A FRAMEWORK FOR SUPPORTING QUALITY OF SERVICE REQUIREMENTS IN A DATA STREAM MANAGEMENT SYSTEM

by

QINGCHUN JIANG

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

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT ARLINGTON
August 2005

To my father, mother, wife, and my son

ACKNOWLEDGEMENTS

I am truly fortunate to have had the opportunity to work with my adviser: Sharma Charavarthy in the past few years. He, by example, has taught me to set up higher standards and goals and then work toward the goals. I am overwhelmed by gratitude for his patience and his guidance throughout my Ph.D study at The University of Texas at Arlington. I would like to thank him for being an excellent adviser. I am also looking forward to working with him outside of the UT Arlington.

Another critical support group during my Ph.D career has been my research committee members. I would like to thank Y. Alp Aslandogan, Diane J. Cook, Sajal K. Das, and Jung-Hwan Oh for their careful reading of this dissertation and for providing me with valuable feedback and comments.

The ITLAB members over the years have provided me valuable relief and encouragement when I needed it. Raman Adaikkalavan along with other ITLAB members (past and current) has been wonderful officemates and provided me with an excellent computing and working environment. They ensured that I could focus on my research by handling all administration details. Similarly, I would like thank the CSE department and the National Science Foundation for providing me financial supporting.

Finally, I am eternally thankful to my family. To my Mom, Dad, and brothers, it is largely because of your constant support and encouragement that I am completing my Ph.D. To my wife, Ailing and my son, George: you have made my life so much wonderful in the past few years and have provided me with inspiration in all walks of life.

July 13, 2005

ABSTRACT

A FRAMEWORK FOR SUPPORTING QUALITY OF SERVICE
REQUIREMENTS IN A DATA STREAM
MANAGEMENT SYSTEM

Publication No. ______

Qingchun Jiang, Ph.D.
The University of Texas at Arlington, 2005

Supervising Professor: Sharma Chakravarthy

Currently, a large class of data-intensive applications, in which data are presented in the form of continuous data streams rather than static relations, has been widely recognized in the database community. Not only is the size of the data for these applications unbounded and the data arrives in a highly bursty mode, but these applications have to conform to Quality of Service (QoS) requirements for processing continuous queries (CQs) over data streams. These characteristics make it infeasible to simply load the arriving data streams into a traditional database management system and use currently available techniques for their processing. Therefore, a data stream management system (DSMS) is needed to process continuous streaming data effectively and efficiently.

In this thesis, we discuss and provide solutions to many aspects of a DSMS with the emphasis on supporting QoS requirements and event and rule processing. Specifically, we address the following problems:

System Capacity Planning and QoS Metrics Estimation: We propose a queueing

theory based model for analyzing a multiple continuous query processing system. Using our queueing model, we provide a solution to the system capacity planning problem and its reverse problem: given the resources and CQs, how to estimate QoS metrics? The estimated QoS metrics not only can be used to verify whether the defined QoS requirements of CQs in a DSMS have been satisfied, but also form the base in a DSMS to manage and control various QoS delivery mechanisms such as scheduling strategies, load shedding, admission control, and others.

Run-Time Resource Allocation (Scheduling Strategies): We propose a family of scheduling strategies for run-time resource allocation in DSMSs, which includes the operator path capacity strategy (PC) to minimize the overall tuple latency, the operator segment strategy and its variances: the memory-optimal segment strategy (MOS), which minimizes the total memory requirement, and the simplified segment strategy, and the threshold strategy, a hybrid of the PC and the MOS strategy.

QoS Delivery Mechanism (Load Shedding): We develop a set of comprehensive techniques to handle the bursty nature of input data streams by activating/deactivating a set of shedders to gracefully discard tuples during overload periods in a general DSMS. We first formalize the problem and discuss the physical implementation of shedders. We then develop a set of algorithms to estimate system capacity, to compute the optimal location of shedders in CQs, and to allocate the total shedding load among non-active shedders.

