DESIGN AND IMPLEMENTATION OF STREAM OPERATORS, QUERY

INSTANTIATOR AND STREAM BUFFER MANAGER

The members of the Committee approve the master's
thesis of Altaf Gilani

Sharma Chakravarthy
Supervising Professor _____

Alp Aslandogan _____

David Kung _____

DESIGN AND IMPLEMENTATION OF STREAM OPERATORS, QUERY INSTANTIATOR AND STREAM BUFFER MANAGER

by

ALTAF GILANI

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE AND ENGINEERING

THE UNIVERSITY OF TEXAS AT ARLINGTON

December 2003

To My Parents, Grand Parents, Brothers and Friends

ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. Sharma Chakravarthy, for his great guidance and support, and for giving me an opportunity to work on this thesis. I am also thankful to Dr. Alp Aslandogan and Dr. David Kung for serving on my committee.

I would like to thank Satyajeet Sonune for his constant support and encouragement throughout this thesis. I would like to thank Dustin for his valuable tips during the implementation of this project.

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

December 03, 2003

ABSTRACT

DESIGN AND IMPLEMENTATION OF STREAM OPERATORS, QUERY INSTANTIATOR AND STREAM BUFFER MANAGER

Publication No. _____

Altaf Gilani, M. S.

The University of Texas at Arlington, 2003

Supervising Professor: Dr. Sharma Chakravarthy

Data intensive applications like Network monitoring, financial applications, sensor-based applications etc are emerging. They have a continuous, unpredictable and unbounded flow of data, referred as streams.

This thesis describes the query processing architecture for stream-based applications. Generic representation of window parameters in a query is explored. Design and implementation issues of SELECT, PROJECT and JOIN are also covered.

An effective buffer management scheme is provided such that more than one operator can share a buffer. Buffer provides means for stream database to work in low main memory environment by making use of primary as well as secondary storage memory. Design and implementation of Buffer management is covered in this thesis.

Server provides a platform for integration of various components of its architecture. Instantiation of query plan is one of the responsibilities of the server. This thesis covers the design and implementation of Server.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

Figure Page

Chapter 1

INTRODUCTION

Applications have started demanding responses with a pre-defined quality of service requirements. Traditional DBMS's do not deal with QoS requirements. Pervasive computing has resulted in many sources of sensor data (e.g., RF tags) from which data arrives in the form of streams. Streaming can arrive at a variable or constant rate depending upon the source characteristics. Monitoring applications are typical users of streaming data. Consider a monitoring application that keeps track of items of interest such as overhead transparency projectors and laptop computers, using electronic property stickers attached to the objects. Here the data generating sources are sensors mounted in the ceiling and the GPS system. Now a security administrator would be interested in getting notified if any item is out of place. Also the last noted location of the item would be of interest [1]. Another application area could be a stock ticker application where a typical user might want to monitor a series of stocks in a given range of values. MavHome [2] is a project with a number of sensors that act together in maximizing a metric for a home/environment based on inhabitants use of the devices. MavHome generates streaming data that needs to be processed for prediction and other needs. All the devices in the MavHome system are constantly generating stream data. Monitoring abnormal behaviors in devices such as thermostat would be of interest.

Checking for the arrival of a particular person and then personalizing the home based on his/her requirements is another example.

These are a few examples of stream-based applications. Others include security, telecommunications data management, manufacturing, sensor networks, web applications, stock and sports ticker and military based applications (like monitoring soldiers in a battle field).

Stream applications have their own characteristics, which makes them different from the traditional data oriented applications. Below, we enumerate the salient characteristics of a stream-based application:

- Streaming data are not the results of transaction processing typically done by humans. These are generated from a variety of sources like sensors, GPS, web servers or some other programs.
- Streaming source generates continuous data and has no limit on the amount of data generated by streams.
- Arrival rate of streams can be variable. It can be irregular and bursty.
- Data can be lost or garbled.

In order to provide a timely response to events, some of the applications have a predefined quality of service requirement (QoS). The query requirements of these applications are somewhat different from traditional queries of DBMS. Queries are classified based on how they are submitted and the life span of the query. Queries can be *Adhoc* or *Predefined* (how they are submitted) or can be *One time* or *Continuous* (life span) [3]. Adhoc queries are submitted on the fly. The nature of adhoc queries adds

a new dimension to the optimization of the system as a whole. In contrast *Predefined* queries are known to the system and hence it is possible to generate an optimized plan for all queries in the system. On the other hand, *One Time* queries have a short life span. It operates on a limited number of tuples that is defined in the query. Comparatively *Continuous* queries are long running. These queries are evaluated continuously as the data streams arrive. Output is produced incrementally and continuously at the end of every window. Majority of the streaming queries are continuous. *Predefined* and *Adhoc* queries can be *One Time* or *Continuous.*

Traditional DBMS were not designed to support the requirements of stream-based applications. They were designed to satisfy to the needs of business data processing applications. Some of the reasons that make DBMS not applicable for Stream data applications are:

- DBMS uses a store and query approach. It will be very difficult to store each and every streaming data into the system. This process is time consuming.
- It cannot give a real time response to streaming queries, which is an important requirement for streaming applications.
- Some of the operators of DBMS are blocking in nature. For example Join and Aggregate operations require the entire data to produce output. This is not possible in a stream-based application.
- DBMS has no provision for supporting quality of service (QoS) requirements of the user.

- Query optimization and scheduling strategies of DBMSs were not designed for optimality but to avoid naïve and inefficient strategies.
- Trigger support in DBMSs cannot scale to large number of triggers.
- DBMSs assume data elements to be synchronized but most of the stream-oriented applications are asynchronous in nature.
- DBMS tends to produce exact answers. But stream data can be lost or garbled there by producing approximate answers.

This lack of support from DBMS for stream-oriented applications prompts us design a system, which supports these requirements. This system is referred to as Data Stream Management System (DSMS) in the literature. Some of the requirements of a DSMS are:

- DSMS should not store streaming data on secondary storage.
- Main memory is the primary storage used in a DSMS.
- It should support asynchronous nature of stream.
- It should have a language to specify long running and window based queries.
- Quality of Service (QoS) requirements of streaming applications should be supported. QoS should drive optimization and scheduling techniques and not I/O as in traditional DBMSs.
- DSMS should output continuously and incrementally.
- New set of operators must be defined that are non-blocking. Join and Aggregates operators should support the window based query requirement.

The primary advantages of this system are:

1. System can operator in a low main memory environment. This is because buffers support both main memory and secondary memory operations. Users sees buffer as a structure for storing tuples and does not know which tuple will go in main memory or secondary memory.
2. System provides accurate results, as there is no data loss. This is because use is just not restricted to main memory for storage.
3. The window representation proposed is a generic window representation. It can be added as part of standard SQL statement. It can represent physical as well as logical windows. Also it can represent snapshot, landmark and sliding window queries both in forward and reverse direction.
4. None of the papers referred in the thesis provides exact implementation details of the stream database system. This thesis tries provides a platform to build a complete stream management system.
5. It shows that exploiting the overlapping nature of windows in case of long running queries (sliding window) can be prove efficient compare to the standard approach of processing a window from start to end. This is show by designing and implementing two variations of Hash Join one with recompute and other variation being reuse.
6. System is designed such that it can operate without a scheduler. It schedulers an operator when data is available and suspends it when

there are no tuples. The rest of scheduling is kept at the discretion of the base operating system. This is referred to as flow based scheduling in this literature.

Some of the limitations of the system are.

1. It does not support historical queries. Hence we cannot specify query windows in reverse direction.
2. Optimizer is proposed in the architecture but it is not designed and implemented.
3. Alternate Plan Generator needs to implement to support global query optimization.
4. This system is prone to failure and has no capability to recover to the point where the system had failed.

Chapter 2

RELATED WORK

This chapter discusses related work. Some of the systems are only for Data Stream Management (e.g. Aurora [1]) where as others are based on specific applications (e.g. Cougar[5].)

2.1 Aurora

Aurora's prime functionality is to process streams based on the configuration set by the *application administrator*. Aurora is a data flow system and uses the primitive box and arrow representation. Tuples flows from source to the destination through the operational boxes. Aurora's query algebra supports seven primitive operations, some of the important ones being select, aggregate, split, union and resample. This architecture supports continuous queries for real-time processing, views, and ad-hoc queries. It maintains historical storage in order to support adhoc queries. All these query types are supported using the same set of operational blocks. Quality of Service is associated with the output. It is specified in terms of a two-dimensional graph that specifies the output in terms of several performance-related and quality-related services. It is the QoS that determines how resources are allocated to the processing elements along the path of query operation.

Input in Aurora is through a GUI. Input begins at the top of the hierarchy and makes use of the *zoom* capability to further assist in the design. Users can then query the system. Facilities for debugging and single stepping are also provided.

Aurora has dynamic optimization policies, which changes the data flow computation graph (or network) at run time to improve the performance. Optimization is based on the types of queries running in the system. It does not optimize the whole network at once. But does it in parts by considering a portion of the network at a time.

Aurora has a Storage Management module, which takes care of storing all the required tuples for its operation. It is also responsible for queue management. Scheduler is designed to cater to the needs of a large-scale system with real time response requirements. Scheduler deals with operators unlike Eddies [4] that deals with tuples for scheduling.

2.2 Streams

Streams (*STanford StREam DatA Manager*) [3] is a prototype implementation of a complete Data Stream Management System being developed at Stanford. A modified version of SQL has been chosen for the query interface. It allows the user to specify sliding window queries in SQL with an explicit referral to timestamps. It assumes that with explicit timestamp, tuples will be delivered in an increasing order. It supports logical and physical windows. Logical windows are expressed in terms of tuples and physical windows are expressed in terms of timestamp. STREAMS support continuous queries but have not addressed the issue of ad-hoc queries. The system generates a query execution plan on the registration of a query that is run continuously.

Query execution plan is nothing but a set of operators connected by queues. Operators make used of *synopsis (an internal data structure)* to store intermediate results. System memory is distributed dynamically among the synopsis, queues in query plans along with buffers handling streams coming over the network, and a cache for disk-resident data.

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

STREAMS has a central scheduler that has the responsibility for scheduling operators. The scheduler dynamically determines time quantum of execution for each operator. Period of execution may be based on time, or on number of tuples consumed or produced. Different scheduling policies are being experimented.

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

2.3 The COUGAR Sensor Database System

COUGAR is specifically targeted to meet the requirements of sensor-based applications. This system is designed considering the characteristics of sensor and their applications. Some of the major challenges facing the COUGAR system are account for the failures of sensor and its communication, uncertainty of sensor data and distributed execution of query without global knowledge of the sensor network. COUGAR focuses on a distributed approach toward query processing where in the workload determines the data that needs to be extracted from the sensors. COUGAR is based on the Cornell PREDATOR object relational database system.

Sensor data is considered as a combination of stored data and sensor data. Stored data are represented as relations and sensor data are represented as time series based on a sequence model. Long running sensor queries are supported by this system. Sensor queries are defined as an acyclic graph of sequence and relational operators.

In COUGAR, signal-processing functions are represented as an Abstract Data Type (ADT) functions. Sensor ADT's are defined for sensors of the same type (e.g. temperature sensor, seismic sensor etc). Public interface to an ADT corresponds to the signal processing function supported by a type of sensor. Sensor queries are SQL like queries with a little modification where in ADT can be included in the SELECT or WHERE clause of the query. Query processing takes place on a database front end where as the signal-processing functions are executed on the sensor nodes involved in the query. On each sensor a lightweight query execution engine is responsible for executing signal processing functions and sending data back to the font end.

COUGAR assumes that there are no modifications to the stored data during query execution that is made sure using Two Phase locking. Virtual Relations are introduced in order to overcome the disadvantages of ADT.

2.4 Fjord: Architecture for Queries Over Streaming Senor

Fjord [6] is sensor data processing architecture for data intensive sensor-based applications. It provides a low-level database engine support required for sensor centric data-intensive systems. The main focus of the system is to provide an efficient, adaptive and power sensitive infrastructure. This system supports the Berkley Highway lab to monitor traffic conditions with the help of sensors that are deployed on Bay Areas freeways.

Fjord's operators export an iterator like interface and are connected together via local piper or wide area queues. It provides support for integrating streaming data that is pushed into the system with disk-based data that is pulled into the system.

Each machine involved in the query runs a single controller in its own thread. Controller accepts message to instantiate operators, connect local operators via queues to other operators that may be running locally or remotely. Queues also export an iterator like interface irrespective of whether the operators are local or remote. This way it makes the operator ignorant of the nature of their connection to remote machine. Each query has its own thread, which is multiplexed between local operators via procedure calls in case of a pull base architecture or via a special scheduler that also control the input and output of data through the operators.

Operators are the primary functional unit of the system. Each operator owns a set of input and output queues. It reads data from the input queue, performs the required operations and directs them to the output queue. No processing takes place in the queues. Stale data are discarded from queues based on the requirements of the applications. It supports non-blocking operators such as selection and projection and blocking operators such as join and aggregate. A main memory symmetric hash join has been implemented which maintains a hash table for each relation. Window based operations are supported for blocking operators. Considering the nature of streams, Fjord provides an optimization by combining multiple queries in a single Fjord. This way a significant amount of computation and memory can be shared there by improving the overall performance of the system.

Sensory proxy is a prime component of this architecture. It acts as an interface between the system and the sensors over which the user will poise the queries. It shields the sensor from having to deliver data to hundreds of interested users. It adjusts the sampling rate of the sensor based on the current condition of the system there by preserving the battery life of the sensor, which is one of its prime advantages. It can also direct the sensors (*smart sensors)* to aggregate samples in a predefined way there-by reducing the data communication.

2.5 PSOUP

PSOUP [7] adheres to the needs of a different class of streaming applications where in adhoc queries and intermittent connectivity also requires the processing of data that arrives prior to query submission or during the period of disconnection. Data

recharging and monitoring applications are the primary targets of this system. PSoup builds on adaptive query-processing techniques developed in the Telegraph project [8] at UC Berkeley. It combines the processing of ad-hoc and continuous queries by treating data and queries in a symmetric fashion there-by allowing new queries to be applied to old data and new data to be applied to old queries. In order to support disconnected operation and to improved data throughput and query response time, PSOUP partially pre-computes and materializes results.

User interaction with the system begins by submitting a query. On registration, PSOUP returns a handle to the user, which can be used for further communication of results. Queries are specified in the form of SELECT-FROM-WHERE clause. It also has provision to specify a window using the BEGIN and END clause. System is flexible to adapt to logical windows (*windows specified in terms of number of tuples*). System can be easily extended to support different sized windows for each stream.

Query and data are stored in structures called State Modules (SteMs). There is one Query SteM for all the queries in the system and on Data SteM for each data stream. PSOUP supports both historical and ad-hoc queries. For historical queries, whenever a query is registered to the system it is entered into the Query SteM and then probed to the Data SteM. For continuous queries, on the arrival of a new data item, it is inserted into the Data SteM and is used to probe the Query SteM. Results of probing are stored in a Result Structure. User queries are answered from this data structure. It is this data structure that allows supporting the period of disconnection with the system.

A generalized symmetric join that accepts more than two streams, is used for joining queries multiple data streams.

2.6 NiagaraCQ

NiagaraCQ [9] is a system that mainly focuses on supporting continuous query processing over multiple, distributed XML files. It is mainly for web-based users as they can scale to a very large number. It is the continuous query sub-system of the Niagara project, which is a net data management system being developed at University of Wisconsin and Oregon Graduate Institute. It takes advantage of the fact many web based queries share similar structures. Groping similar structures can save on the computation cost, memory cost and I/O cost. Moreover grouping of selection predicates can eliminate a large number of unnecessary query invocations.

NiagaraCQ uses a novel approach of group optimization. It uses an incremental group optimization strategy with dynamic re-grouping. When a new query arrives, the existing groups are considered as possible optimization choices instead of re-grouping all the queries in the system. The new query is merged into existing groups whose signature match that of the query. Another advantage of this system is that it uses a query split scheme that requires very little modification to a general-purpose query engine. After the signature of a new query is matched, the sub-plan corresponding to the signature is replaced with a scan of the output file produced by the matching group. This optimization process then continues with the remainder of the query tree in a bottom-up fashion until the entire query has been analyzed. NiagaraCQ also supports grouping of change-based and time-based queries in a uniform manner.

NiagaraCQ defines a simple language for creating and removing continuous queries. Continuous queries can be written by combining XML-base queries with timing clause. These queries are deleted from the system on the expiration of the time specified in the query. Since it is a large-scale system, not all the information required by the system can fit in main memory. Hence caching is used to obtain good performance with a limited amount of memory. It caches query plans, system data structures and data file to improve performance.

NiagaraCQ is developed using Java (JDK1.2) and a validating XML Parser from IBM. Main components of the system are

- A continuous query manager that provides a query-interface to submit the query and later executes the given query.
- An optimizer to perform group query optimization.
- An event detector to detect time event and changes of data sources.

Chapter 3

STREAM DATABASE ARCHITECTURE

Data Stream Management System (DSMS) being developed for processing queries from MavHome, provides a query execution platform for streaming based application. This is a complete system where in a query, submitted by the user, is processed at the server and the output is returned back to the user. It is a client server architecture. Following are the important modules of DSMS.

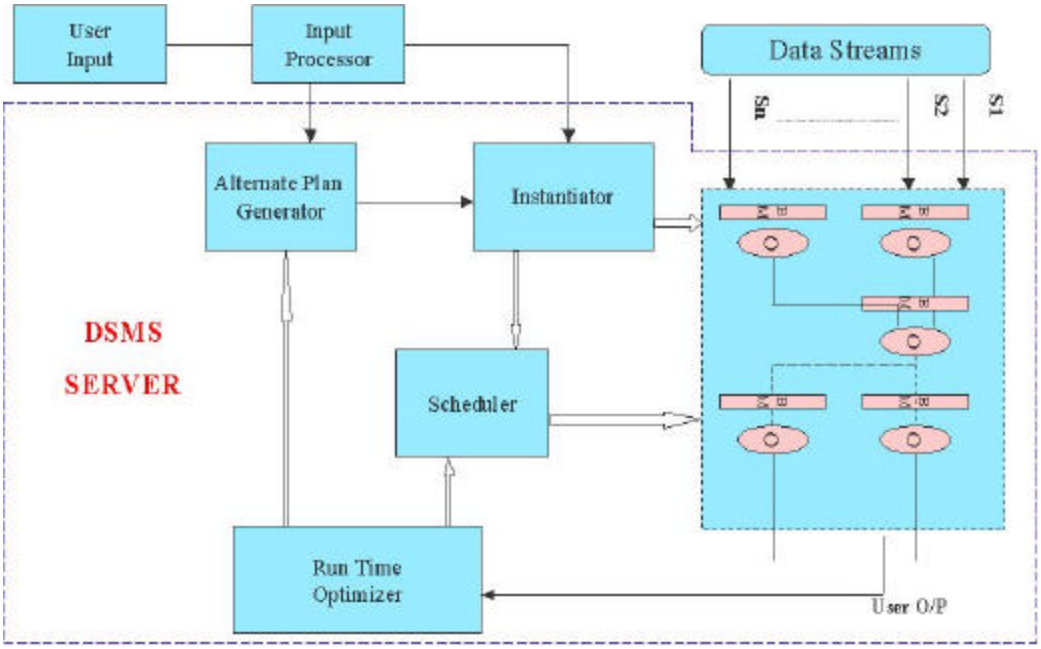

Figure 3.1 DSMS Architecture

3.1 Client (User Input)

DSMS provides a predefined set of services to its users. It is command driven and protocol oriented. The Server listens to particular port for a command input from the client. Each command is associated with a protocol that defines the communication between the client and the server for the service to be successfully offered. Protocol clearly specifies the input it expects from the client, the function it is going to perform using the given input and the output that will be sent to the client. Client needs to be well aware of the protocol in order to obtain the desired service from the DSMS server. In general a client should provide the following functionality.

