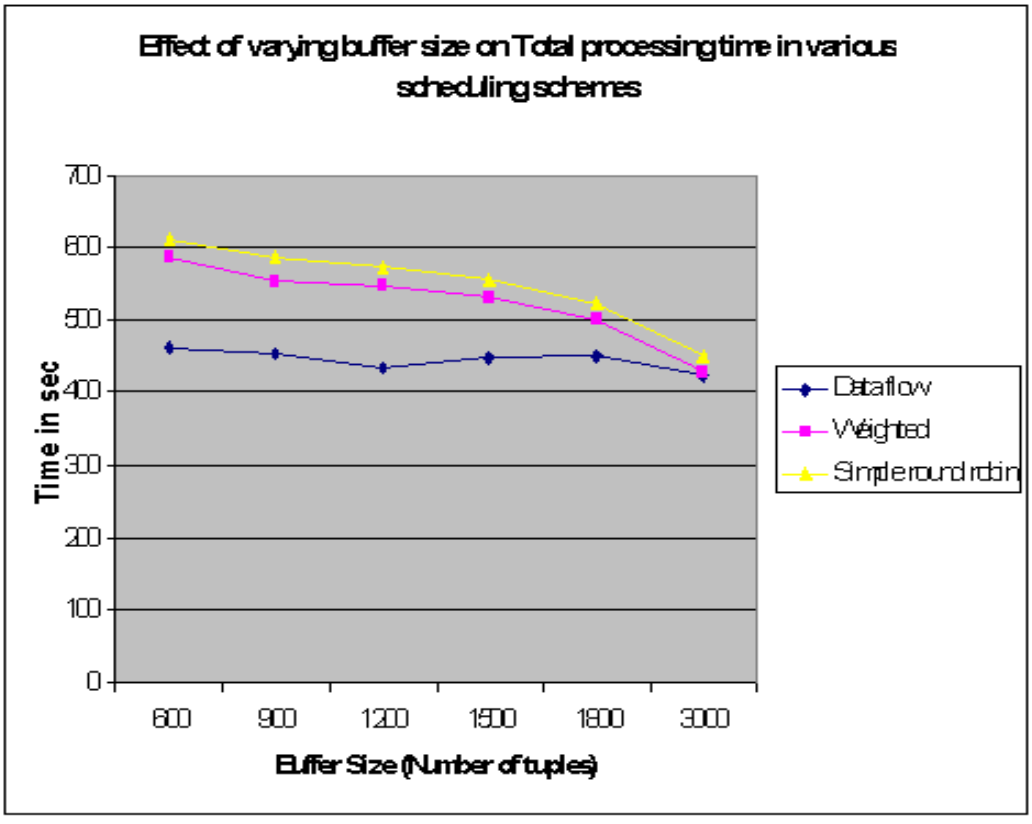

Figure 5.18 Effect on Total processing Time by varying dataset

This effect was observed by varying the scheduling schemes. It is observed again that data flow greedy approach outperforms the other two scheduling schemes. Also weighted scheduling outperforms simple scheduling in all the cases. “Join” which is more complex and time consuming than “Select” is assigned a higher priority. Since “Select” operators are closer to data source in query tree, they are assigned higher priority than “Project” as they need to cope up with high-speed streams. This meaningful distribution of priorities to operators generates better result as observed in the Figure 5.18. Simple scheduling scheme assign fixed priority to operators, hence cannot be used effectively to satisfy QoS requirements.

5.5.4 Interesting issues in Scheduling:

Scheduler is a thread, which removes operator reference from the ready queue and schedules operator for execution for assigned time quantum. Once it starts the operator thread, it calls wait () method and goes to sleep. Consider round robin scheduling in which every operator is assigned the same time quantum say 10. If an operator finishes its operation on consuming 5 time quanta, it would be more appropriate if scheduler thread wakes up immediately rather than sleeping for another 5 time quanta (completing its full waiting time). If sleep () method had been used, scheduler thread would have woken up after 10 time quanta while wait () method allows it to wake up as soon as operator finishes its operation thus saving time and improving efficiency.

Operators must register themselves with a scheduler, which facilitates communication between two entities. Operator has a setScheduler () method in which scheduler instance is passed which binds operator to the specific scheduler. Similarly operator instances are passed in the ready queue of the scheduler, which allows complete access to operator including its state and priority.

Round robin scheduler and priority scheduler are extended from *abstract* class called *Scheduler*, which contains the following generic functionality:

1. addReadyQueue (Operator optReference)
2. removeReadyQueue (Operator optReference)
3. run ()

All three methods are abstract methods, which are implemented in specialized schedulers. Their implementation differs from one scheduler to another based on scheduling policies. Even the time that could be read as a configuration parameter either from a file or from a command line is assigned in the parent class.