Event and Rule Processing We develop an integrated model, termed Estream, to process complicated event expressions and rules under the context of data stream processing through a group of enhancements to a DSMS. Our Estream model greatly improves the expressiveness and computation ability of DSMSs in terms of processing complex real-life events and makes DSMSs actively respond to defined events over data streams and carry out defined sequences of actions automatically.

Our algorithms and solutions developed in this thesis can be used individually to assist QoS support in a DSMS. The most important contribution of this thesis is that these algorithms and solutions form a framework for supporting QoS requirements QoS requirements in a general DSMS.

Finally, the theoretical analysis is validated using a prototype implementation. We have prototyped the proposed solutions, algorithms, and techniques developed in this thesis in a general DSMS, termed MavStream, in C++.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Traditional database management systems (DBMSs), consisting of a set of persistent relations, a set of well-defined operations, and a highly optimized query processing engine, have been researched for over 25 years and are widely used in applications that require persistent storage and processing of ad hoc queries to manage and process large volume of data. Usually, the data processed by DBMSs are less frequently updated data items such as customers information, employees' salary, and so on and the features provided by a DBMS are consistency, concurrency and recovery over well-defined operations over relations such as *insertion*, *deletion*, and *update* and its efficiency for supporting large volume of transactions over persistent data.

However, the past few years have witnessed a large class of data-intensive applications that produces high-frequency data updates such as stock markets, sensor applications, and pervasive environments. Also these applications produce data continuously. In other words, they produce data 24 hours a day and 7 days a week and the data is typically presented in the form of a *data stream*. As a result, the volume of a data stream is huge. On the other hand, these applications need sophisticated processing capability for continuously monitoring incoming data and finding interesting changes or patterns over the data in a timely manner. An example is to find the outliers from credit card usage transaction logs, rather than sophisticated transaction processing provided by DBMSs. These applications are different from DBMS applications in terms of their data sources and computation requirements.

It is clear that these applications do not fit the traditional DBMS data model and its querying paradigm since DBMSs are not designed to load high-frequency updates in the form of data streams and to provide continuous computation, expressed as *continuous queries*, over stream data. The techniques developed in DBMSs can not be directly applied to these applications. New data models and data processing techniques are required to match the requirements of an increasing number of stream-based applications. The systems that are used to process data streams and provide needs of those steam-based applications are termed Data Stream Management Systems (DSMSs).

1.1 Data Streams

A data stream is defined as an infinite sequence of data items that are usually ordered by their arrival time stamp explicitly or by their attributes (e.g., packet sequence identifier in an IP session).

1.1.1 Data Stream Applications

Many applications from different domains generate data streams and need data stream processing. In the following, we give a few examples of such applications:

Finance applications: Stock prices, cash flows, credit card transactions, among others, are all presented in a form of data streams. The online analysis over these streams includes discovering correlations, identifying trends and outliers (fraud detection), forecasting future values, etc.

Computer network management applications: The SNMP (simple network management protocol) data, the routing table information (BGP table information), the network traffic information are representative streams in the network field. All these data arrive rapidly and are usually unbounded in size. Traffic engineering and network security [4] are two representative applications of data stream processing systems in computer network management field.

Telecommunication management applications: The call-detail record (CDR) information, and various network management messages from network elements, such as alarm message, performance message, and so on, fall into the category of streaming data. The online billing system requires processing CDR information in real-time in order to generate billing information on the fly. The universal fault management system is required to analyze alarm messages from various sources, such as transport networks (SDH, SONET), switch networks (5ESS,DMS100), signaling systems (SS7), and intelligent network systems (800 service and other value-added systems), in order to locate the primary causes of various faults in real-time.

Homeland security applications: The security information from various sensors (e.g., scanners, cameras) at an airport checking point is presented in the form of data streams, which is further used to detect abnormal behavior through analysis of incoming information that is correlated with the information from other sources.

Sensor applications: Sensor monitoring [5][76][32] is another large group of applications of data stream processing system, which are used to monitor various events and conditions through complex filtering and joining sensor streams. For example, the highway traffic monitoring and querying, the smart house at UTA, etc..