- Provide an interface (preferably a GUI) from which user can choose one of the services offered by DSMS.
- Based on the service chosen, it should provide an interface to collect the required input from the user.
- Covert the input from the user into a form specified in the DSMS protocol.
- Communicate with the server by sending the command for the service chosen and later follow it with the input constructed for the command.
- Wait for a response from the server. Present the collected response in a user understandable fashion.

Client can be one of the following types:

- It can be a GUI based client developed using Java swing or Java AWT.
- It can be a web-based client that uses Java technology.

- It can be a non-GUI based client where input comes from a predefined source such as a file. This can be used for experimental purpose.

3.2 Operators

Operators are the basic building blocks of DSMS. Each operator is an independent running unit. Operators work in close association with buffers. Query is modeled as a tree of operator connected using buffers. Operator pulls data from input buffers, processes the data and pushed the generated data on to the output buffers. The smallest schedulable entity in a DSMS is the operator. To support this property, operators need a mechanism that allows it to be controlled from external sources (mainly the scheduler). It supports operations such as start, stop, suspend and resume which allows a fine-grained control over its execution. During its life span, operators can be in one of the following four states viz. ready to run, running, suspended or stop.

Each operator has at least (e.g. select, project etc) one input queue and atmost two input queues (e.g. join). It can have more than one output queues as queues may be shared among operators in a query tree. Operators can have a priority associated with it. Priorities can be user defined or generated by the system over the life span of the operator. It helps the system take decisions for fair resource allocation between queries/operators.

Operators are divided into two categories based on the way they process tuples.

1. Window-Based Operators: These are the operators that require certain amount of data before it can generate its output. Hence these operators operate on a

window worth of data and produce their output. Join and aggregate are examples of window-based operators.

2. Non-window Based Operators: These are the operators, which work on a single tuple at a time. They don't need to wait for other tuples. Select, project and split are examples of non-window based operators.

   Following are the operators implemented in DSMS:

- Select (Filter): Select does a conditional filtering of tuples. It picks up a tuple from the input buffer, evaluates the condition and sends the satisfying tuples to the output buffer.
- Project: Project brings out the desired attributes from a given tuple.
- Join: It is a binary join where-in it combines a set of stream based on some pre-specified condition.
- Aggregate: It performs functions such as max, min, count, average on a window worth of data.

3.3 Buffer

Buffers are the temporary storage that is being extensively used in DSMS. They are the bridge, which connects the operators to make the query tree. Data flows from the stream into the buffers, this act as an input to the operators, which produces new data that are again output to some buffer. Hence buffers are very important resource that affects the overall performance of the system as a whole. Buffers are categorized into two types based on how they make use of main memory, which is a very costly resource in this system.

UnBounded Buffer: Entire storage space of unbounded buffer comes from main memory. These buffers are very fast operating one and should be used intelligently as improper usage of this may affect the overall performance of the system.

Bounded Buffer: Bounded buffers have a predefined size up to, which the data will be stored in main memory. If data exceeds beyond the specified size, it will be stored in secondary storage. This whole operation of storing data in secondary memory is transparent to the user. Buffer management policy ensures minimum access/usage of secondary memory.

Operators are the primary users of buffers. More than one operator can be reading from a buffer at the same. But only one operator writes to a buffer at one time. Hence appropriate locking must be ensured to preserve the integrity of data in buffers. Also the process of purging (process of removing tuples from the buffer) is made more complex as there can be more than one consumer of buffer data. Purging logic should make sure that a tuple is removed from the buffer only if all the operators consuming tuples from the buffer have read it.

3.4 Streams

Streams are the primary data source of DSMS. Raw data (tuples) are fed through the streams. They are not a predictable source of data. It can have a constant flow or may be bursty in nature.

Each stream has to be well defined. It should clearly specify all the attributes that are part of the streams, their data types and their position in the stream. Definition of streams differs from application to application. It is the responsibility of the

administrator to define streams before data is sent to the system and queries are defined over it. Stream definitions are stored throughout the lifetime of the system. These are used for query building, condition evaluation in operators etc.

Data from streams are collected into buffers and then used by the query operators to generate results for the query. Buffer size of the base streams should be considerably large. Unbounded main memory buffers are good the performance of DSMS.

3.5 Scheduler

Scheduler is one of the central components of DSMS that controls the execution of operators. Scheduler is executed as a high priority thread running in the system and is started with the Server. It maintains a ready queue, which is a list of operators that are in ready to run state. Based on the scheduling policy, it picks up an operator from the ready queue and runs it for a time quantum. The length of the time quantum is based on the scheduling policy. One of the following conditions may occur during the running state of the operator.

1. Operate may finish its execution. In this case, operator stops its execution and informs the scheduler about it. If the time quantum of the operator is not completed, scheduler releases the execution of the operator and removes the operator form the ready queue.
2. Operator may not have enough data to process. In this case, buffer suspends the operator. Thereafter, operator informs the scheduler about it suspension, which in turn releases its execution and removes the operator from the ready

queue. Whenever some data is available in the buffer, buffer will bring the operator to ready-to-run state by adding it into the ready queue.

3. Time quantum of the operator may get expired. In this case, scheduler suspends the execution of the operator and puts the operator in ready to run state by adding it to the end of the ready queue.

In order to ensure timely response and good performance of the overall system, scheduling policy needs to be chosen carefully. The following two policies are being used in the initial phase of DSMS.

1. Round Robin: Here the ready queue is a simple FIFO queue. Scheduler picks up the top most operators from the ready queue and schedules for a fixed time quantum. Disadvantage of this policy is that queries with high priority will not get higher quantum of time.
2. Weighted Round Robin: This policy aims at giving more time to operators with high priorities. Here operators are scheduled in a round robin fashion but the time quantum chosen is a function of the priority of the operator. Higher the priority higher is the time quanta. This policy avoids starvation as all operators are scheduled in a round robin fashion on a FIFO basis.

3.6 Run Time Optimizer

DSMS is a real time stream management system. It is not only important to produce correct output but also to produce in an efficient manner keeping in mind the Quality of Service (QoS) specification of the user. Also the systems state is constantly changing during its lifetime as number of queries may be added, delete or modified.

This dynamic nature of the system brings forth the need for a module that not only monitors the systems performance but also comes up with a plan that can produce an efficient output. Run Time Optimizer is a module to accomplish this. Its prime responsibilities are.

1. Make sure that the QoS specification for queries provided by the users are met. For this it constantly monitors the output of each and every query and makes sure it is does not violate the value specified by the user. If so it should take measure to bring it back to an acceptable value.
2. Monitor the performance of the system and optimize it as needed.

For this it may take the help of Alternate Plan Generator to generate an alternate plan for a query that might not be efficient by itself but may prove to be efficient when combined with other queries in the system. In order to achieve its goal, it can dynamically increase or decrease the priorities of the operators. Run Time optimizer is currently not supported in the system.

3.7 Alternate Plan Generator

Initially user submits a query plan to the system. This is an executable plan but may not be the most efficient plan to run. Several equivalent plans may be available for this query. In order to generate these plans, a separate module is required. This module is the Alternate Plan Generator.

Alternate plan generator takes a base plan submitted by the user, and generates one or more plan that can generate the same output. It can then be merged with the existing plan with the system to conserve resources such as computation and memory.

This dynamic regrouping [10] is needed to satisfy the QoS requirements. Generated plans can be used for one of the following purposes.

- Alternate plans can be used in order to obtain an initial optimal plan for the query.
- Generated alternate plans can be used by the Run Time Optimizer to generate an overall efficient global plan.

Alternate Plan Generator is another topic to be addressed in the future.

3.8 Instantiator

A plan object is tree of operator nodes. Each Operator node has enough information for the corresponding operator to be instantiated. Plan object is one that is initially submitted by the user or generated by the Alternate Plan Generator (using the initial plan). The role of Instantiator is to take a plan object and convert it into operator objects that can be scheduled. A complete cycle for query instantiation is as follows:

- It takes in a plan object and traverses in a bottom up fashion so that the operators are instantiated in the order in which data flows (from leaf to the root).
- It reads the information from the operator data node, identifies the operator, and converts the data into a form required by the operator and instantiates the operator.
- It then adds the operator to the scheduler's ready queue.

3.9 Server

DSMS server is a TCP Server that listens to a particular port. It provides a set of services to the client. Each service is associated with a command. Following the

command it executes the protocol for the service. The set of commands supported by the server is given below:

- Execute a query.
- Register a stream.
- Send all streams to the client.
- Send a particular stream to the client.

Additional commands can be added to the server. Apart from supporting the above commands, server also takes care of starting all the major components of DSMS. It starts the scheduler as a high priority process. Alternate plan generator is instantiated. Run Time Optimizer is started as a separated thread. Server also maintains the data structure to store and retrieve stream definition. It is also a place to hold data structures that are used for various house keeping activities like experimental results, configuration data etc.

Chapter 4

DESIGN

4.1 Schema:

Stream Database supports more than one stream. Each stream feeds tuples into the system. Each stream has a set of attributes and data types. Stream information needs to be stored in the system during the processing of queries over those streams. Collectively the set of attributes and the data types, which represents a stream, is called a *schema.* The concept of schema is analogous to the concept of *tables* in DBMSs.

4.1.1 Requirements of schema for stream databases are:

A schema represents a stream in the system. Schema should be able to store the attribute name, its data type and its position in the tuple. This information set should be expandable for future purpose. System should be able to support large number of schemas. Schema can be either user given or system generated. System generated schema are used for storing temporary information, which is generated as a part of query. These types of schemas are referred as *intermediate/temporary schema.*

Details of schema should be accessible based on attribute's name or its position. It should be easily accessible by all modules of a stream database, which is a prime requirement for operators. Operators such as SELECT, PROJECT and JOIN accesses the schema to bring the input into a form, which is efficient for processing. For example, the attribute name in the PROJECT operation are converted into a list of

positions using the schema so that the PROJECT operator can use the position information on the input tuples and extract the required field.

Consider an example of MavHome data. Here a stream is constantly generated that represents the activities of various devices in the room. In order to represent this stream, a schema needs to be created. Figure 4.1 shows the skeleton of the schema.

Position	Attributre Name	Data Type
1	TransID	String
2	DeviceID	String
3	Status	String
4	PropertyValue	String
5	Command Source	String
6	SrcTimeStamp	number

Figure 4.1 Example of MavHome Transload Schema.

4.1.2 Schema Design:

A dynamically growing list is recommended to store the schema information of a stream database since there is no limit on the number of schemas stored in the system. Each row in the list stores information for a schema, which can be expanded in the future. In order to support this expansion the schema information structure need to be dynamic in nature. Schema information can be accessed based on attribute's name or its position. To support these access-mechanisms the schema attribute information is stored as a separate entity. Moreover, the total number of information stored for a schema attribute should be expandable for the future, so a structure that grows dynamically is needed. Hence all the information about an attribute of a schema such as attribute's

name, position, data type, etc is stored in a separate structure. This way there would be *N* structures for each of the *N* attributes of the schema.

A separate structure should be maintained that provides access to attribute information based on attribute name. Similarly another structure is needed that will provide access to attribute information based on attribute's position. Both of these structures should be accessible using the schema name.

Figure 4.2 shows the representation of schema in terms of its structure described above.

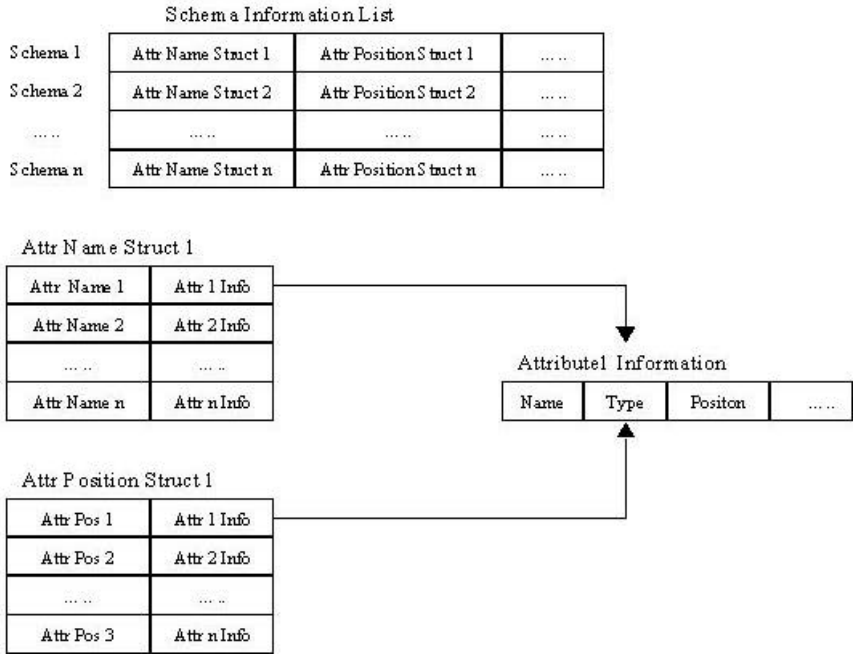

Figure 4.2 Schema Representation

4.2 Buffer Management

4.2.1 Requirements for Buffers and their Management:

The following are the requirements for a buffer in a DSMS.

- DSMS should support operations in limited main memory environment.

- Buffer Management should be such that the use of secondary storage should be minimized, which helps in improving the overall efficiency of the system.
- Buffer Management should be such that each buffer should support multiple operators.
- Buffer should support flow based scheduling. (In this scheduling scheme, an operator suspends itself when it has processed all tuples from the input buffer. After that it is the responsibility of buffer to resume the operator when new tuple(s) arrive.)
- It should provide an efficient purging mechanism for removing stale tuples from the main memory.
- It should have a policy that decreases the overhead of purging secondary storage tuples.

4.2.2 Buffer Design:

Buffers are the basic building blocks of a DSMS. They are the primary storage structure of DSMS. Buffers are used for storing tuples that flows from the streams (source) to the end user (destination) through a set of operators. Buffers are non-processing components of a DSMS; i.e. the state of tuples is not changed while they are in buffers. Tuples flow in a "first in first out" (FIFO) fashion through the buffers. Hence buffers are implemented as FIFO queue.

Buffers are used for the following two purposes:

1. Buffer acts as storage for the streams that are the primary sources of tuples. As shown in Figure 4.3, each stream is associated with a buffer.

2. In addition to streams, operators also use buffers. Operators read from a buffer, process the data and write the output tuples to a buffer.

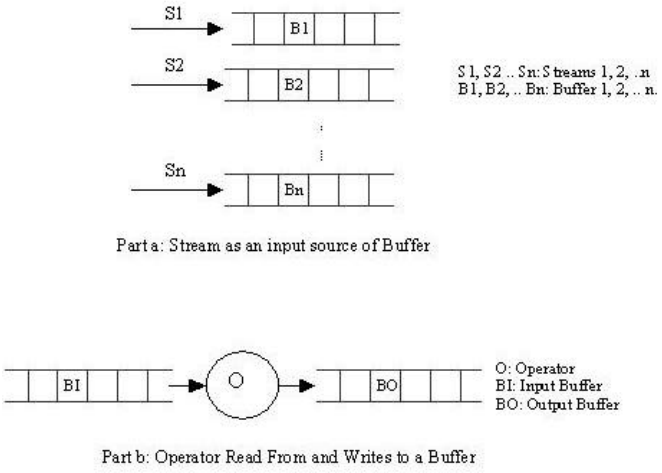

Figure 4.3 Buffer Usage in DSMS

One of the distinguishing features of a DSMS is that it provides exact answers and not approximate answers [11]. In order to achieve this feature, all tuples are considered, which means DSMS has to provide a mechanism for storing every tuple in limited memory environment. Based upon this requirement buffers are classified as follows:

Infinite Main Memory Buffer: In this case, all the tuples are stored in the main memory. These buffers are very fast in operations as compared to its counterpart. There is no limit on the amount of tuples that can be stored in these buffers. Hence, in a limited memory environment, these buffers should be carefully used. Stream Buffers are good candidates for main memory buffers as they control the process of feeding the tuples to the systems.

Limited Main Memory Buffer: Limited Main Memory buffers have a limit on the number of tuples that can be stored in main memory. If the number of tuples stored in main memory exceeds the specified limit, the remaining tuples needs to be stored on secondary storage. Tuple latency of limited main memory buffers can be very high as compared to its counterpart because of the use of secondary storage device. One of the vital requirements of this buffer is the choice of the number of tuples that can be stored in main memory (limit). Limit boundary should be chosen in such a way that the number of tuples stored in secondary memory is less. It is possible that no tuples land in the secondary storage thereby providing better performance. There are chances that less amount of tuples get stored in the secondary memory and these tuples are fetched into the main memory of the buffer while the consuming operator is in suspended state. This gives the consumer an illusion that it has an infinite main memory buffer.

Main memory and CPU time are the important resources in a DSMS. The use of main memory is directly related to the choice of buffer type and the selection of memory limit on secondary buffers. There are certain factors that need to be taken into account while selecting these buffer parameters. These factors are:

- Input rate: Rate at which tuples are enqueued into the buffer.
- Tuple consumption rate: Rate at which tuples are read from the buffer. Since buffers can have more than one consumer, the rate of the slowest consumer must be considered as the output rate, as tuples cannot be removed from the buffer until all consumers have read it.

It is difficult to predict these parameters in streaming applications as the choice of buffer size and its type is made dynamically. The following are the two places at which these choices can be made.

- Buffer type and its limit can be decided when buffers are created. Based upon the nature of streams, priorities assigned to operators, type of queries and current system conditions, a heuristics should be develop to predict the input rate and the tuple consumption rate of the queries.
- As queries are added, removed or completed in the system the load on the system changes dynamically. Also, the QoS requirement forces the optimizer to change some parameters in the system in order to provide a desired level of QoS to the users. Because of these changes in the system, the input and output rate of operators gets changed. Hence, we need to reconsider the buffer parameters to provide an optimal solution.

The above two responsibilities lies with a runtime optimizer, which is a future work to this project.

4.2.3 Multiple Operators Reading from One Buffer:

There are two alternatives for associating buffers with operators, which are as described below.

One Operator per Buffer: Here only one operator is associated to read from a buffer. The advantages of this method are.

- It does not require a complex logic for buffer management.

- Also, the average tuple latency will be less as a tuple stays for a minimum amount of time in the buffer.

But the disadvantages are:

- If many operators are reading the same data from different buffers, there will be considerable amount of main memory that will be wasted in duplicate tuples. The amount of memory wastage is directly proportional to the number of operators reading the same data.
- Also, it requires a considerable amount of time to create the duplicate buffers.

Multiple Operators per Buffer: To avoid the disadvantages of one operator per buffer, another scheme is considered, which supports multiple operators reading from the same buffer.

4.2.4 Design of Multiple Operators Reading from a Buffer

Operators are the only consumers of tuples from the buffers. There can be more than one operators reading from a buffer but the consumption rate of each operator is different. In order to support these different rates of tuple consumption, buffer maintains a pointer *(currentUnReadElement)* for each operator associated with the buffer. This pointer points to the next tuple in the buffer that has not been read by the operator. Beyond this pointer, operator has read all elements from the buffer.