These scheduling schemes are static and cannot be adapted to changing optimization goals under changing system states dynamically. Scheduling operators and changing their priorities adaptively at run time to satisfy quality of service requirements and making the best possible use of run time resources is the ultimate goal.

CHAPTER 6

CONCLUSION AND FUTURE WORK

This work includes the design and implementation of query processing architecture for processing continuous streams to provide real time response to streamed queries. This architecture is push based in which tuples are processed as and when they arrive unlike traditional DBMS that pulls data from the disk. Adding a window clause to standard SQL is one of the proposed extensions to the query model of traditional DBMS. New sets of specialized non-blocking operators have been designed to operate on streams that produce results incrementally and continuously. "Split" operator is designed to partition an incoming stream into multiple outgoing streams based on some application logic. One of the fundamental issues in data streams is timestamp ordering. All windowed operators such as "Join" and "Aggregate" not only consume tuples in timestamp order but they produce tuples also in timestamp order for higher windowed operators.

Scheduler is designed with three scheduling schemes to schedule streamed operators to satisfy QoS requirements. Flow based scheduling start operators as soon as they are instantiated relying on operating system's scheduling. Simple round robin assign fixed weight to all operators of all queries and schedules them in a round robin

manner thereby avoiding starvation. Weighted round robin scheme is more realistic in satisfying QoS as different operators of the same query can have different priorities.

Another important aspect is the interface provided by this system. Query is represented by a data flow graph consisting of operators connected with queues. Instantiator traverses this plan object in post order and instantiates operators respecting the query definition. DSMS server provides a set of services such as addition and deletion of schema, and instantiation and termination of queries and operators.

As far as future work is concerned, there is much to be done. Alternate plan generator needs to be developed to produce alternate equivalent 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. Run time optimizer is needed to monitor the output for QoS requirements. It can tune all the components of the system to satisfy desired QoS requirements. The current scheduling schemes are static and needs to be modified to support adaptive scheduling by dynamically assigning priorities to operators depending on the system load. The ultimate goal is to make the scheduling algorithm to be an optimal one, which can change its optimization goal under different system states, and take the QoS requirements into consideration.

REFERENCES

1. Motwani R., Widom 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**.** 26, No1.

2. Babcock B., Motwani 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), June 2002.

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

4. Centintemel U., Rasin 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.

5. Carney D, Cetintemel 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, August 2002.

6. The Stream Group, *The Stanford Stream Data Manager*. IEEE Data Engineering Bulletin, 2003.

7. Franklin M., Chandrasekaran S., *Streaming Queries over Streaming Data.* In 28th VLDB Conference, The International Journal on VLDB, Hong Kong, China, August 2002: p. 140-156.

8. Madden S., Shah M., Hellerstein M. & Raman V, *Continuously adaptive continuous queries.* In Proc of SIGMOD Conference, Wisconsin, Madison, June 2002.

9. Chandrasekaran S., Cooper O., Deshpande A, Franklin M., Hong W., Krishnamurthy S., Madden M., Raman V., Reiss F., Shah M., *TelegraphCQ: Continuous Data Flow Processing for an Uncertain World.* 1st CIDR Conf., Asilomer, CA, January 2003.

10. Widom J, Babu S., *Continuous queries over data streams.* SIGMOD Record, 2001: p. 109-120.

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

12. Madden S., Franklin J. *Fjording the stream: An architecture for queries over streaming data.* In Proc. Int. Conf. on Data Engineering, 2002: p. 555-566.

13. Bonnet P., Gehrke J., Seshadri P., *Towards sensor database systems.* In 2nd International Conference on Mobile Data Management, Hong Kong, China, January 2001.

14. David J., Chen J., *Dynamic Regrouping of continuous queries, Wisconsin, Madison.*

15. Chen J., David J., Tian F. & Wang Y., *NiagaraCQ: A scalable continuous query system for internet databases.* In proc of ACM SIGMOD Conf. on Management of Data, 2000.

16. SreekantT, *Recoverable global event detector for distributed active applications.* 2002.

17. Krishnamurthy S., Chandrashekaran S., Hellerstein M., Deshpande A., Franklin J. & Mehul Shah., *Windows Explained , Windows Expressed.* Submitted for publication, 2003.

18. Jiang Q., Chakravarthy S. *Scheduling strategies for Processing Continuous Queries over Streams.* Technical Report, 2003.

19. Dustin J., Chakravarthy S. *Queuing analysis of SPJ queries over continuous data streams.* Technical Report , Arlington, TX, 2003.

20. Group. M. *http://mavhome.uta.edu*

BIOGRAPHICAL INFORMATION

Satyajeet Sonune was born Septemeber, 1978 in Mumbai, India. He received his Bachelor of Engineering degree in Computer Science and Engineering from Mumbai University, Maharashtra, India in September 2000. In the Fall of 2001, 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 2003. His research interests include stream processing.