On-line applications: Online bank systems generate transaction streams which need to be analyzed immediately to identify potential fraud transactions. Online auction systems such as Ebay [2] generate real-time bid streams and the systems need to update the current bid price and make decisions in a real-time manner. Large web systems such as Yahoo [7] and search engines such as Google [3] generate numerous webclicks and user-query data streams and they need to analyze the streams to enable applications such as personalization, load balance, advertising, and so on on the fly.

Others: Health care applications, such as patient's condition monitoring. Applications based on Global Positioning Systems (GPS), Supply-Chain applications from large retailers such as Wall-Mart that use RFID (radio frequency identification) tags, and others.

1.1.2 Data Stream Characteristics

Actual data streams (number of attributes, value ranges etc.) generated by different applications are different from each other. However, these streams share a set of characteristics that should be taken into consideration when designing a data stream management system. Those characteristics are summarized below:

- Data items arrive continuously and sequentially as a steam. They are usually ordered by time or some other attribute(s). Therefore, the data items from a data stream can only be accessed sequentially.
- Data streams are usually generated by external applications and they are sent to DSMSs which can be either centralized or distributed or P2P systems, over networks. The DSMSs do not have direct access or control over those data stream sources.
- The amount of data in a data stream is very large. For example, the AT&T CDR records are about 1 giga bytes per hour [69] and an OC-768 at Sprint can carry traffic data at speeds as high as 40 gigabits per second. Theoretically, the size of a data stream is potentially unbounded. Therefore, those data streams cannot be stored first and then processed. Processing requirements does not permit storage first and processing next. However, the stream (or parts of it) may be stored after

processing for archival or other purposes. Also the nature of the input does not allow one to make multiple passes over a data stream while processing.

- The input characteristics of a data stream are usually not controllable and unpredictable. The input rate of a data stream ranges from a few bytes per second, for example the readings of a sensor, to a few gigabits per second such as traffic data over an OC-768 channel and it is not controllable. Also the input patterns of a data stream are irregular and are usually bursty in nature. For example, a number of data streams with a so-called self-similar property as reported from compute network research are highly bursty. These streams include the local Ethernet traffic streams [86], the HTTP traffic flows [43], and Email-messages.
- The data type in data streams varies from well-structured data streams such as temporary readings to semi-structured message streams such as HTTP log streams, message streams from a circuit switch, and complex XML document streams to unstructured streams such as Email streams, document streams, and so on.
- The data items from data streams are not error-free.

The above characteristics of data streams are very different from those assumed for relations processed by traditional DBMSs and as a result they pose many new challenges in the design of DSMSs as we will discuss in 1.1.5. It is also clear that DSMSs are proposed for a new category of data generated by applications that are not considered (at least not fully considered) by DBMSs.

1.1.3 Data Stream Application Characteristics

Many applications, as discussed earlier in 1.1.1, need to process data streams and they have different requirements for an underling DSMS. Fortunately, these requirements can be categorized into a set of abstractions that need to be supported by a DSMS just like that the requirements of DBMS-based applications have been abstracted into a set of requirements to be supported by a DBMS.

- Continuous processing of newly arrived data is important for all stream-based applications. This requires a query to persist in the system as long as its input data stream(s) is not terminated.
- Most stream-based applications can deal with approximate results as long as other critical (e.g., real-time processing) requirements are satisfied. Approximate results are acceptable when accurate results are not available. For example, an Internet Search Engine Provider such as Google wants to personalized the result-pages based on the processed query, such as inserting related advertisements or recommendations and so on. For each user query from user query streams, a DSMS computes its relevance to all active advertisements in DBMSs and then inserts the top 5 advertisements into the query-response pages and returns to the user. During high-load periods, the system can not find the exact top 5 advertisements due to the limited resources and it is acceptable to provide 5 most-related advertisements.
- Timely processing of data is another critical requirement as many stream-based applications need to respond to pre-defined or abnormal events in a real-time manner.
- Many stream applications have specific Quality of Service (QoS) requirements. Common QoS requirements include response time (also termed tuple latency), precision, throughput, and so on and there exits a trade-off among these QoS metrics for different applications.
- Complex event processing and rule processing is another important requirement of many stream-based applications in order to detect events/conditions and fire triggers in a timely manner when abnormal or user-defined events are detected. For example, through analyzing the temperature readings of sensors in a smart-

house, a DSMS needs to take a sequence of actions such as dial-911 for a fire alarm once it detects a sharp increase of the temperature in one room.