Initially each consumer operator has to register with the buffer. Buffer maintains a table, which contains the operators registered with the buffer and its *currentUnReadElement pointer*. On registration *currentUnReadElement* for that

operator is initialized to zero. This pointer needs to be adjusted when elements are dequeued from the buffer.

A design alternative was to keep this pointer in the operator itself. In this case, during the process of purging tuples, buffer had to communicate with all the operators that were registered with the buffer to obtain the latest value of *currentUnReadElement*. Also, after purging logic, buffer would have had to send the amount of tuples that were purged back to all registered operators so that the value of *currentUnReadElement* can be adjusted appropriately. Furthermore, it needs to make sure that registered operators are not reading tuples from the buffer while the buffer is processing the purging logic. Therefore, to avoid all these possible communications between operators and the buffer, the *currentUnReadElement* was kept within the buffer.

Operator	CurrentUnReadElement Pointer
O1	1456
O2	2378
⋮	⋮
On	645

Figure 4.4 CurrentUnReadElement Pointer

4.2.5 Purging Logic:

Purging logic is a complex process as more than one operator is registered with the buffer. Before dequeuing a tuple from the buffer, purging logic needs to make sure that the element is of no use to any of its consumers. In short, all operators have read the element. For this, the purging logic makes use of *currentUnReadElement* pointer of

each buffer. It finds out a point in the buffer until which all the operators have read from the start of the buffer. To do this it finds the lowest element from the table shown in figure 4.4. For example, 645 would be the point in the buffer until which all elements can be safely deleted. It then dequeues all elements to that point.

Another important decision is to determine when to call the purging logic. Purging logic is called whenever an operator reads a tuple using its *currentUnReadElement* pointer. Just peeking into the buffer does not require the calling of purging logic.

4.2.6 Buffers Support for Scheduling:

Scheduling of operators is done with the help of a scheduler. Each operator is allowed to run for a specific time quantum. Once the time quantum is elapsed some other operator will replace it based on the scheduling policy. There is another policy called flow-based scheduling, which relies on the scheduling of the base operating system. In this policy, an operator is scheduled only if the input tuples are available for processing. Rest of the scheduling is kept at the discretion of the operating system. Hence in this scheme a query tree is instantiated in a bottom up fashion and operator are scheduled in the same order. The leaf operators, which get data first, are scheduled first. If data is not available, operators are suspended.

In all the policies of scheduling, if no input data is available, the operator suspends it-self. This is an intelligent way of saving costly CPU resources. But to support this policy, we need a mechanism through which the operator can be

rescheduled. This requires a communication between the operator and its input buffer. The process that is carried out in order to support this scheme is given below.

- If no data is available in the input buffer, operator will suspend itself. The reason is the lack of tuples in the input buffer, therefore it informs the buffer of its suspension.
- Buffer adds the operator to the suspension list. This is a list of operators that have been suspended because no data was available in the buffer.
- When new tuples are enqueued in the main memory buffer, it processes the suspension list and one of the two things will be done based on the scheduling policy used. Buffer resumes the operator picked from the suspension list. This is done in case of flow based scheduling. For other scheduling policy, buffer puts the operator in ready to run state and the scheduler will schedule it based on the policy used.
- Finally Buffer removes the operator from the suspension list.

4.2.7 Buffer Management:

The objective of buffer management is to provide an illusion to the user that it has an infinite memory at its disposal. It is an easy task in case of infinite main memory buffer, which has a huge amount of main memory. The real problem comes with limited main memory buffers. For limited main memory, there is an upper limit on the number of tuples that can be stored in the main memory. If tuples exceed this limit, they are stored in secondary storage buffers. Producers and consumers of the buffers should be given an interface whereby they can store tuples in the buffer and retrieve the stored

tuples from the buffer. The information that the tuple is stored in or retrieved from main memory or secondary memory should be transparent from an operator's viewpoint.

Each buffer is associated with two secondary storage files. In traditional log based applications, if an application wants to read some data from the file, it has to first do a sequential scan on the file to reach the point where the object is stored. This can prove to be an unnecessary overhead of scanning through the file before reaching the actual data. This overhead is directly proportional to the file size.

To avoid sequential scan of log files, this design brings in a mechanism of indexing into the file and reading only the required object from the correct position. All the tuples in the buffer log file are stored as serialized byte. In order to get a particular tuple from the buffer log file, it would require the exact starting position of the byte stream for the tuple and the byte size of the tuple. This information is obtained at the time of writing the tuples into the buffer log files. Before writing into the log files, this information is stored in a main memory index table. As shown in the figure 4.5, each entry into the index table consists of a Sequence Number, the byte size of the tuple and its offset into the log file. For each buffer log file there is an associated index table.

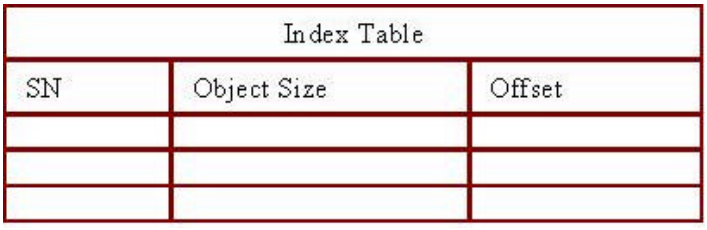

Figure 4.5 Index Table

4.2.7.1 Need for Two Log Files:

Lifetime of a buffer is associated with the lifetime of a query, an exception to it being stream buffers whose lifetime is equal to that of DSMS. For continuous long running queries buffer lifetime is very large. Hence more and more number of tuples gets accumulated in the buffer log file over the life cycle of the query. We should have a mechanism to purge these tuples; otherwise it will not allow new tuples to be brought into the buffer from secondary storage. One method of doing this is removing a tuple from the log file after it has been used. But this may prove to be very costly because of the following two reasons:

1. The cost of purging individual tuple is high when secondary storage is involved.
2. It also requires adjusting the offset of all the remaining tuples in the index table by the size of the tuple that was purged.

The cost of the above two increases with number of tuples in the buffer thereby making the whole process inefficient.

In order to make the purging logic cost effective, the concept of two buffer-log files is introduced. Here we keep storing the tuples into the first buffer log files till a predefined limit is reached. Once that limit is reached, the tuples are stored in the other buffer log file. The buffer manager starts reading tuples from the first secondary storage log file as soon as enough memory is available in the main memory buffer. Then the buffer managers switches the reading process to the next log file the moment all tuples are read from the first file. After this the first file can be cleared quickly by reopening it into write mode. This way we can delete all tuples from the file with negligible cost.

Also, the corresponding index table needs to be cleared when the buffer log files are cleared.

A separate process is used for reading tuples back from the secondary storage file. This process is invoked when the main memory buffer is empty by x% (where x is a configuration parameter) of its allotted limit. This way we can minimize the interference of secondary storage and increase the overall efficiency of the system. The process of reading tuples from the file should initiate from the first file and then proceed to another. It should also take care of the process to switch the reading to another file when all tuples from the current file are read and still there is some place left in the main memory buffer. The whole process should be repeated until main memory is exhausted or there are no tuples left in the secondary memory.

Figure 4.6 depicts the buffer management policy used for DSMS.

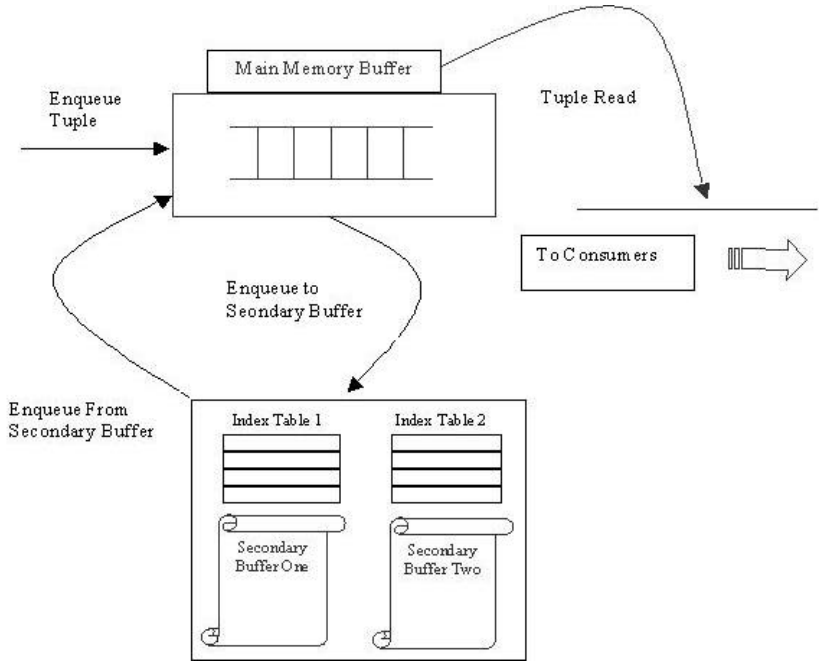

Figure 4.6 Buffer Management in DSMS

4.2.8 Operations on Buffer: The operations performed on a t buffer are

4.2.8.1 Enqueue:

Enqueue provides an interface for the producers of the buffer to store tuples in the buffer. Producer of the tuple is not aware that the tuple is going to main memory or to the secondary memory. Enqueue compares the size of the buffer with its specified limit. If the limit of main memory is crossed, enqueue stores the tuple in the appropriate secondary storage log file. It also holds the responsibility of creating the log file only when it does not exist. Before storing the tuple in secondary storage, enqueue first places the tuple information in the index table. Then from the index table it knows the offset of the tuple in the file. Using this information, it stores the tuple in the appropriate log file.

4.2.8.2 Dequeue:

Dequeue removes a tuple from the main memory. It is primarily called by the purging logic. Another responsibility of dequeue is to start the process which reads tuples from secondary storage into the main memory buffer. It checks if the main memory buffer is x% empty (where x is a value read from DSMS configuration parameters) before starting the external memory read process. Deque ue also adjusts the *currentUnReadElement* pointer of all registered operator after removing an element.

4.3 Operators

Operators are the one of the key components for query execution for stream database. Traditional DBMS database operator works on the data set that is available

before execution of the operator. Here the flow of tuples is between operators. One operator passes on a tuple directly to another operator in the hierarchy. In short, the flow of tuples is synchronous. Also some of the operators such as JOIN and AGGREGRATE are blocking in nature. In this case, the entire set of data is expected before producing the output. Moreover the applications are such that the data set is available before hand for the operators to process. For example in order to compute a JOIN, all the tuples from both the tables are available for processing and this way the joining techniques had a different focus and scope for improvement.

Stream database has totally different set of characteristics, which does not suit the design of traditional DBMS operators. Moreover the design of traditional DBMS operators was done keeping in mind the query processing of business processing applications that had a store and query approach. Streaming applications require incremental and continuous delivery of output to the users. Moreover there is no limit on the number of tuples that are fed to the system. These properties require the operators to be non-blocking.

Tuples are asynchronous in nature but the operators can process tuples at a fixed rate. Hence there comes the requirement to convert the asynchronous streams to one that can be easily fed to the operators. Buffering tuples into stream buffers helps to convert asynchronous streams to synchronous one. Operators reads data from Stream buffers, processes it and outputs the data to the output buffers. A buffer provides a connecting bridge between two operators to route the tuples from the stream buffers to the end user.

4.3.1 Requirements of Stream Operators:

- Operator should be a schedulable entity. Execution of operator can be controlled by external components such as Scheduler and Buffers.
- Operators can read either from one or two input buffers. Operators like SELECT and PROJECT requires only one input buffer where as JOIN operator requires two input buffers.
- Operators should be able to output the same tuple to more than one output buffer. This might be required when a single operator can server more than one query. This feature will come into play when common operators between queries are combined in global query optimization.
- Operators should be able to support flow-based scheduling. In Flow based scheduling, an operator should start processing with the arrival of tuples in input buffers and suspend itself when all the tuples in the input buffer are processed. Hence there should be a provision in each operator to suspend its execution when all tuples have been read from the input buffers.
- Operator should have the capability to stop its execution when the end query condition is reached. In this manner the entire query will be executed on it own. This way there is no need for the scheduler to monitor the stopping of the query.
- Each operator should have the provision to support Quality Of Service (QoS) requirements specified by the user.

4.3.2 Operator Model:

An operator model is designed to meet the requirements of any stream operator. All operators should be in accordance with the operator model.

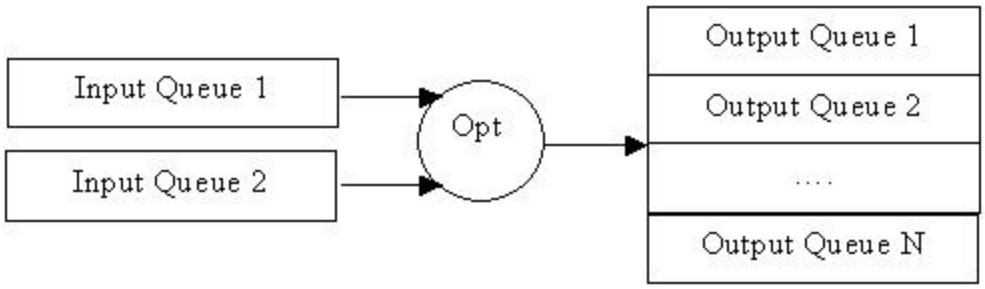

Figure 4.7 Operator Model

Each operator reads data from input buffers, processes it and outputs the generated tuples to the output buffers. Two important properties that each operator possesses are state and priority that are essential for scheduling [8] stream operators.

1. State: State represents the current execution state of an operator. At any given time after instantiation, an operator can be in one of the following state:
    a. Ready To Run.
    b. Running.
    c. Suspended.
    d. Stopped.
2. Priority: Priority defines the precedence of execution of the operator. Priority is one of the ways of controlling the QoS requirements set by the user. Priorities range from zero to nine with zero being the lowest priority and nine being the

highest. If the user initially sets no priority, a default priority of five is given to the user.

4.3.3 Operator State:

Lifetime of an operator spans across four execution states. The state of execution depends on the following parameters.

1. Availability of tuples: Since operators supports flow-based scheduling, they can run only when tuples are available. If no tuples are available, an operator will suspended its execution and waits for tuples to arrive. On the arrival of tuples, buffer brings the operator into "running state" or "ready to run" state depending on the scheduling policy.
2. Availability of CPU resources: CPU cycles are the primary resources for operator execution. If CPU cycles are available to an operator, it can be in running state. Based on the scheduling scheme or availability of input tuples, the CPU resources can be switched to other operators and bring the current operator to "ready to run" or "suspended" state.
3. Scheduling Policy: Scheduler is a component that controls the execution of an operator. State transition primarily depends on the scheduling policy in use.
4. End Query: An operator is transitioned into the stop state when end query condition is reached.

Depending on these parameters, an operator can be in one of the above-mentioned states of execution. Figure 4.8 shows the state transition diagram.

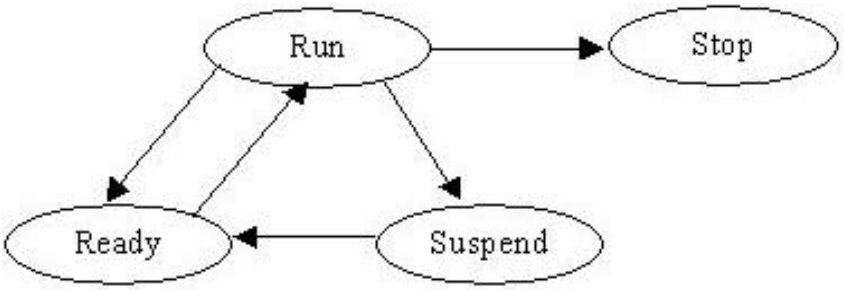

Figure 4.8 Operator State Transition Diagram

1. Ready to Run: This state says that operator has enough input tuples to carry on its execution. It is waiting only for CPU resources. This state tells the scheduler that the operator is ready to run and is in contention for scheduling. Initially when the operator is instantiated, it starts in this state. During execution of an operator, if tuples are available and the time quantum assigned to the operator is expired, the scheduler brings the operator back to this state.
2. Running: Based on the scheduling scheme, an operator will be chosen for execution if it is in ready to run state.
3. Suspended: If tuples are not available to the operator, it will suspend itself and informs the buffer of its suspension. Later when tuples are available in the input buffer, it will either bring the operator in ready to run state or running state based on the scheduling policy. For flow-based scheduling it will be brought directly to running state and for all other policies it will be transitioned to ready to run state.

4. Stop: An operator is transitioned in the stop state when the operator processes all the tuples and the end query condition is reached.

Buffers, Scheduler, Instantiator and the Operator itself are responsible for the state transition.

4.3.4 Operator Priority:

Controlling the execution of operators can easily influence the overall throughput of the system. Hence there is a need to provide a control over the execution of an operator. Priority plays a pivotal role in providing this control. Each operator has a priority that ranges from 0-9, the default being 5.

Priority can be set in one of the following manners:

- Default Priority: If user does not have any specific QoS requirements, then the query Instantiator will instantiated the operator with a default priority of five.
- User Specified: A user is allowed to specify certain level of QoS by specifying the priority level for a query. When that query is submitted for instantiation, all operators of the query are set according to the priority of the query.
- System Generated: A priority can also be assigned/changed during the execution of an operator. If one operator is being used by more than one query or if it is one of the slow moving operators, then to increase the efficiency of the system as a whole, the optimizer can change the priority of the operator. Again after monitoring the QoS output for the query, if the optimizer realizes the degradation in the query output quality, then it might change the priorities of some operators in the system such that the desired QoS for the query is obtained.

4.3.5 Operator Types:

Basically there are two types of operators depending how they treat the incoming tuples.

- Non-Window Based Operators: Non-Window Based Operators can process one tuple at a time. It takes a tuple from the input buffers, processes it and then generates the output into the output buffers. These operators do not have to wait for a set of tuples to begin their execution. These are naturally non-blocking in nature. SELECT and PROJECT are examples of non-windowed operators.
- Window Based Operators: Some operators like JOIN and AGGREGATE requires a set of tuples in order to begin their execution. In traditional DBMS's these operators are blocking in nature. They have to wait for the entire set of data to be available before delivering the output to the user. However, the nature of streaming application and the requirement of streaming operators demands the operator to be non-blocking. Hence, a solution generally accepted for streaming data operators is to work on windows. A computational window defines a set of tuples based either on timestamp or the number of tuples on which the operator needs to execute. This way the operator does not block for the entire set of tuples before it starts to deliver the output. Output is completely delivered at the end of each window.

4.3.6 Query Windows:

Windows are classified as physical or logical based [14] on how windows are defined.

- Physical Window: Certain applications demand processing of tuples based on the actual life time, in terms of hours, minutes, days, months, etc. Hence the windows are distinguished based on physical time stamp. For example, *Give me the daily count of the number of students entering for the month of January 2003.* Here the window length is of one day and the query's lifetime spans over a period of one month.
- Logical Window: Windows can also be classified in terms on number of tuples. Each window has a certain (user-defined) number of tuples on which the user needs the query output. For example, *For every 1000 transactions generated from MavHome, find out the count for the transactions coming from Room A.* Here the window is of 1000 tuples and the condition to be checked is for Room A and the output is aggregate using the count functions. This is a continuously running query as there is no end query condition.

Both Physical and Logical windows can be further classified depending on how the window is moving to get the next window bounds. The classification is as follows:

Snapshot Window: Snapshot is a single fixed window. Here the windows start and end points are fixed. These types of windows usually occur in one-time queries. On generating the output for that window, the query is removed from the system. For example, Count *the number of people entered the lab from 9:00 am – 5:00 pm on Dec 13 2003.*

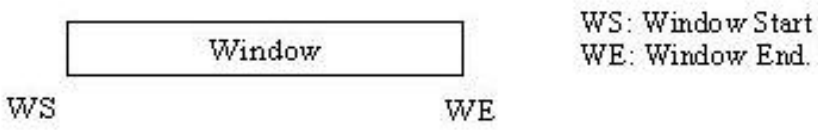

Figure 4.9 Snapshot Window

Landmark Window: Landmark window has a fixed starting point and the end point is moving. If the end point moves in the forward direction, it is a forward landmark window. On the other hand if the end point moves in the reverse direction, it is a reverse landmark window. Reverse landmark refers to historical data or data that has already been processed. Currently this system does not support queries that process historical data. Hence reverse landmark window is no currently supported.

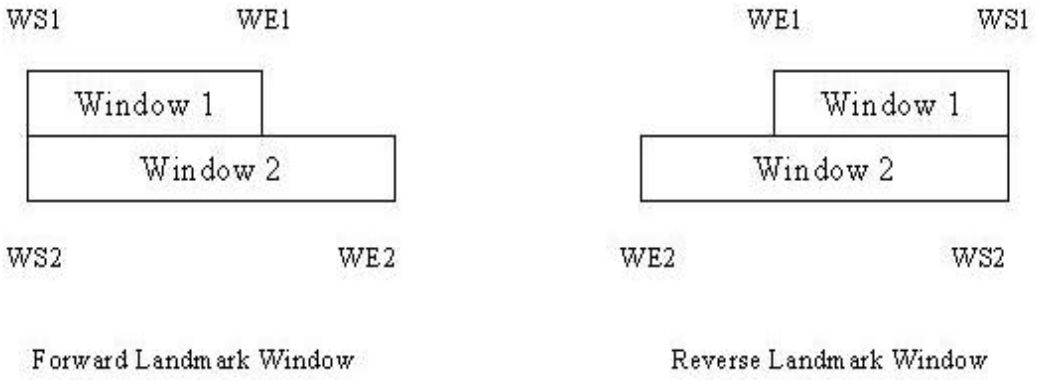

Figure 4.10 Forward and Reverse Landmark Window

Figure 4.10 shows the forward and reverse landmark windows. The difference between the two can be seen from the direction in which the window is moving. For forward window, it refers to the current or the future tuples where as it refers to the past tuples in reverse landmark. An example of forward landmark query is: *List the usage of the devices in Room A from 9:00 am – 5:00 pm on Dec 03 2003 every hour*. In this query the initial window size is of one hour and for every hour the size of window is

increasing by an hour but in the forward direction. The last window is of the size 8 hours.

Sliding Window: Sliding window has both its ends moving in the same direction and by the same amount. Sliding window can be a forward sliding window or reverse sliding window based on the direction in which the window is moving. Again reverse sliding window refers to past data and forward window refers to current or future data. Sliding window can be further classified based on the starting point of the next window.

Overlap Sliding Window: Here the starting point of the next window is less than the ending point of the current window. In this case, there are some tuples in the window that are overlapping in both the windows. Overlapping can be in forward or reverse direction as shown in figure 4.11. An example of forward sliding window query is: *Give a count of the number of devices turned off over one hour interval between 10 am – 11 am on Dec 15 2003 every 10 minutes.* Here the window length is of one hour. The first window is starting at 10 am – 11 am and the next window is between 10:10 am – 11: 10 am. Hence there is an overlap of 50 minutes between the two windows.

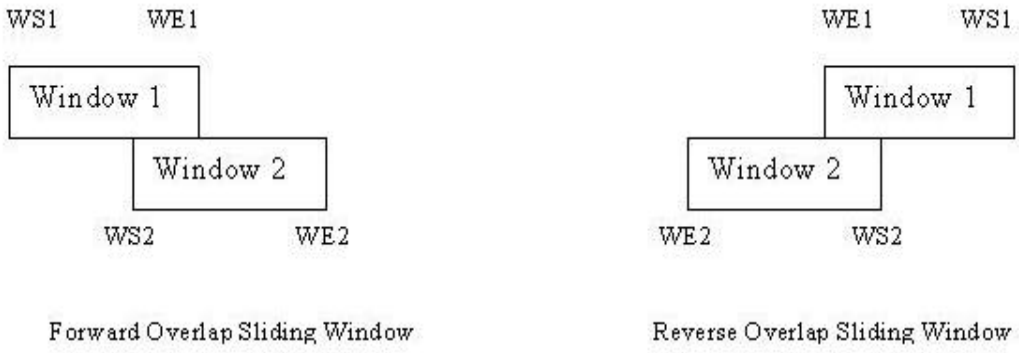

Figure 4.11 Overlap Sliding Windows

Disjoint Sliding Window: Here there is no overlap of elements between the two adjacent windows. They are completely disjoint in nature. Starting point of the next window is greater than or equal to the ending point of the current window. As shown in the figure below, disjoint sliding windows can be forward or reverse sliding windows based on the direction in which the window is moving. An example for disjoint forward sliding window is: *Display the average temperature of Room A every hour from 10pm to 10 am on Dec 15 2003.* The first window length of one hour is between 10pm-11pm, next between 11-12pm and so on.

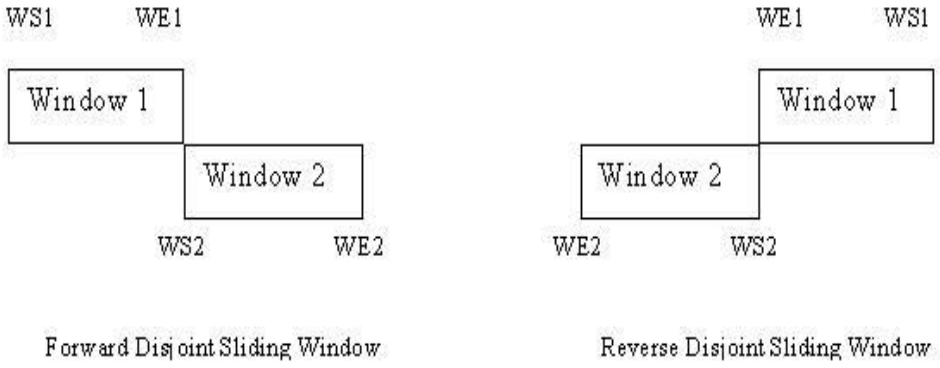

Figure 4.12 Disjoint Sliding Windows.

4.3.6.1 Window Representation:

In order to represent stream-based queries, the representation should clearly handle all of the above window specifications. Normal SQL-based query language does not have the provision to support window-based operations. Hence an extension is needed in order to support window-based queries. Also, this representation should support both physical and logical windows. Furthermore, it should have the capability to support all kinds of moving window queries.

The following four clauses are proposed for a generic window representation.

1. Begin Window: Begin Window defines the starting point for the first window.
2. End Window: End Window defines the ending point for the first window.
3. Hop Size (lower bound, upper bound): Hop Size represents the amount by which the window will be moving in either direction. It is the hop size that determines the type of query window.
4. End Query: End Query defines the end query condition.

For physical window representation all of the above clauses are specified in terms of physical time stamp and for logical queries, these are specified in terms of number or tuples. For example, *Display the average temperature of Room A every hour from 10 am to 1o pm on Dec 15 2003 is represented as*

Begin Window = 10 am on Dec 15 2003.

End Window = 11 am on Dec 15 2003.

Hop Size = (1hr, 1hr).

End Query = 10 pm on Dec 15 2003.

Or Display the count for the transaction generated for Room A for every 100 transaction is represented as

Begin Window = 1st tuples

End Window = 100^{th} tuple

Hop Size = (100,100)

End Query = infinity.

We can control the different types of window movement by varying the hop size. Below given are the hop movements for different types of window.

Forward Only Window: Both the hop bound needs to be positive for this type of window.

Reverse Only Window: Negative values in both the hop bound gives a reverse window.

Landmark window: Here the lower bound is fixed and set to zero and the upper bound is a variable. A positive value of upper bound gives a forward only landmark query whereas negative value gives a reverse landmark window.

Sliding Window: Here both the lower and upper bound are specified. If both the values are positive it is a forward only sliding window whereas negative values move the window in the reverse direction.

4.3.7 Generic Operator:

Based on the requirements stated for the stream operators, it is evident that there are certain properties, data structures and methods common among all operators. Each operator is a schedulable entity. There is a lot of interaction between operators and various components of the stream database. Buffers, Scheduler as well as the optimizer need to communicate with the operators. Hence, it is necessary to present a common view for all operators rather then viewing them as SELECT, PROJECT and JOIN. If all other components view all operators as single Object Operator then the process of implementing this communication can be simplified. Moreover, new operators can be added in the future without changing other modules. All these requirements lead to combining the common attributes and functionality of operators into a common

(generic) operator. Actual operators such as SELECT, PROJECT and JOIN are inherited from the common operator.

The following structures/methods are included in the common operators.

- Priority.
- State.
- All operators can output tuples to one or more output queues. Hence the output queues can be made a part of the generic operator. Input queues cannot be included in the generic operator as there is only one case that includes two input queues. Only the join operator has two input queues and so it will be more logical to include the input queues in the actual operators.

Generic operator should include methods that are useful to control the execution of the operator like start, stop, suspend, resume etc.

4.3.8 SELECT:

4.3.8.1 Requirements:

SELECT operator is similar to the FILTER (SELECT) operator of DBMS. The primary function of the select operator is to filter a given input stream based on the specified condition. It has one input queue, and can have more than one output queue. Before starting the operation, it assumes that the condition to be evaluated has been set by the query Instantiator based on user input. SELECT takes in a tuple from the input queue and checks whether it satisfies the given condition. If the condition evaluation is successful then the tuple is output to all the output queues associated with SELECT. If

condition evaluation fails then the tuple is discarded and the next tuple in the queue is processed. SELECT checks for the endQuery condition and stops its execution if the current tuple has passed the end query condition.

4.3.8.2 Design:

One of the major components of SELECT evaluation is the condition evaluator. The major issues while designing SELECT was whether to build a condition evaluator or to use some of the existing evaluators in the market. A lot of effort is required to build a flawless and efficient condition evaluator that can take in any valid condition and return whether the output is true or false. Hence a decision was made to use the existing condition evaluators available for free. FESI (Free Ecma Script Evaluator which is a powerful utility that can perform many functions in Java, and condition evaluation is one of them), a java-based tool that provides the functionality of condition evaluation was selected for the purpose of condition evaluation.

FESI's condition evaluator works in the following way.

- It takes in a condition that needs to be evaluated.
- It also requires the operand values in order to evaluate the condition on the tuples.
- It returns true or false on evaluating the condition.

Another design issue was to bring the condition to a form that can be evaluated by the condition evaluator. One of the initial requirements was to make sure that the condition string was not processed each time a tuple is evaluated. To achieve this operands are separated out from the condition string and their position is obtained from

the schema. Using the operand position information, the mapping of tuple value and its operands was done. Eventually, the condition evaluation is made simple and efficient.

A common design issue with all operators was to provide a support for flow-based scheduling. If no tuples are available from the input buffer, the operation of the operator needs to be suspended. One way to support this was to ask the buffer to suspend an operator if no tuples are available in the buffer. However, this would have put unnecessary burden on the buffer and the operation would also have been time consuming, as there needs to be communication between buffers and the operator. To avoid this, the logic was incorporated in the operator itself. If no tuples were available to be read from the input buffer, the operator would suspend itself. Before suspending itself, it would inform the input buffer of its suspension. This way buffer knows that the operator got suspended, as there were not tuples available for the operator to process. In turn, buffer has logic to resume all suspended operators as soon as a new tuple is enqueued into the buffer. In this way flow based scheduling is supported, which improves the overall efficiency of the system.

```
while (query is alive) {
        if (tuples are available in input queue){
                read tuple from input queue.
                replace attributes of the condition string by the corresponding tuple
                field values to generate modified condition string.
                evaluate the modified condition string using Fesi evaluator.
                if (condition is satisfied)
                        output tuple to its output queue.
        else{
                Informs the input buffer of its suspension.
                wait on the input queue (operator suspended).
        }
}
```

Figure 4.13 SELECT Algorithm

Algorithm for SELECT evaluation is given above. As mentioned in the requirement, it obtains a tuple from the input buffer and set the operand values to the condition evaluator and evaluates the condition. If the condition evaluates to true, the tuple is inserted into the output queue.

If no tuples are available in the input buffer, SELECT informs the input buffer of this condition and suspends itself.

4.3.9 PROJECT:

4.3.9.1 Requirements:

Functionally, PROJECT is analogous to the PROJECT operation of a traditional DBMS. It takes in a tuple and discards the specified fields in the form a new tuple. It has one input queue and may have one or more output queues. One of the inputs to project is the position of list of attributes that needs to be projected out from the tuple.

4.3.9.2 Design:

Project operator requires a list of field operands that needs to be projected. The actual operation is done using the position of operands in the tuples. User submits a list of operands that needs to be projected. This list can be converted to a position list by the query Instantiator or in the Project itself. Either of the two would have given correct output without affecting the performance. In our design it was decided to convert the operand list to position list at query Instantiator and pass only the position list to the project.

Support for flow based scheduling is same as discussed in the SELECT operator.

An important property of the PROJECT operator is that it is a contracting operator. This means that the input buffer schema will be contracted to form a new schema. Hence, the output of PROJECT produces a new schema. This schema information is input from the DSMS client during query instantiation.

Algorithm for PROJECT is given below. Here the assumption is that the position list is already set before starting the project operation. If the tuples are available in the buffer, it pulls up one tuple and creates a new tuple by outputting the fields

specified in the position list. This new tuple is output to all the output queues associated with PROJECT. If no tuples are available, then PROJECT suspends itself but before that it informs the input buffer of its suspension. On receiving a tuple in the buffer, the buffer will resume the PROJECT operator.

The algorithm for *Project is as f*ollows:

```
while(query is alive ){
        if (tuples are available in input queue){
                read tuple from input queue.
                create a new tuple with projected attributes only.
                send new tuple at the output queue.
        }
        else{
                Inform the input buffer of its suspension.
                wait on the input queue (operator suspended)
        }
}
```

Figure 4.14 PROJECT Algorithm

4.3.10 Hash Join

The processing requirement of a join operator is entirely different from non-windowed operator discussed so far that works on single tuple at a time. Join, on the other hand, is considered a blocking operator that operates on multiple tuples on each of its input queues. An operator is said to be blocking if it cannot produce its first output

until it has the entire input set is available. In a conventional DBMS where inputs are clearly defined in terms of finite relations, computation does not involve much complexity. However, streams being indefinite and potentially unbounded in size, forces re-examination and modification of the design of traditional Join algorithm to operate on infinite streams. The design requirements explained below would justify the additional functionality, which needs to be supported for providing real time response to streamed queries.

4.3.10.1 Design Requirements:

- Non-blocking operation
- Incremental and Continuous output
- Timestamp Ordering
- Duplicate Avoidance

Non-blocking operation: Data streams are not stored on disk or memory rather they arrive online asynchronously. This nature of incoming stream makes the processing more difficult as the input bound is not clearly defined. In order to precisely define input boundaries for blocking operators, the concept of window is introduced. A "Join" operator considers all tuples falling in the window within its input bound. Once all tuples within the current window are processed, windows are moved to consider next set of continuous data. Also, there is a window class that provides functionality for creating and manipulating windows. Furthermore, it can provide window definition to any number of windowed operators. Moreover, simultaneous and/or independent movement of windows for each operator can also be controlled.

Incremental and Continuous output: In the conventional Join operator, all elements from one relation are hashed prior to first tuple processing and the elements from other relations are then used to probe the already hashed tuples. Once all the computations are made, the entire result is produced at the output. This algorithm fails in processing continuous streams since it is blocking and does not produce continuous and incremental output. The stream operator in contrast produces output at regular intervals, not necessarily at the end of window. This incremental output increases efficiency of the system as the higher operators in a query tree whether windowed or non-windowed are not blocked and the end users are also kept updated of the results.

Timestamp ordering: A window-based operation requires its input as well as output to be timestamp ordered. Timestamp ordering allows this operator to detect window termination. "Join" reads every tuple and compares its timestamp ('x') with the timestamp associated with current end window ('y'). If 'x' < ='y', the tuple is processed since it falls within the current window. If 'y' > 'x', it detects window termination since all tuples beyond the current tuples are guaranteed to fall outside the current window as they are assumed to be in the increasing order of timestamp. Output tuples must also be timestamp ordered since higher operators in a query tree can be blocking operators. It is the responsibility of the operator to produce results not only in timestamp ordered but also incrementally and continuously to avoid blocking of higher operators.

Duplicate avoidance: The "Join" operator has two input queues that are populated either by data streams or its children in a query tree. It has two hash tables, one corresponding to each of its input queue. Tuples are consumed for processing as soon as they enter the

input. These contrasts with the design of conventional DBMS join operators that do not start computation until entire input set is available. The join operation includes the task of reading one tuple at a time from its input queue, hashing it in its corresponding hash table based on the value of the joining attribute and probing the other hash table to find the tuples that fall in the desired window bound and satisfying the join condition. If two tuples named ‘x’ and ‘y’ are considered for processing at the same time then the output would be two ‘xy’ if they satisfy the joining criteria. In order to avoid duplicates in the result, every tuple is processed atomically. An atomic action ensures that the next tuple is not considered until current tuple is completely processed.

4.3.10.2 Design Alternatives:

Two threads instead of single thread: Initially this operator was designed using two threads that had inherent problem of duplicates, as the two threads were running independently and asynchronously. Left thread was consuming tuples from its left external buffers, hashing the tuple in its corresponding left hash table and probing the same tuple against the tuples present in the right hash table. The matched tuples were joined and produced at the output. The operators maintain these hash tables internally. Right thread works analogously on its corresponding external buffer and internal hash table simultaneously. The simultaneous processing of tuples increases parallelism and response time but does not guarantee correctness, as some of tuples in the output can be duplicated. Duplicates are produced when both the threads read tuples at the same time from their corresponding external buffer that satisfy join condition. Then the join operation would be carried out twice, once by the left thread followed by the right

thread generating duplicates. To overcome this problem, two threads were synchronized to allow atomic processing of tuples. Synchronization completely defeated the purpose of parallelism, as only one thread could be active at a time. Since correctness of the algorithm is more important than efficiency, this algorithm is implemented using a single thread.

Avoid processing of tuples in timestamp ordering: Tuples are processed in order of their associated timestamp. This is essential to produce output in timestamp order. Assume two tuples 'x' and 'y' read from left and right external buffer respectively. If 'x' < 'y', 'x' will be processed first else 'y' is processed. When the tuple being processed is joined with tuples in the internal hash table, resultant tuple would have timestamp m = max (x, y). This algorithm ensures that all output is automatically sorted on 'm'. If this timestamp ordering is not respected while reading from external buffers, tuples produced at the output may not be always sorted on timestamp and the higher operators would produce incorrect results. The only alternative to produce the correct result is to sort the output before delivering them to higher operators that comes at the expense of running a sort algorithm. A major drawback of this approach is blocking operation, as higher operators cannot see their first input until all outputs are produced and sorted by previous windowed operator. Hence this alternative was also ruled out due the problems mentioned above.