it is not feasible to support the above requirements for stream-based applications by simply loading the data stream into a DBMS and using the state-of-the-art DBMSs. A DBMS may be useful for providing persistent data management and transaction support for some stream-based applications. However, it can not provide continuous results. It has almost no support for QoS requirements and it always assumes that accurate results are required by applications. Therefore, a new stream processing system is needed to support the new requirements from a large number of newly emerging stream-based applications.

1.1.4 Data Stream Model

The characteristics of data streams presented in Section 1.1.2 clearly indicate that the data model used in traditional DBMSs is not a good fit for streaming data. A new model, namely data stream model which is considered as a more suitable model for streaming data, is used for processing data streams.

In the data stream model,

1. Data items arrive in the form of a data stream, which is considered as a sequence of data items presented in a form of $< t, v >$, where t is the time stamp at which the data item is generated and v is the actual data which can be as simple as any primitive data or as complex as semi-structured messages or unstructured documents. A newly generated data item is appended to the existing sequence of data items from the same data source.
2. The data items can only be accessed in a sequential order as they are generated and only the data items that have been seen can be accessed. Moreover, data items from data streams can only be accessed as few times as possible due to the cost

associated with accessing an item multiple times. Normally, each data item can be accessed only once.

3. The data sources that actually generate data streams are usually external to DSMS and they are not controlled by the DSMS. The characteristics of a data stream is unpredictable and uncontrollable.
4. The size of a data stream is infinite (or unbounded).
5. The arrival pattern of a sequence of data items from a data stream can be irregular and bursty.
6. The items from a single data stream arrive based on their time stamps and no out of order arrival is allowed in a single data stream. All data streams use a global timer.

The data stream model is widely accepted and used when studying data stream processing. A DSMS based on this model needs to compute the results in real-time as new items arrive and in a continuously manner. The resources such as CPU cycles and main memory size required by the model to compute functions over streams are limited and usually inadequate as compared with the size of the sequence of data items seen so far. All research results in this thesis are based on the data stream model and assumes that judicial use of resources is needed for satisfying the QoS and other requirements.

QoS is important to the success of a DSMS. A fundamental QoS problem is how to efficiently and effectively deliver pre-defined QoS requirements in a DSMS. In this thesis, we exploited the following QoS delivery mechanisms: *scheduling strategy* and *load shedding*. We also investigate the problem of QoS verification problem, which is: "*how do we know the query results really satisfy predefined QoS metrics?"*. Finally, we discuss the integration of event and rule processing into a data stream processing system.

1.1.5 Research Challenges and Problems

The characteristics of data streams and the newly emerging requirements of stream-based applications pose many challenges in processing data streams [18], ranging from theoretical study of data stream algorithms to query specification and computation expressiveness by extending SQL (structured query language) and to various components in a full-fledged DSMS. However, we limit the problems to a few components of a DSMS in this thesis our focus is on QoS related issues of a general DSMS. Specifically, we discuss the following problems in this thesis in the context of a DSMS.