4.3.10.3 Variations in Hash Join:

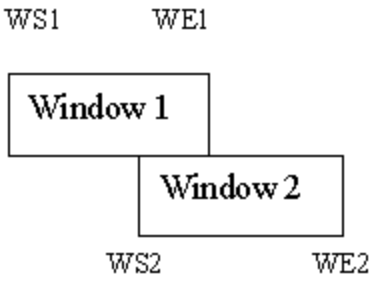

Figure 4.15 Sliding Window Overlap

There are two variations in Hash Join based on whether common computations are exploited in overlapping windows. In the case of a disjoint window, there is no overlap region and hence all windows are computed independently. In overlap windows, as seen in figure 4.15, the computation between WIS and W2E are common that can be re-used in the computation of the next window. Based on this concept, hash join are classified as:

- Hash Join with temporary storage
- Hash Join without temporary storage

Hash Join with temporary storage:

In this variation, common computation between two successive windows is exploited. There is a temporary storage involved for storing the computation of current window that is used for the computation of next window and hence the name “Hash

Join with temporary storage". The process of reusing result produced by the current window in the computation of next window is explained below.

During the processing of the first window, tuples falling in the overlapped region are identified. In figure 4.15, common computation involves processing tuples between W2S and W1E. Only those tuples that fall between W2S and W1E and satisfy the join condition are produced not only at the output but also replicated in a temporary storage. This process continues until the current window is completely processed. Prior to processing of next window, the results of temporary storage are placed in the output queue corresponding to the second window and hence the name "Join with temporary storage". This avoids re-computation of tuples in overlapped region, which can be substantial if window size and overlap are large.

Hash Join without temporary storage: This variation does not preserve common computation as each window is computed independently. There is no temporary storage involved since the results of current window are not materialized for the next window. The performance difference in these two variations increases as the window size and overlap increases. This variation can be used when memory is critical since no temporary storage is involved. However, in stream processing where response time is critical, "Join with temporary storage" can surpass this variation.

4.4 Query Instantiator

Query Instantiator takes in a query as input from the user and brings it into an executable form. It is also responsible for starting the process of query execution.

4.4.1 Requirements of query Instantiator:

- Design should clearly specify how query information should be submitted to stream database.
- It should specify what all data should be accompanied as part of user input for the query.
- It should indicate how would the server respond to the client when a request for query instantiation is submitted?
- It should support flow-based scheduling.
- It should instantiate operators from the information provided as part of query input.
- Appropriate configuration parameters must be used while instantiating operators and buffers.
- Identify operators that require stream as their input and associate appropriate stream buffers for the operators.
- Create new buffers to store the output of operators.
- Create a passage for flow of tuples from stream buffers to the root operator and then to the application/user by connecting proper buffers between operators.

4.4.2 Design of Query Instantiator:

This section describes various design issues, challenges and options considered while designing an Instantiator.

Query User Input: Query input defines the form in which query related data is submitted to the server for instantiation. There are two ways of submitting the query data.

String based Input: One option is to define an SQL like interface to represent a query. For example, Display *the average temperature of Room A every hour from 10 am to 10 pm on Dec 15 2003.* To represent this query the corresponding SQL like interface would be

SELECT avg (temp) FROM roomStream

WHERE BEGIN WINDOW = 10 am on Dec 15 2003

END WINDOW = 11 am on Dec 15 2003

HOP SIZE = (1hr, 1hr)

END QUERY = 10 pm on Dec 15 2003

Validation logic needs to be implemented at the client side, which takes in the query and validates the syntax of the query. The validation logic also needs to check whether it refers to a correct stream or not and the fields specified in the query are as per the description of the schema of the stream specified in the query. To validate this the entire string had to be parsed once at the client side. Once it is submitted to the server, it needs to be again parsed and an initial plan needs to be created. From the plan the operators needs to be instantiated. This holds good for other commands such as create schema, view schema etc.

Object based input: Here the user is presented with a GUI based interface where in the user is allowed to submit an initial plan of the query. GUI provides a set of schemas that are already defined on the server and are being fed with data. User selects the schemas

on which the query will be based. Then the interface provides a list of operators from which the user can select. Once the operator is selected, user is asked for a specific list of parameters for that object. For example, for the SELECT operator, user will be asked to enter the filter condition on the stream and for join user needs to specify the joining attribute and the join condition. For each operator user specifies the input source from which the operator would read the data. The input source could be a stream or it could be output from other operators. If a new schema is generated at the output of an operator then the new schema is sent to the server for creation. A new schema is created for operators such as join or project where the base input schema is either expanded in the case of join or contracted in the case of project. Thus a new schema definition is created, which higher operators in the tree can use. This way user specifies the entire query tree. The client interface then creates a query tree consisting of operator data nodes and is sent to the server for instantiation. For example, Display the common device usage between rooms A and B in the MavHome lab. The corresponding query tree for this query would be

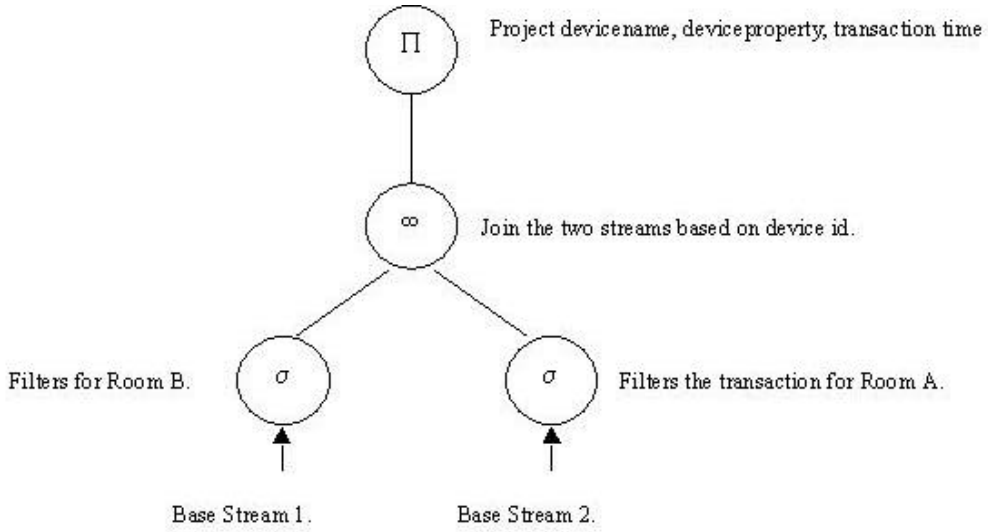

Figure 4.16 Query Tree of Operator Data Node

The query tree shown in figure 4.16 is a tree of operator data node. This tree is built by the client interface based on the input provided by the user. Now the question is how to represent the operator data node.

4.4.2.1 Operator Data Node:

The following are the requirements of the Operator Data Node.

- There should be one representation that can hold sufficient data to instantiate any of the operators supported by the system.
- It should clearly specify the type of operator it is representing.
- It should specify the parameter for the operator that it represents. These parameters help the query Instantiator to set the properties of the operators.
- It should specify the output stream name. This represents the output tuple that are generated from the operators. It is especially useful for operators that

generate an intermediate schema at their output. Also it informs the Instantiator the input source and schema for the next operator in the tree.

- It should specify the input stream on which the operator will process.
- It should specify the window parameters required for window-based operators like Join.

Based on the above requirements, a generic representation of Operator Data Node was designed. The data node had the following elements.

- Operator Type: This specifies the type of operator like SELECT, PROJECT, JOIN etc.
- Stream One: A schema represents the buffer from which tuples are fed to the operator. Stream One specifies this schema. For operators like SELECT, PROJECT and AGGREGATE only one input stream is required.
- Stream Two: For operators like JOIN, which executes on two input buffers, Stream Two provides the schema definition on the other buffer. This parameter is not defined for operators like SELECT, PROJECT and AGGREGATE.
- Input Parameters: This specifies the input parameters for the operator in use. The value of input parameters varies from operator to operator. For SELECT it specifies the condition on which the filter operator is carried out on the input buffer. For JOIN it specifies the joining attribute on the two streams. For PROJECT it specifies the list of fields that needs to be output as part of project operation. For example, consider the query *Display the common device usage*

between rooms A and B in MavHome lab. The input parameter for the operators in this query would be:

SELECT1: ROOMID = A

SELECT2: ROOMID = B

JOIN = DEVICEID, DEVICEID

PROJECT = DEVICEID, ROOMID, TRANSDATETIME.

- Window Parameters: This parameter becomes useful for operators that are window based. It specifies all the clauses required to represent a window. The following are the window parameters that are included.
- Begin Window: Specifies the start of the window.
- End Window: Specifies the end of first window.
- Hop Size: Specifies the bound that needs to be added to begin and end window to obtain the next window.
- End query: Specifies the time at which the query will be terminated.

Figure 4.17 shows an example representation for the Operator Node. This node represents the SELECT operator in the query specified in Figure 4.16.

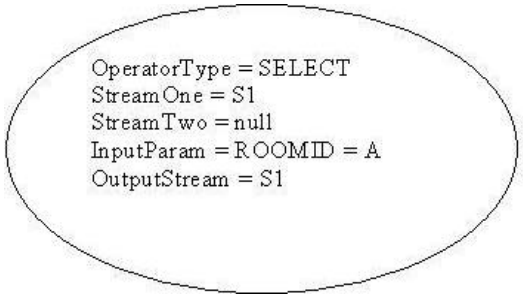

Figure 4.17 Example of an Operator Data Node

4.4.3 Instantiator Support for Flow Based Scheduling:

In flow-based scheduling operators are scheduled on the availability of data. If data is available, the operator is scheduled else its operation is suspended. In order to support flow based scheduling, Instantiator should make sure that the operators that are likely to receive data first are instantiated first. If an operator is receiving data from another operator then it should be instantiated after the tuple-producing operator is instantiated. To meet the requirements of flow-based scheduling, operators are instantiated in a bottom up fashion. In this scheme the leaf operators that are reading data from the stream are instantiated first. The operators in the tree till the root operator follow these. Root operator is the last one to be instantiated as it is last operator in the chain to receive the tuples inputted from the base stream.

Once the operators are instantiated, based on the scheduling scheme used, one of the following actions is taken.

For a flow based scheduling scheme, operators are started as soon as they are instantiated.

For Other scheduling schemes, operators are added to the ready queue of the scheduler. Later the operators are scheduled based on the scheduling scheme. Still the operators are added to the queue based on the sequence in which the data is flowing.

4.4.4 Base Stream and Buffer List:

Each stream feeds its tuples into a buffer. There can be more than one stream in the system each feeding tuple into a different buffer. There is a need to maintain information about the association between the streams and its buffers. This is done with

the help of StreamBuffer list. This list has two values, one is the stream name and the other is the associate buffer.

Stream Name	Buffer Instance
S1	B1
S2	B2
⋮	⋮
Sn	Bn

Figure 4.18 Stream Buffer List

Whenever a new stream is registered with the system, a buffer is associated with the stream. This information is stored into the Stream Buffer List, which is useful when associating stream buffers with operators.

4.4.5 Operator Instantiation and buffer mapping:

In order to instantiate operators, the query tree is traversed in a bottom up fashion. It picks up an operator data node, identifies the operator type and based on its type instantiates the operator. There are three aspects to operator instantiation.

Creating the Operator Instance: Each operator data node has a field called OperatorType. This specifies the type of operator that the operator data node represents. Based on the operator data type, appropriate operator is instantiated. For example, if Operator Type is SELECT then an instance of SELECT operator is created.

Associating Input Buffers: With this operation appropriate input buffers are associated with instantiated operators. Based on the position of the operator in the tree one of the two types of operators can be associated in the tree.

Stream Buffers: Stream buffers are associated with leaf operators in the query tree. StreamOne and StreamTwo data of operator data nodes are used for associating appropriate stream buffers to the operator. Instantiator reads these values from the data node and read the corresponding buffer information from the StreamBufferList. The obtained buffer is then associated as an input buffer to the operator.

Operator Output Buffer: For non-leaf operators the output buffer of previous operator in the list becomes the input buffer of the current operator in the tree.

Creating New Output Buffers: For each operator a new output buffer is created and is associated with that operator in the tree. Instantiator also stores the information of the newly created output buffer in some temporary storage so that when the next operator in the tree is instantiated, this operators acts as an input buffer to the newly instantiated operator.

Input Parameters: Each operator wants the input parameters in a specified format. Some times the input supplied as part of operator data is good enough to pass it to the operator, but some of the operators may require modifying the input parameters to bring in to a form acceptable by the operator. Here is how the input parameters are supplied to each of the following operators.

SELECT: The condition string read from the operator data node for SELECT operator is passed as it is to the instantiated SELECT operator.

JOIN: Operator data node supplies the joining attribute as an input parameter. Since the join is a binary join, only two attributes are supplied one for left input stream and other for right input stream. These values are comma separated. But this form is not

acceptable by the JOIN operator. What join operator requires is the position of the joining attribute in schema for each of the streams. Hence it the mapping of attribute name to its position in the schema and supplying it as an input become the responsibility of the Instantiator. Instantiator does a look up in the schema using the attribute name and finds out its position and sets the appropriate position in the Join Operator.

PROJECT: For project operator, operator data node supplies the list of attribute names as part of input parameters. But project operator accepts a list of position of attributes in the tuples that needs to be projected. Hence the Instantiator finds out the position of each of the given attribute name using the schema corresponding to the StreamOne in data node. It then provides the position list as part of the input to the project operator.

Window Information: SELECT and PROJECT are interested in knowing the END QUERY information of the Window. JOIN is a window-based operator and hence it requires knowing all of the window information. Instantiator sets the required information for each of the operators.

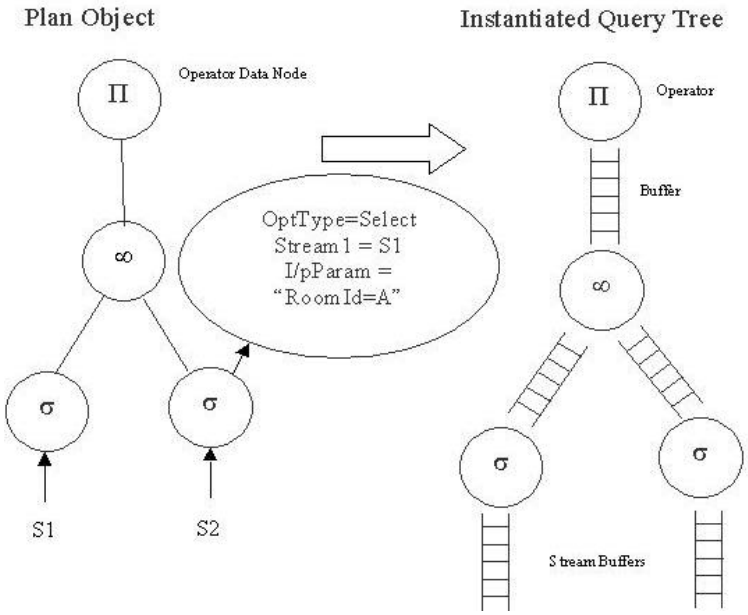

Figure 4.19 Process of Instantiation

Figure 4.19 shows the process of instantiation. Instantiator takes in a plan object, which is a tree of Operator Data Node. It traverses the tree in a bottom up fashion, instantiates the each operators, based on the position of the operator in the tree, associates appropriate buffers to the operators and schedules them for execution. As shown in figure 4.19, a query tree of operator node is converted into an instantiated query tree of actual operators connected through buffers.

4.5 DSMS SERVER DESIGN

4.5.1 Communication Model

DSMS is implemented as a client-server architecture. Communication between the client and the server needs to be command driven and protocol oriented. A predefined set of commands that the client can invoke is available. Based on the users

requirements, client sends out a command to the server to execute a particular operation. Each command has a well-defined protocol that defines the communication between the client and the server. The protocol clearly specifies what data needs to be sent to the server, what operations are going to be carried out at the server and what the client should expect back from the server. Based on these requirements of communication, a socket-based server is proposed. The other choice available and considered was of Remote Method Invocation based server. This choice was ruled out, as the idea of distributed server is not considered. This system is analogous to any other commercial DBMS available in the market wherein the protocol for client server communication is well defined and any client can be used provided it follows the said protocol.

Server implements a set of commands, which provides primitive control over user-generated schema and query. More commands can be added as required. To invoke each of the above given commands, user first sends out a command identifier. For simplicity an integer command identifier is chosen. This helps the server to distinguish the command. After the command, comes the protocol which defines the input and output operations between the client and the server. Following is the description of the operations of the command supported and its protocol.

New Query: Using this command, user is able to submit a new query to the system. User submits the query in terms of an initial plan. Plan is nothing but a query tree consisting of operator data nodes. Each operator data node is a generic object, which encompasses the necessary information to instantiate the corresponding operator. This

plan is submitted to the *Instantiator* which instantiates each operator in the plan and connects them using buffers. Once the given query is instantiated, a query handle is given to the user. Server also updates the query data structures, which provides a repository for storing statistical information for each query that has been instantiated during the life span of the server. Using this handle, user can further control the query (e.g. stopping or modifying the query).

Stop Query: User should be able to stop a long running query as desired. DSMS expects the client to send the handle (or ID) of the query that needs to be stopped. From the Query Data Structure, the server picks up the operators running for that query and stops each of the running operators. System, then updates the state of the query in the query data structure to be terminated.

New Stream: With this command user is able to create a new schema with the server. Client takes the schema input from the user and converts it into a form given in the schema definition. This is what is sent as an input for the command. Server adds this schema to the DSMS schema table. It acknowledges to the client whether the given schema was successfully added or not. Proper error message is communicated to the user if the schema was not added successfully for e.g. client sends a schema whose name already exists in the system. For this case, server will return back to the client informing about the duplicate schema.

Get All Streams: This command enables the user to pull the entire schema information from the server. Upon receiving this command, server sends the entire schema table (as described in the Schema definition) to the user.

Get a Stream: If user wants details about a particular schema, this command can be used. First, user sends the name of the schema whose information is required. Later, server follows by sending the schema information as defined in the Schema definition to the user.

4.5.2 DSMS Client:

DSMS Client is responsible for taking request from user and converting them into a form required by the server. Client should be fully aware of the commands being supported and the protocol that needs to follow for those commands. Client communicates with the server over the agreed socket number where the server is listening. Based on the user request it sends the command identifier. For example, for a new query it would send an identifier "1" and then would covert the users request in to plan tree of operator data node and then sends the plan object to the Server and waits for a query handle to be returned. Client can be one of the following types.

- GUI Based Client: A GUI based client can be constructed in the language in which DSMS is built. For example, for a Java based DSMS the client could be built using Swing or AWT. The GUI should allow some of the basic operations user wants to perform like running a query, stopping a query, creating a schema, deleting a schema etc.
- Non GUI Based Client: This type of client can be used for testing purpose where there is no GUI for the user. Hence the commands have to be hard coded in the program or can be read from an external storage as a file.

- Web Based Client: Here a web-based interface is provided to the user. Actual client side functionality resides at the web server. For this DSMS a web based client is proposed. Worldwide accessibility is the primary advantage of this method.

4.5.3 Server Functionality

Server as a whole is responsible for invoking and maintaining various components shown in the architecture of DSMS. These include the Scheduler, Operators, Buffer, Plan Generator and Run Time Optimizer. For effective operations of DSMS as a whole, server implements the following functionality.

- Server maintains the schema for DSMS. Schema manipulation (add, modify and deletion of schema) is done based on users request.
- Associates buffers for each schema where the data of each schema will be streamed. It also provides a smooth reading of each stream into their appropriate buffer in a separate thread.
- Provides a command-based interface where in the client can invoke command based on users request.
- Invokes various components such as scheduler, buffer, run-time optimizer, etc and initializes different parameters of each components which are necessary for their normal operation.
- Instantiates a query plan.

Components of server (Scheduler, Buffer, Operators, etc) need to be initialized based on the user input or configuration parameters set in the Configuration file.

Initially, server reads all the configuration parameters from the DSMS Server Configuration file and keeps them in memory. These parameters are then used throughout the life span of server and are also used by various components like Scheduler, Buffer, and Operators etc.

Scheduler is invoked as a separate thread. One of the important components of the scheduler is the ready queue, which is a list of operators that are ready and waiting to be scheduled. As part of new query instantiation, server initially populates the ready queue for scheduler. Based on the policy specified in the Configuration parameter, appropriate scheduling policy is chosen for operator scheduling. This is useful during experimentation early on. Later this can be replaced by some logic which will help us to select the scheduling policy depending on the type of queries, load of the system, system configuration etc.

Buffers can be bounded or unbounded and the size of buffers can be read from the configuration parameters. Later this can be replaced by a logic, which decides these parameters for buffers based on the query, its priority, current memory available and so on.

Run Time Optimizer can be implemented as a separate thread, which constantly monitors the QoS of the query and changes the overall plan to satisfy as many users as possible.

Plan generator can be a separate entity, which provides methods that take in a plan and provides set of plans that can be used in place of the original plan without changing the overall output of the system. Plan Generator can be used to initially

produce an optimized plan or can be used by Run Time Optimizer to improved the overall quality of the system.

Chapter 5

IMPLEMENTATION

5.1 Schema Implementation

As explained in the design, the following four components needs to be implemented:

- Schema Information List.
- Attribute Name Structure.
- Attribute Position Structure.
- Attribute Information.

The core information regarding a schema is stored in the Attribute Information structure. This structure needs to be a dynamically growing structure as there could be more information stored in the future. Hence, Attribute Information is implemented as a Vector as Vectors do not impose a limit on the number of elements they can store.

Each schema has its own Attribute Name Structure that provides an easy access to the attributes of the schema based on its name. There is no limit on the number of attributes that can be stored in a schema. To support a quick retrieval of schema information based on attribute name, this data structure is implemented as Hashtable in Java. The key of the Hashtable is the attribute name and the value is the reference to the corresponding Attribute Information Vector. This way a quickly accessible and dynamically expandable Attribute Name Structure is implemented.

The requirements of Attribute Position Structure are similar to that of Attribute Name Structure. The only difference is that the attribute information needs to be accessed based on attribute position. Hence, Attribute Position Structure is also implemented as a Java Hashtable. Here the key is the position number and the value is the reference to the Attribute Information Vector.

Schema Information List needs to store all information regarding a DSMS schema. Initially only Attribute Name Structure and Attribute Position Structure are included to provide easy access to schema attributes. But in the future additional elements can be stored. To support this requirement, Schema Information is stored in a Vector that has no limit on the number of elements it can store. Here the first element stored is the reference to Attribute Name Hashtable and the second element stored is the reference to Attribute Position Hashtable. Elements from number three onwards can be added in the future.

There are no restrictions on the number of Schema that can be supported by the system. Hence to provide a quick and easy access to schema information, Schema Information List is implemented as a Java Hashtable with the key as Schema name and the value as the Schema information Vector. In order to access information regarding an attribute in the schema, we need to provide the Schema name and the attribute name or its position. Using the schema name, we can hash into the Schema List and get the Attribute Name or Position Structure and depending on the value of attribute name or position, we can access the attribute information associated with it.

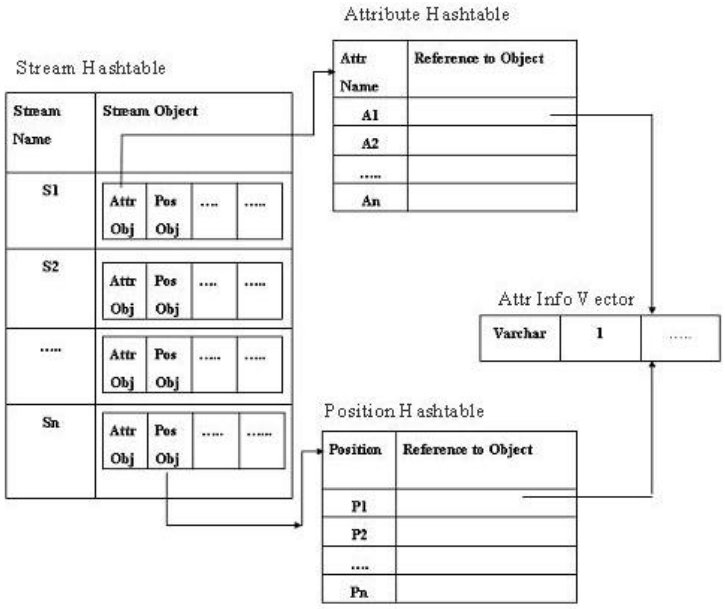

Figure 5.1 Schema Implementation

5.2 Buffer Implementation

5.2.1 Implementation of Main Memory Buffer:

Buffers are used for storing tuples of data streams. The total number of attributes and their types are user defined and hence it is not predictable. This dynamic nature of tuples in the streams can be represented with a data structure that can store any number attributes of different data type. These requirements can be easily satisfied using a Vector data structure.

Buffer is a queue of stream tuples. Here tuples flow in and out in a First In First Out (FIFO) manner. Buffers can be unlimited in size and should be able to easily provide the Enqueue and Dequeue operations of a queue. Hence Buffer is implemented

as a separate class called Buffer. This class provides all the buffer management and storage functionalities required to implement a stream buffer.

Buffer stores the tuples in Vector data structure, which is unlimited in size. For limited main memory buffers, the limit is enforced in the enqueue operations, which is explained further in the literature.

5.2.2 Implementation of Buffer Log Files

In addition to the cost of opening and closing a file, considerable overhead is incurred when reading or writing to a file. In addition, sequential access (typically used) is not appropriate as we are using the log files to bring tuples into the buffer. Concept of secondary storage buffer is different from the log files used in case of DBMS recovery. Here, the main emphasis is to minimize the file IO in writing to and reading from logs. We use an indexing mechanism for reading and writing into a log file. Java Serialization is used to persist the data values. The log files are written in append mode. All tuples that fall beyond the limit of main memory buffers are logged into the secondary storage file.

5.2.2.1 Basics of Java Object Serialization.

Java object serialization provides the ability to write or read java objects to and from a byte stream. A mechanism is provided through which Java objects and primitives can be encoded into a byte stream suitable for streaming to a network or to a file-system. The Java Serialization API provides a set of functions for developers to handle object serialization. The API is not quite large and is easy to understand and use.

A java object needs to be serialized before it can be persisted. Object is marked serializable by implementing the java.io.Serializable interface or by inheriting that implementation from its object hierarchy. Serializable interface has no methods, so the object itself need not implement any methods. Using this interface an indication is given to the Java virtual machine to use default serialization mechanism to serialize this object. ObjectInputStream and ObjectOutputStream are used for serialization and de-serialization respectively. The writeObject method of ObjectOutputStream class is responsible for saving the state of the object. On the other hand, the ObjectInputStream class provides the readObject method that is responsible for restoring the object from the serialized byte stream.

The example that follows shows the process of writing an object of class Point into a file and then reading the same object using object serialization. WritePoint class serializes the Point object to a file. ObjectOutputStream wraps around the FileOutputStream and writes the Point object to a file called Points.sav. It formats the object as a stream of bytes and saves it in the Point.sav file. On the contrary ReadPoint class de-serializes the serialized Point object. It wraps an ObjectInputStream around FileInputStream and then reads the byte stream from Point.sav to reconstruct the Point object from it. The code sample is shown in Figure

```
import java.io.*;
public class Point implements Serializable {
   public int intXCoordinate;
   private int intYCoordinate;

   public Point(int intXCoordinate, int intYCoordinate) {
      this.intXCoordinate = intXCoordinate;
      this.intYCoordinate = intYCoordinate;
   }
}
```

```
class WritePoint {
   public static void main(String [] args) {
      Point p = new Point(10, 20);
      ObjectOutputStream oos = null;
      try {
         oos = new ObjectOutputStream(
         new FileOutputStream("Point.sav"));
         oos.writeObject(p);
      }
      catch (Exception e) {
         e.printStackTrace();
      }
      finally {
         if (oos != null) {
            try {oos.flush();}
              catch (IOException ioe) {}
            try {oos.close();}
              catch (IOException ioe) {}
         }
      }
   }
}
```

```
class ReadPoint {
   public static void main(String [] args) {
      ObjectInputStream ois = null;
      try {
         ois = new ObjectInputStream(
         new FileInputStream("Point.sav"));

         Object o = ois.readObject();
      }
      catch (Exception e) {
         e.printStackTrace();
      }
      finally {
         if (ois != null) {
            try {ois.close();}
              catch (IOException ioe) {}
         }
      }
   }
}
```

Figure 5.2 Example Code for Object Serialization.

Figure 5.2 shows a very concise and easy way to implement serialization. The same code can be used to persist a complex object as the serialization mechanism works by transitive reachability. Reachability means that all the objects reachable from this object will also be serialized. If the object passed to the writeObject method contains references to other objects, the passed object and the other objects reachable from it will also be serialized. Java Object Serialization handles cyclic graphs. Each object visited is marked. If a cycle exists and an object is visited again, the mechanism knows that this

object has already been serialized. Therefore, it only puts enough information into the serialized form so that the cycle can be rebuilt when the data is de-serialized.

5.2.3 Reading and Writing into Log Files [13]:

In order to serialize an object into a file in append mode, the application needs to skip to the end of the file and then append the newly serialized object byte stream using the *writeObject ()* method of ObjectOutputStream. But the process of skipping through serialized bytes in a file is not provided in ObjectOutputStream or in FileOutputStream. Similar is the case when reading an object back from the file. The r*eadObject ()* method of ObjectInputStream class will throw a StreamCorruptedException because ObjectInputStream cannot demarcate the byte streams of objects of different type. It is not able to distinguish between the end of one object and the start of another.

To overcome these problems, a mechanism was devised in to handle the serialized byte stream. Here the serialization and de-serialization is handled at byte level and not at object level. This scheme calculates the size of the serialized object and using this information the application can easily distinguish between the objects. While serializing an object, the application should make note of the offset in the file and the size of object that is being written. This information should be used in the process of de-serialization to retrieve the correct object. Here the application will skip the file pointer to the starting point of the object and will read the number of bytes specified for the object.

Objects ByteArrayInputStream and ByteArrayOutputStream used over RandomAccessFile provides a feasible solution to the above-mentioned problem. A

RandomAccessFile is a considered as an array of bytes. It provides a File Pointer that can be used to index into the array. A log file can be read from or written into at the same file. Hence it is opened in a read-write mode. In write mode bytes are appended at the end of file and the number of bytes that are written to it advances the file pointer. In order to read an object from a file, *seek* method is used to set the FilePointer to appropriate location and then appropriate number of bytes are read for the object.

In order to find the size of serialized byte stream of an object ByteArrayOutputStream is used. Using this class the object is written into an output stream in which data is written into a byte array. It provides a size () method through which we can find the size of the object that is written into the memory. Similarly, ByteArrayInputStream is used to de-serialize the byte array read from the RandomAccessFile. The byte array obtained is passed to the ObjectInputStream that gives it a shape of an object that can be read using the readObject ().

```
logWrite(RandomAccessFile rf, Object obj,
long offset)
 byte[] barr = null;
 int objsize,
 try {
    bos = new ByteArrayOutputStream();
    so = new ObjectOutputStream(bos),
    so.writeObject(obj),
    objsize = bos.size();
    barr = new byte[objsize];
    barr = bos.toByteArray();
    rf.seek(offset),
    rf.write(barr);
    bos.close();
 } catch (Exception e) {
    System.out.println("Error in Logwrite");
    System.out.println(e),
  }
  finally {
   if (so != null) {
    try {
       so.flush();
       so.close(),
    } catch (IOException ioe)
    {ioe.printStackTrace();}
   }
  }
```

```
logRead(RandomAccessFile rf, long objsize,
long offset)
 byte[] barr = new byte[(int)objsize];
 try {
    rf.seek(offset);
    int r = rf.read(barr);
    bis = new ByteArrayInputStream(barr);
    si = new ObjectInputStream(bis),
    obj  = si.readObject();
 } catch (Exception e) {
        System.out.println(e);
   }
  finally {
   if (si != null) {
     try {
        si.close();
     } catch (IOException ioe)
     {ioe.printStackTrace();}
   }
  }
```

Figure 5.3 Reading and Writing into Log File.

Figure 5.3 shows the logRead and logWrite method that are used to read and write tuples into the secondary storage file. LogWrite method uses the technique described above to serialize the object into the bytes and append it to the end of the given file. On the other hand, LogRead skips through the file to the appropriate position and read the specified number of bytes, converts it into an object and returns it to the application.

Two Log Files:

Each buffer is associated with two log files. As explained in the design chapter it helps in purging of tuples in secondary storage. For each file the following information is maintained.

- RandomAccessFilePointer: Using this pointer all read and write operations are carried out.
- TotalFileElements: This specifies the total number of tuples that file is holding at that point of time.
- TotalReadFileElements: This specifies the total number of elements that has been read from the log file back to the main memory buffer.
- FileName: This specifies the name of the file as stored on the disk. Filename is associated with currentTimeStamp when the buffer was created. When a buffer is created two timestamps are recorded and are assigned to each of the two log files. An extension of ".log" is appended to the file name. This way we can create unique file name for all buffers throughout the lifetime of DSMS.

All this information is packaged in BufferFileData class file. Each buffer holds two objects of this class file viz. bfdForFileOne and bfdForFileTwo. These objects represent the two-log files that the buffer is supporting. Buffer needs to read tuples from or write tuples into one of the two log files. In order to support this, it maintains two other objects of BufferFileData viz bfdForReadFile and bfdForWriteFile. The former represents the file from which tuples needs to be read and the later represents the file into which the data needs to be written. Initially when the buffer is created both of them points to the first file. Changing of bfdForWriteFile from file one to file two in done in enqueue operation and that of bfdForReadFile is done in the process to read tuple from secondary storage.

5.2.4 Swapping the two Buffer File

```
    if ( (vctBuffer.size() > intBufferSize && intBufferSize != -1) ||
bfdForWriteFile.intTotalFileElements > 0){
      if (bfdForWriteFile.intTotalFileElements > intBufferFileCapacity){
       // Swap the File.
       try{
        if (bfdForFileOne.rf == null){
         bfdForFileOne.rf = new
RandomAccessFile(bfdForFileOne.strFileName,"rw");
         bfdForFileOne.intTotalFileElements = 0;
         bfdForFileOne.intTotalReadFileElements = 0;
         bfdForWriteFile = bfdForFileOne;
         itIndexTableForFileOne = new IndexTable();
         itIndexTableForWrite = itIndexTableForFileOne;
        }
        else if(bfdForFileTwo.rf == null){
         bfdForFileTwo.rf = new
RandomAccessFile(bfdForFileTwo.strFileName,"rw");
         bfdForFileTwo.intTotalFileElements = 0;
         bfdForFileTwo.intTotalReadFileElements = 0;
         bfdForWriteFile = bfdForFileTwo;
         itIndexTableForFileTwo = new IndexTable();
         itIndexTableForWrite = itIndexTableForFileTwo;
        }
       }
       catch(FileNotFoundException e)
         {System.out.println(" Enqueue " + e.getMessage()); System.exit(1);}
      }
      long lngOffset = itIndexTableForWrite.writetoIndexTable(objInputElement);
      logWrite(bfdForWriteFile.rf,objInputElement,lngOffset);
      bfdForWriteFile.intTotalFileElements++;
}
```

Figure 5.4 Code for Swapping Two Buffer Log Files in Enqueue

Figure 5.4 shows the code to swap two buffer log files. Initially bfrForWriteFile points to the first file. If the number of tuples in the main memory buffer exceeds the predefined capacity of the buffer, the tuple needs to be written to the secondary buffer. Initially, buffer creates the first log file and initializes the index table corresponding to it. Later it sets the bfdWriteToFile as the first file. Henceforth all tuples entering into the buffer will be written to the secondary buffer till all elements from the secondary buffer have been completely read into the main memory buffer. To provide an easy

purging of tuples in secondary storage, a limit is specified on the total number of tuples that can be stored in a file. If the file exceeds that limit, buffer creates a second file and starts writing into the second log file by pointing the bfdWriteToFile to the other file. Now tuples are enqueued into the other file till all elements have been read from the first file and the second file exceeds the specified limit. This way there could be more number of tuples stored in the log files than its specified limit. Hence there could be an uneven growth in the size of the two log files.

To enqueue a tuple in the secondary storage, first the tuples information is stored in the current write index table. Using the offset that index table provides, the tuple is written into the bfdWriteToFile using the logWrite method.

5.2.5 Reading From Secondary Storage:

There is a separate thread to read data from the secondary buffer log files. This thread makes use of bfdForReadFile object and logRead method to bring a tuple from the secondary storage log file to the main memory buffer. This thread is started from the dequeued method. It is started only if there are tuples stored in the secondary storage file. This is verified by checking if the totalElements of bfdForReadFile is greater than zero and totalReadElements is less than the totalElements stored in the file.

The process to read from secondary storage continues until there is space left in the main memory and there are unread tuples from the secondary storage. Initially bfdForReadFile points to bfdForFileOne. The read process reads all the tuples from first file and enqueues it in the main memory buffer. During the reading process if all tuples

are read from first file and there are unread tuples from the other log file, and there is some space left in the main memory buffer then bfdForReadFile needs to be swapped to the other file. The swap process does the following things:

- It sets the current bfdReadFile's filepointer to null.
- Resets the current read indextable.
- Swaps the bfdReadForFile to point to the other log file.
- Swap the current read indextable to the one corresponding to the other log file.

```
   while(vctBuffer.size() < intBufferSize){
    if (bfdForFileOne.rf == null && bfdForFileTwo.rf == null)
     break;
    if (bfdForReadFile.intTotalReadFileElements >=
bfdForReadFile.intTotalFileElements){
     if (bfdForReadFile == bfdForFileOne && bfdForFileTwo.rf != null){
      bfdForReadFile.rf = null;
      bfdForReadFile.intTotalFileElements = 0;
      bfdForReadFile.intTotalReadFileElements = 0;
      bfdForReadFile = bfdForFileTwo;
      itIndexTableForRead.clearIndexTable();
      itIndexTableForRead = itIndexTableForFileTwo;
     }
     else if (bfdForReadFile == bfdForFileTwo && bfdForFileOne.rf != null){
      bfdForReadFile.rf = null;
      bfdForReadFile.intTotalFileElements = 0;
      bfdForReadFile.intTotalReadFileElements = 0;
      bfdForReadFile = bfdForFileOne;
      itIndexTableForRead.clearIndexTable();
      itIndexTableForRead = itIndexTableForFileOne;
     }
     else
      break;
    }
    Node indexNode =
itIndexTableForRead.getRecord(bfdForReadFile.intTotalReadFileElements++);
    Vector vctTupleFromFile = (Vector)
logRead(bfdForReadFile.rf,indexNode.getByteSize(),indexNode.getFP());
    enqueueFileTuple(vctTupleFromFile);
 }
```

Figure 5.5 Code for Reading from Secondary Memory

5.2.6 Experimental Evaluation:

All the experiments were performed on an unloaded machine with 2 Xeon processors, 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 and is used as a stream to this system. Before feeding the MavHome data to the streams, the source time stamp is modified such that its value is in an increasing order with the *first* tuple showing its value to be *one* and the *nth* tuple showing its value to be *n*. Tuples are generated with a poison distribution to simulate real time data.

This experiment shows the effect of varying buffer size on the system. Query used in the experiment is "select txtDeviceID, txtStatus, txtPropertyValue from S1, S2 where S1.txtPropertyValue=S2.txtPropertyValue and S1.txtStatus=On and S2.txtStatus=On."

This experiment was run on five Windows each consisting of 5000 tuples. The Buffer Size was changed from being unbounded to 35000, 30000, 25000, 20000, 18000, 15000, 12000 and 10000 tuples per buffer.

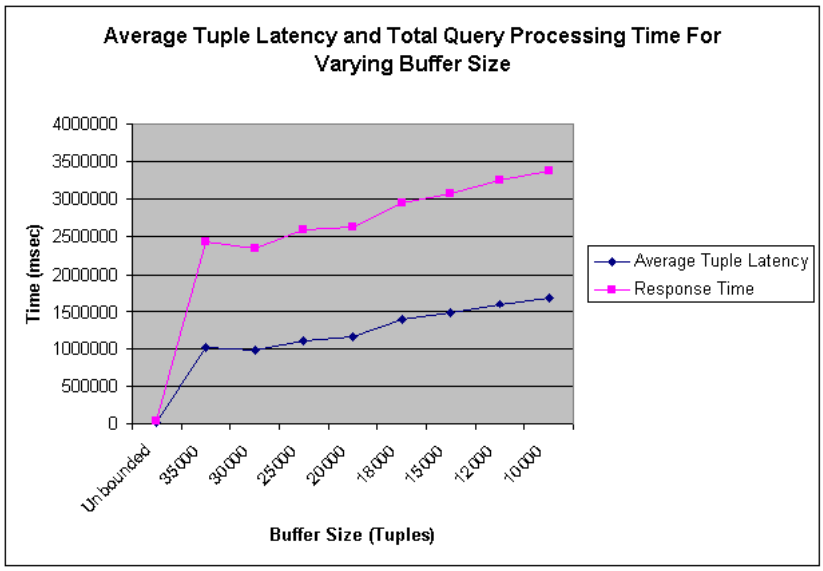

Figure 5.6 Effect of Varying Buffer Size.

Observation:

From the graph (Figure 5.6) it can be seen that unbounded buffer quickly produced the output. This is because all the tuples were main memory resident. As the buffer is contracted, more and more tuples are getting output into the secondary buffer. This induces a lot of I/O cost for reading and writing tuples. Hence the total tuple processing time of the query increases. Moreover a tuple starts staying for longer time in the queues because of IO read-write that increases the tuple latency.

5.3 Query Window Implementation:

All window-based operators need to define the following four clauses.

- Begin Window.
- End Window.

- Hop Size (lower bound, upper bound).
- End Query.

These clauses are defined based on the user definition of the query. After defining these parameters, an operator using a window would require to get the current window definition, it would also require to check the bounds of the next window. Also, the operator would be interested in knowing whether the process of moving to the next window brings the end query condition or not. To accommodate all these parameters of a window and its operation, a separate window class is defined. The object of this class can be used in window-based operators to provide a controlled access to window and its parameters.

The QueryWindow class has the following parameters:

- lngBeginWindow.
- lngEndWindow.
- lngHopSize[].
- lngEndQuery.

Long values are chosen for these parameters so that they can represent both physical window as a time stamp and logical window as number of tuples. Hop size is an array with the zeroth value being the lower bound and the first value being the upper bound.

This class supports the following operation.

- GetFirstWindow(): Returns the lower and upper bound of the first query window.

- GetNextWindow(): Returns the lower and upper bound of the next window from the current window.
- IsQueryTerminating(): Returns true if the lower bound of the window falls beyond the end query specification. Other wise it returns a false value.

5.4 Operators

5.4.1 Select:

SELECT is implemented by inheriting the generic operator. It stores the condition as a string that will be evaluated for filtering the data. It uses FESI as the condition evaluator.

One of the challenges in implementing SELECT was to provide the operand values from the input tuples to FESI. We cannot directly supply the tuple value to FESI for evaluation. Also it does not make sense to set all the operands from the input tuple into FESI's condition evaluator. Only the operands mentioned as a part of the condition needs to be evaluated. To read the operand value from the input tuple, we need to know the position of the operand in the tuples. The position of the operands in the tuples is stored in the schema. Hence an interface needs to be provided that can find the operands from the condition string and then find their position in the tuples. Using the position information, the corresponding values from the input tuples needs to be set against the operands in the FESI's condition evaluator.

The following two functions help retrieve the position of operands from the input string.

- *findOperandsInConditionString:* This method parses the condition string and separates out the operands from it.
- *findPositionOfOperands:* It uses the operand list provided by *findOperandsInConditionString* function and finds out the position of each operand from the schema. This list of position is used by the SELECT for setting operand values in the condition evaluator.

The conversion of condition string to a list of position using the above-mentioned functions is done before the actual tuple evaluation is started. This is done when the condition string is submitted to the SELECT operator.

5.4.1.1 SELECT Example:

Consider the evaluation of the following condition string on the input string.

Condition String: txtDeviceID = ‘M18’ and txtStatus = ‘On’.

The schema for the input buffer is as described below.

Attribute Name	Position
TxtTrandID	1
TxtDeviceID	2
TxtStatus	3
TxtPropertyValue	4
TxtCommandSource	5
NumSourceTimeStamp	6

FindOperandsInConditionString: Given the condition string as an input, the output of this function is txtDeviceID, txtStatus.

Given the operand list as input to *findPositionOfOperands,* the output given is 2 and 3.

1	M18	On	10	RF_Remote	10001010102

Give this tuple for processing, the operands operand value map would be

TxtDeviceId = M18

TxtStatus = On

Evaluation: Output of evaluation would be true.

5.4.2 Hash Join:

Hash join is implemented as a non-blocking operator that works on a window unit of data and produces results incrementally and continuously to process continuous queries. As explained in the design section, it is implemented using a single thread and processes each tuple atomically to avoid duplicates. It reads tuples from each of its external buffers, compare their timestamp and the one with lower timestamp is considered for processing. If the left tuple is being processed, it is inserted in the left hash table. It is then probed in the right internal hash table and matched with all the tuples falling in the window and satisfying the join predicate. The results are produced incrementally that are consumed by higher operators. It is this insertion and probing phase that is to be done atomically to avoid duplicates. Tuples read from right external

buffers are processed analogously. It is important to understand the implementation issues prior to delving into implementation details. Following are the implementation issues for the “Hash Join” operator:

- Window Generation
- Maintaining internal hash tables
- How timestamp ordering is maintained
- Atomic action to avoid duplicates

Window Generation: It is important to understand the concept of a window prior to the implementation specific details of join operator since the working of “Join” operator heavily depends on window processing. One of the requirements of stream operator is their ability to work on windows independently. Operators of the same query should be able to operate on different windows simultaneously. To facilitate this requirement, an object of “Query Window” is defined that deals with window creation and modification.

Internal hash tables: “Join” operator has two external buffers associated with it that provides synchronized input to this operator. There are two internal hash tables maintained by this operator, one corresponding to each of the external buffers. These hash tables are used not only for computing hash join but also to avoid repetitive scan on external buffers. A single tuple processing involves insertion and probing phase. During insertion phase, the tuple is placed in the corresponding internal hash table upon which it is no longer needed in the external buffer for the following reasons:

- The tuples in internal hash table participates in the join operation with new incoming tuples.

- It is the internal hash table that is scanned to find the match when a new tuple is probed against it.

Since the responsibility of finding the matched tuple is delegated to internal hash table, repetitive scan on external buffers is avoided. Internal hashtables not only reduces load on the external buffer but also allows the tuples that are read from external buffers to be removed to create space to accommodate new tuples coming either from data streams or child operators.

Timestamp ordering: The significance of timestamp ordering is explained in the design section. It is needed to determine the end window bounds and produce correct results. Tuples generated by the data sources are time stamped as soon as they enter the system. Hence for the first windowed operator, all incoming tuples are timestamp ordered. Unless some measures are taken to maintain timestamp ordering in the output of current windowed operator, the output will not be timestamp ordered, which is essential for higher windowed operators as well. To produce tuples in a timestamp order, each tuple is blocked at the input and is not considered for join until it finds a corresponding tuple from the opposite stream. When it is joined with tuples present in internal buffers, resultant tuples are generated with the timestamp of the higher of the two. All tuples in the output queue are guaranteed to be timestamp ordered.

Duplicate tuple avoidance: The use of two threads to increase the parallelism of the entire "Join" operation was discarded since it was resulting in duplicates at the output. The only way to avoid duplicates in the output is to process individual tuple atomically. The next tuple is not considered for processing until the insertion and

probing phase of the current tuple is completed. This avoids the possibility of processing two tuples simultaneously that arrive with the same timestamp in left and right external buffers. Synchronized operation resolves time conflicts and process single tuple at a time ensuring duplicate avoidance.

5.4.2.1 Hash Join without temporary storage:

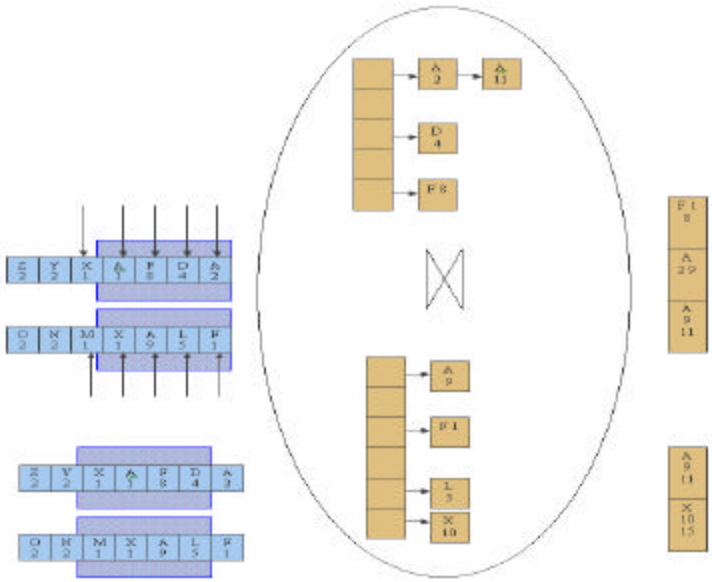

Figure 5.7 Hash Join Without Temporary Storage

Figure 5.7 indicates that join has two external buffers and two internal hash tables. Operator reads from these external buffers, performs the needed join operation and outputs the result incrementally and continuously in its associated output queue. Windows are defined on the external buffers that mark the input bounds for the operator. It is assumed that window definitions are always the same for both the external buffers. These window bounds are defined using “Query Window” class

explained above. The APIs of this class are used to control window movement besides providing window information. The window slides only when all the elements from each of the external buffers are completely processed. Left window bounds are defined by LW1S and LW1E, which stands for "Left Window One Start" and "Left Window one End" respectively. Right window bounds are defined by RW1S and RW1E.

Call purging logic: All the past tuples that are not relevant in current window processing are purged to create space in the external buffers. Since the operator knows LW1S, it compares the timestamp of the read tuple 't' with LW1S. If 't' < LW1S, it is considered as a stale tuple. This process is repeated until the "join" operator finds the first tuple that falls in the current window bound. The position of this tuple is marked and stored as Highest Read Element (HRE). If multiple operators share the buffer, all HREs are compared and the one with the smallest value is declared Highest Common Read Element (HCRE). Thus all elements prior to HCRE can be safely discarded as they are already been read and utilized by all the operators sharing the buffer.

Window Processing: Every buffer has a CurrentUnreadElementPointer (CUEP) for each operator that point to the current element to be read from the buffer for the respective operator. The value of CUEP is incremented each time a tuple is read by the "Join" operator. Consider a single tuple processing for the join operator. Tuples are read from each of the input queues with the left tuple (LT) timestamp as 'x' and the right tuple (RT) timestamp as 'y'. If 'x' < 'y', 'x' is processed which involves probing LT into right hash table based on the value of joining attribute. Since 'x' corresponds to LT, it is hashed into the left internal hash table based on the value of the joining attribute. It is

then joined with all tuples lying in the bucket (tuples having same attribute value called collision, map to the same location and are stored linearly at the probed location) of the probed location and the output is generated. During the join operation the resultant tuple has the common attribute removed from the right stream. The timestamp of the resultant tuple is the higher timestamp of the two tuples involved in join since this algorithm ensures ordering on higher timestamp. Since 'x' < 'y', CUEP of left external buffer is incremented by 1. This insertion and probing phase must be done atomically to avoid duplicates.

Window movement: In overlap windows, it is important to remember the start time of the next window while the current window is being processed. Since the window definition is known, the start time is marked as soon as the first tuple is encountered that falls in the next window. This position is saved as Start Next Window Pointer (SNWP). For disjoint windows, no modification is needed as they both point to the same location. In Hash Join without temporary storage, the internal hash tables are completely cleared since every window is computed independently. Since it does not reuse the computation the CUEP that points to LW1E is modified to point to SNWP to continue processing the next window from start. The common computation is not stored anywhere for utilizing in the next window computation; this shade is called "Hash Join Without Temporary Storage".

5.4.2.2 Hash Join With temporary Storage:

This variation exploits the common computation in the overlap window by reusing the results of current window computation in the next window. While the current window is processed, the common computation is identified and the results are stored in the temporary storage besides producing them as output.

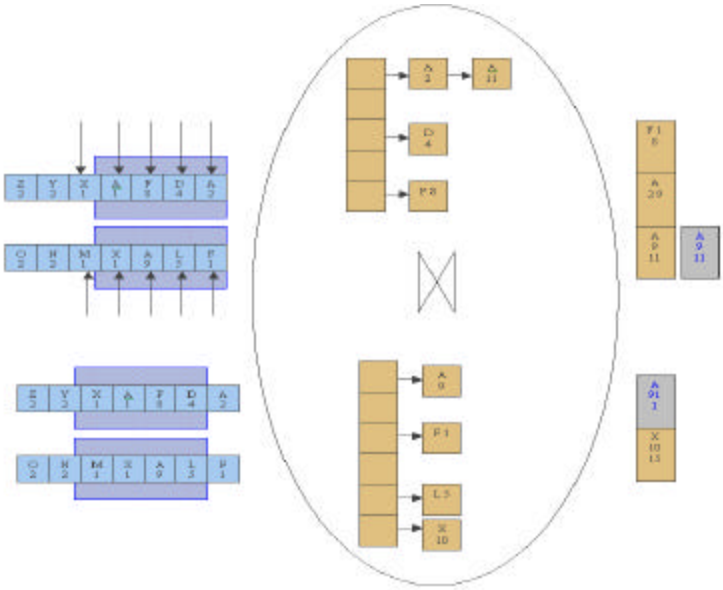

Figure 5.8 Hash Join With temporary Storage

This variation exploits the common computation in the overlap window by reusing the results of current window computation in the next window. While the current window is processed, the common computation is identified and the results are stored in the temporary storage besides producing them at output. Tuples falling beyond SNWP are considered to be a part of the overlapped area. The result accumulated in

temporary storage is copied as it is in the output queue corresponding to next window. Once the window is completely processed, internal hash tables are not completely cleared unlike the other variation, rather entire hash table is enumerated and only those tuples whose timestamp falls below LW2S are removed. Since the computation is being re-used, CUEP does not point to LW2S but points to LW1E and moves only in forward direction. Hence all tuples falling in the next window computation in the internal hash table are preserved. Purging logic is called at the end of each window computation that removes all elements marked up to HCRE to create space for incoming tuples generated either by data sources or child operators. Since the computation is re-used, it is more efficient with respect to response time but less efficient with respect to memory utilization since hash tables are never cleared besides involving temporary storage for storing common computation.

5.4.2.3 Experimental Evaluation:

The Experimental Set up is same as the one used in Buffer Experiments.

1. Effect of Varying Data Rate on Hash Join.

HashJoin Without Reuse is fed tuples with a variable data rate. The data rate used were 100, 150, 200, 400, 800, 1600, 3000, 5000, 8000 and flooding (Here the time gap between two tuples was almost zero. Hence it flooded the stream buffers with data). Some of the constant set of information across this experiment is as given below.

Total Windows: 5

Tuples per Window: 20000.

Data Set: MavHome Data Set with the time stamp modified to form an incremental value starting from one. Same data set as one used in buffer.

Window Overlap = 10%.

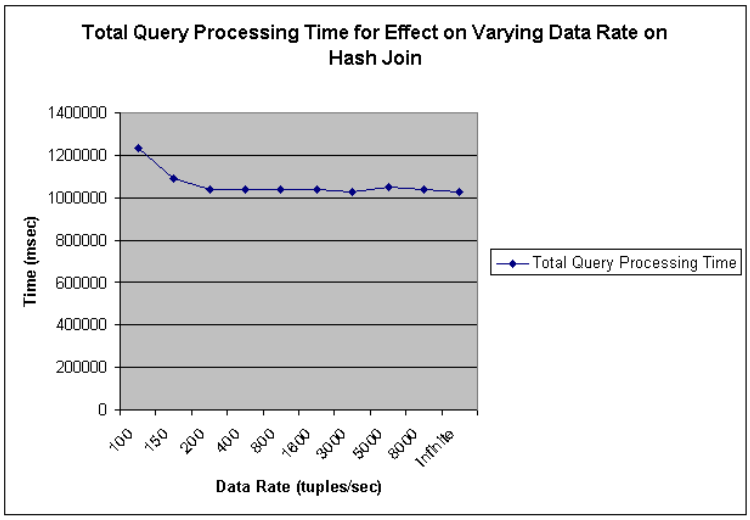

Figure 5.9 Total Tuple Processing Time

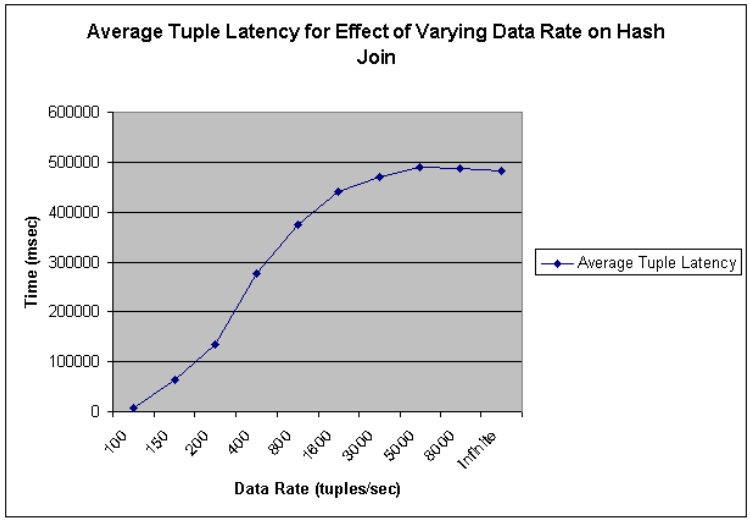

Figure 5.10 Average Tuple Latency

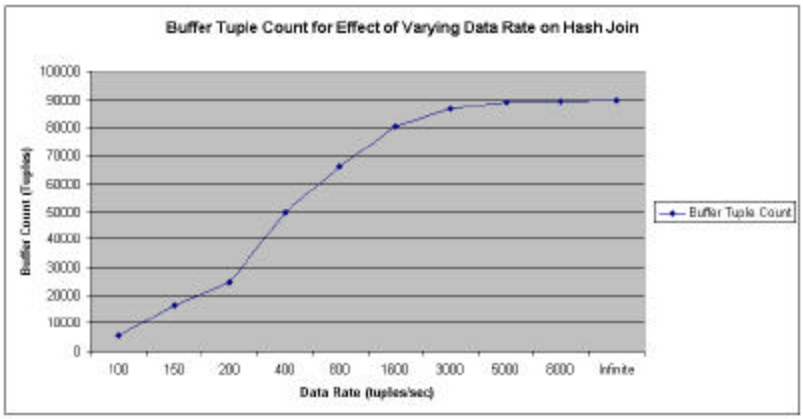

Figure 5.11 Buffer Size

Observation:

From the graph 5.9, it can be seen that initially when the data rate is very low at 100 tuples/sec, the total query processing time is quite high. But as the data rate increases, the total query processing time decrease but this happens till some point and after that the total query processing time stabilizes to a constant value. This is because the amount of data input to the join is the same but the data rate has reached a value such that hash join will always get some tuples in the buffer and will never suspend. Here we see that the HashJoin is operating at its maximum processing capacity and because of that the total query processing time always remains the same.

From the graphs 5.10 and 5.11, we can see that as the data rate increases, the average tuple latency increase and the Buffer Tuple count also increases. This is because more tuples are being input to the Join but it has a limited processing capability and so the amount of time a tuple spends in the queue increases thereby increasing the overall average tuple latency.

2. Comparison of Hash Join With and Without Temporary Storage (Reuse and Without Reuse).

Keeping a constant data rate, and running the experiment over the same number of windows having same number of tuples, average tuple latency, response time and Hashtable size are compared for Hash Join with and without temporary storage. To show the effect of reuse, window overlap was increased from 10%-75% of the current window. The data set is modified such that we have a uniform distribution of tuples. We have the same number of tuples in each window and each window will generate the same number of output irrespective of the amount of overlap. This way we can compare the results between both the joins across windows with different overlaps. Here the window is kind of a logical window as the timestamp is incrementally ordered starting from one

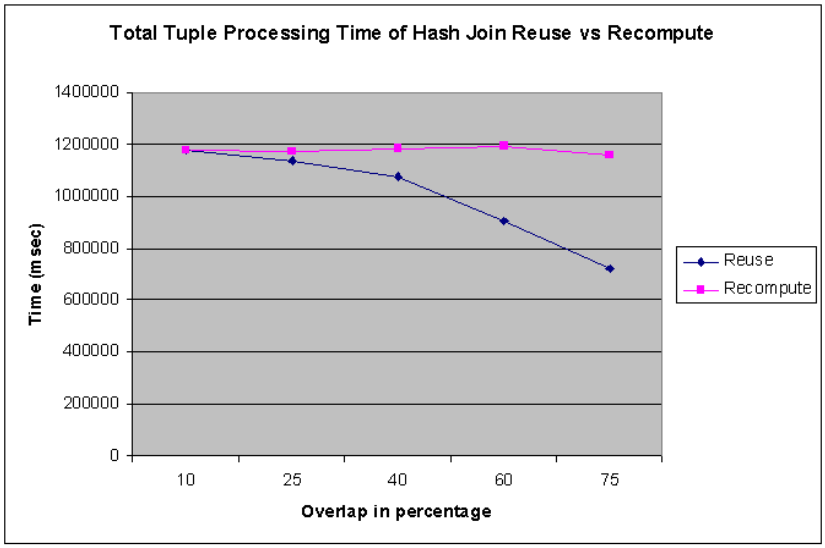

Figure 5.12 Total Tuple Processing Time (Join Comparison)

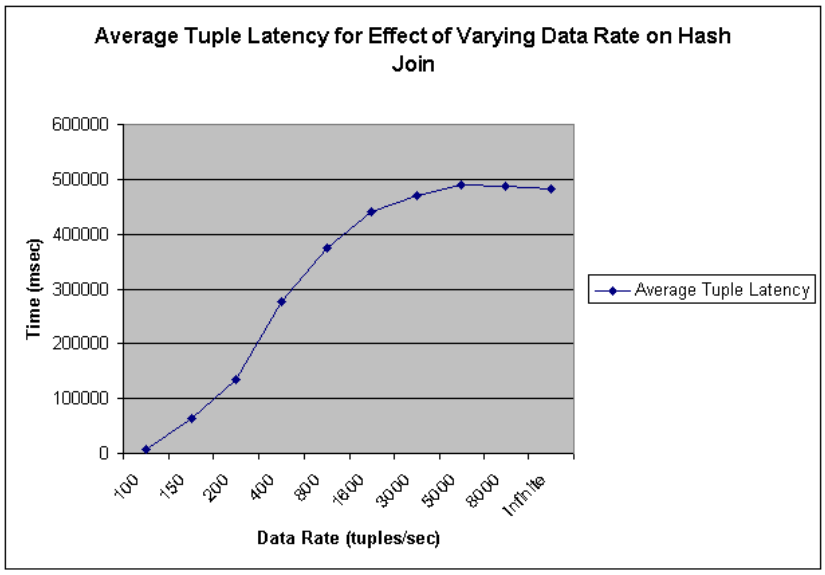

Figure 5.13 Average Tuple Latency (Join Comparison)

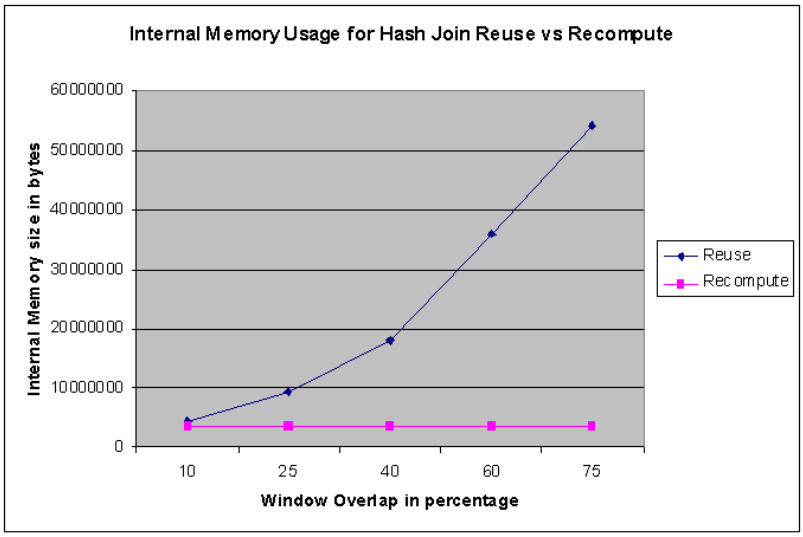

Figure 5.14 Internal Memory Usage (Join Comparison)

Observation:

Total Query Processing Time: There is no affect on the total query processing time for Hash Join Without Temporary Storage. This is because it has to read all the

tuples for all windows and process them irrespective of the amount of overlap. Moreover the data set is such that the number of tuples generated would be the same for all cases and so a constant response time is observed. For Hash Join With Temporary Storage the overlapping of windows is exploited by reusing the output tuples generated for the overlap region. Therefore we can see that the total query processing time decreases as the overlap is increasing.

Average Tuple Latency (ATL): ATL of HashJoin With temporary storage is always lower that that of Without Temporary Storage. This is because tuples stay for longer time in queues for Hash Join Without Temporary as they needs to be recomputed for all windows irrespective of the overlap. On the other hand HJ With Temporary Storage overlapping tuples does not add to the overall tuple latency.

Internal Hash Table Size: For HashJoin With Temporary storage the size of internal Hashtable increases as the overlap increases. This is because as the overlap increases, more and more number of tuples is stored in temporary storage.

5.5 Instantiator

5.5.1 Instantiator Implementation:

DSMS client provides an interface through which the user creates the initial plan for the DSMS query. Each query plan is constructed as a tree data structure. DSMS client sends out command number “2” in order to instantiate a query. On receiving command “2”, server waits to receive a plan object as part of query instantiation protocol. DSMS client sends out the plan object created using the user input to the server, which then instantiates it and schedules the operators in the query tree.

5.5.1.1 Plan Object:

Plan object is a tree data structure. The tree is created using the OperatorNode class.

```
import java.io.*;

public class OperatorNode implements Serializable {

public OperatorData optData;
public OperatorNode optNodeLeftChild;
public OperatorNode optNodeRightChild;

}
```

Figure 5.15 Operator Node Class

Figure 5.15 shows the structure of Operator Node class. The operator node class has the following three properties.

- OperatorData: This is the operator data node that stores the required information in order to create and instantiate an operator object.
- NodeLeftChild: This points to an object of Operator Node class and represents the left child of the current object.
- NodeRightChild: Again this points to an object of Operator Node class and represents the right child of the current object.

Communication between client and server takes place over the socket. All communication between the client and the server is object based. Objects converted into serialized bytes are sent over sockets between server and the client. Hence the OperatorNode object implements the serializable interface. This helps to send out the plan object from the client to the server. What client sends is the root object of the

OperatorNode to the Server. In turn this sends the entire query tree using the transitive reachability mechanism. This means that all objects reachable from this object will also be serialized. The same process is repeated for the child OperatorNode till we reach the leaf nodes in the tree. Hence the entire tree is given as an input to the server just by outputting the OperatorNode representing the root.

Figure 5.16 shows a plan object of the query: Display the common device usage between rooms A and B in MavHome lab.

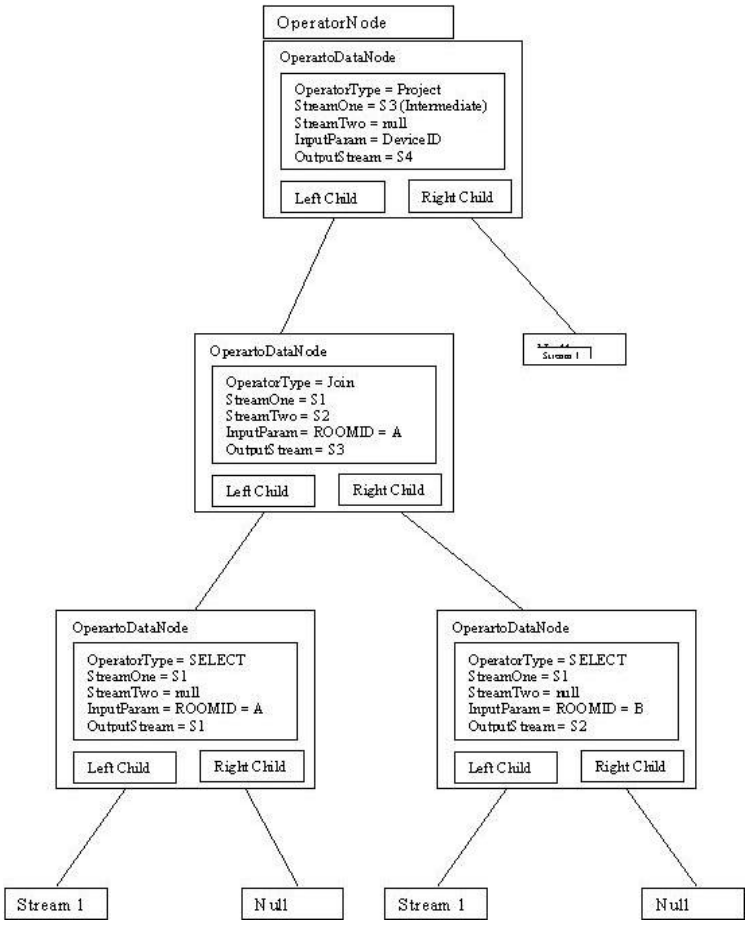

Figure 5.16 Plan Object for a Query

5.5.1.2 Operator Data Node:

Operator Data Node is implemented as a separate class with the following parameters.

strOperatorType: It is a string that stores the type of operators e.g. SELECT, PROJECT, JOIN.

strStreamOne, strStreamTwo: Left and Right Stream names are stored as a string.

strInputParameters: As a string it can store the input parameters for any operator. For example, the condition string for SELECT operator for a particular query is ”deviceID=M8 and RoomId=R1” or for PROJECT the list of attributes to be projected could be “deviceId,RoomID,trandDateTiem”.

lngBeginWindow, lngEndWindow, lngEndQuery: Windowing parameters are stored as a long value so that it can represent both physical as well as logical windows. For physical window, it can store the tuple time stamp and for logical window it can store the number of tuples.

lngHopSize[]: Hope Size is implemented as a long array, which will store two values. The zero value will represent the lower bound and the first value will represent the upper bound for the hop of the window.

5.5.1.3 StreamBufferList:

It is implemented as a Hashtable with the key as Stream Name and value of the object of Buffer in which the stream’s tuples are being fed. This is done to provide a quick mapping between Streams and its buffers.

5.5.1.4 Instantiation from Plan Object:

The process of instantiating operators from the plan object and linking them with buffers is implemented as a recursive algorithm. It takes in the OperatorNode and a Boolean value, which says whether it is a left child, or not. Initially the value of root node is passed and it is declared to be a left child even it does not have any parents. The query tree is traversed in a post order fashion. This is done so as to get a bottom up instantiation of operators there by supporting flow based scheduling.

Once an Operator Node is extracted from the tree, it is checked to see if it is a leaf node. If so, then using the Stream names from the OperatorData, it will pick up the corresponding buffer from the StreamBufferList and pass it on to instantiate the operator.

For non-leaf nodes, the Left and Right input buffers are set while instantiating the operator. Based on whether the instantiated operator is a left or right child the corresponding output buffer of that operator is set to the left input buffer or the right input buffer of the parent operator in the tree.

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

5.6 Server Implementation

DSMS is implemented as a TCP Server listening to port number nnnn. Client communication with the server is command driven and protocol oriented. DSMS

defines a list of functions (commands) that it supports. For each command, a protocol is defined that drives the client and server communication. All protocols clearly define what input the server is expecting from the client and what would be the server's response. Communication between client and server is object based For example, when a client sends a query tree object, the server instantiates it and returns a query handle which the client can use for further query control.

DSMS server maintains different kinds of data structures used by several modules shown in the DSMS architecture. These data structures are updated at various events.

5.6.1 Server Implementation

DSMS server implementation model is based on the following two important classes.

- TCPServer: TCPServer is an abstract class *TCPServer* that implements the generic functionality of client server communication. It mounts the DSMS server at a given port and listens for the clients to communicate. Once the client communication is detected, it creates a *NetStream* Object using the client socket. Thereafter it calls the *service* method. This method is an abstract method, which takes in an object of *NetStream* and implements the actual functionality of the server. *Service* is an abstract method, which needs to be implemented in the class extending TCPServer.
- NetStream: NetStream provides functions for an object-based communication between the client and the server. It accepts a socket as its input. It wraps the

input and output streams of the given client socket using the Object input and output streams there by providing object based communication over the sockets. *NetStream* implements java.io.Serializable there by providing a serialized object base communication over the sockets.

Java object serialization provides the ability to write or read java objects to and from a byte stream. It allows Java objects and primitives to be encoded into a byte stream suitable for streaming to a network or to a file-system. The Java Serialization API provides a standard mechanism for developers to handle object serialization. The API is small and easy to use.

5.6.2 DSMS Server Commands/Functionalities

DSMS Server extends the class *TCPServer*. Hence it needs to implement the *service* method of *TCPServer*. All the functionalities of DSMS are implemented in the *service* method. DSMS is a command-based server. Client sends a command to the server and then follows the protocol for that command. Integer based commands are used. For e.g. to send a query to the server, client invokes the command number "1". Each command corresponds to a functionality that DSMS server provides to the server. A limited set of commands is currently available in the system. But this can be extended as and when needed.

Chapter 6

Conclusion and Future Work.

This thesis explains the design of query processing architecture for streaming data. This architecture provides a model for representing a general-purpose stream database application. Different types of streaming queries are identified and a general-purpose query representation is proposed. This representation covers all types of windows.

A generic streaming operator is proposed which satisfies the query processing requirements of stream data. This provides a base model for implementing future operators. SELECT and PROJECT are designed and implemented to support streaming data. Two shades of Hash Based Joins are designed and implemented. One that exploits the overlapping nature of query windows by using a temporary storage and the other that treats all windows in the same manner. Experiments have shown that for overlapping windows, Hash Join With Temporary Storage is efficient over Hash Join Without Temporary Storage.

Query Instantiator is designed and implemented that accepts a plan object generated by Stream database clients and instantiates the required operator to provide stream-based output to the end user.

The DSMS Server is designed and implemented to provide a platform for integrating various components of Stream Database Architecture.

To completely implement the proposed architecture for stream database, an alternate plan generator and a runtime query optimizer needs to be designed and implemented.

Alternate Plan Generator accepts a user given plan and generates different equivalent plans for the same query. These alternate plans are considered by the runtime query optimizer to generate an efficient global plan. In addition, the runtime query optimizer also monitors the QoS to see if they are being satisfied and if not take some corrective action (change priorities, load shedding) to try and satisfy as many QoS requirements as possible.

The DSMS is a server-based architecture and is prone to system crash. Hence there arises a need for a Recovery System, which can bring the system back to the state at the time of crash. This is currently being investigated.

REFERENCES

1. Carney D, C.U., Cherniack M., Convey C., Lee S., Seidman G., Stonebraker M., Tatbul N. & Zdonik S., *Monitoring Streams.* In Proc. 28th VLDB, 2002.

2. http://ranger.uta.edu/smarthome/

3. Babcock B., B.S., Motwani R., Widom J, *Models and Issues in Data Stream Systems.* In Proc. ACM Sigmod/Sigact Conference on Principles of Database Systems, 2002.

4. Hellerstein, A.R., *Eddies: Continuously adaptive query processing.* In ACM SIGMOD, 2000: p. 261-272.

5. Bonnet P., G.J.S.P., *Towards sensor database systems.* In 2nd International Conference on Mobile Data Management, 2001.

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

7. Franklin., C.S.M., *Streaming Queries over Streaming Data.* In 28th VLDB Conference, 2002.

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

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

10. David., J.C.J., *Dynamic Regrouping of continuous queries.*

11. Motwani R, W.J., Arasu A., Babu S., Datar M., Olston C. & Varma R., *Query Processing Resource Management and Approximation in a Data Stream Management System.* IEEE Data Engineering Bulletin, January 2003. **26, No1**.

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

13. Sreekant., T., *Recoverable global event detector for distributed active applications.* Masters thesis in CSE Department 2002, University of Texas at Arlington.

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

BIOGRAPHICAL INFORMATION

Altaf Gilani was born September 08, 1978 in Mumbai, India. He received his Bachelor of Engineering degree in Computer Engineering from Veermata Jijabai Technological Institute (VJTI) of Mumbai University, Maharashtra, India in Aug 2000. There after, he worked as an Associate Consultant with I-flex Solutions from Aug-2003 till July 2001. 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. He is currently working as a Design Engineer with NOKIA INC. His research interests include query processing for stream-based applications.