System Capacity Planning and QoS Verification: The problem of system capacity planning in DSMSs is described as follows: given a set of continuous queries with their QoS specifications over a set of data streams, what kind of resources such as CPU cycles, memory size and others do we need to compute the given continuous queries and satisfy their QoS requirements. The reverse problem is equally important. Given a system in terms of resources and a set of continuous queries with their QoS specifications over data streams, does the system really be able to satisfy specified QoS requirements of continuous queries. In other words, we would like to verify whether the final query results really satisfy their pre-defined QoS requirements. To the best of our knowledge, there is no solution for system capacity planning for DSMSs and the currently proposed solution to the QoS verification problem is to continuously monitor the final query results and verify them directly. Sampling techniques can also be used to reduce the cost. In order to further decrease the cost and only verifying violations only at runtime, we exploit the problem of how to predict the QoS metrics of final query results by using queuing theory. Specifically, we propose solutions to the problems of predicating the tuple latency of final query results in a DSMS and main memory required to process the queries in a DSMS through modeling the query processing system as a network of queuing systems.

Scheduling Strategies: A DSMS needs to compute functions over data streams in real time and in a continuous manner. The problem of resource allocation, also termed scheduling strategy, arise in a multiple continuous query processing system when multiple items from input streams arrive. Namely, we need a mechanism to determine which query or which operator of a query should compute over the newly arrival data items first. In a multiple query processing system, different queries have different QoS requirements. Some of them favor a real-time response time. Some of them prefer accurate results. Others may need both of them or neither of them. From a query QoS requirement prospective, a DSMS can allocate resources by employing a scheduling strategy based on their QoS requirements. However, the problem is more complicated than it appears because: *1*) a DSMS may have a limited amount of resources (i.e., CPU cycles and Main Memory size); *2*) different objects (queries or operators) have different CPU cycle requirement and different memory release capacity, and *3*) the input pattern of a data stream is irregular and highly bursty. Therefore, scheduling a different object to process one tuple requires different CPU cycles and releases different amount of memory. This means a scheduling strategy has significant impact on various aspects of performance of a DSMS and it also poses different strategies to maximize the available physical resources to handle bursty nature of input data streams.

For example, suppose there are two operators A and B in the system and it needs 1 second to process 1 tuple with a size of 2 bytes at operator A and output 1 tuple with a size of 1 byte (processing rate 1 tuple/second, memory release rate 1 byte/second). However, it needs 2 seconds to process 1 tuple with a size of 2 bytes at operator B and output a tuple with a size of 1 byte as well (processing rate 0.5tuple/second and memory release rate 0.5 byte/second). If input streams of both operator A and B have an input of 1 tuple per second for 5 seconds and then a pause, a scheduling strategy which schedules A first if there is any tuple waiting at A, then schedules B, requires a maximal memory of 10 bytes (or 5 tuples at B) which are waiting at B right after the bursty input period. Another strategy which gives B a higher priority than A requires a maximal memory of 15 bytes (5 tuples at A and 2.5 tuples at B) right after the bursty input period. If the system only has 12 bytes memory in total, the second scheduling strategy definitely causes the system to crash. Similarly, a scheduling strategy has a different impact on overall tuple latency and throughput of a DSMS.

Load Shedding: The scheduling strategies discussed above can be used to allocate resources carefully to satisfy the requirements of different queries in a DSMS given sufficient total amount of resources. However, a DSMS may still be short of resources in order to process all registered queries during temporary overload periods due to the highly bursty input patterns in the input data streams. Under this case, it is infeasible for a scheduling strategy, no matter how good (or optimal) a scheduling strategy is, to satisfy all QoS requirements of all registered queries in the system. A natural solution to this problem is to gradually discard some unprocessed or partially processed tuples from the system. It is worth noting that discarded tuples also degrade the quality of final query results. However, recall from the characteristics of stream-based applications, many stream-based applications can tolerate approximate results. The process of gradually discarding some tuples from a DSMS with a goal of minimizing the errors introduced towards the final results is termed *load shedding*. The load shedding process is necessary and important in a DSMS to deal with the bursty nature of its input data streams.

One issue that arises during load shedding is the choice of tuples to discard and how much to discard. Another issues is when to discard and when to stop discarding. We will discuss details of the problem and our solutions to the problem in Chapter 5.

Event and Rule Processing: Many applications need DSMSs to processing streaming data. They also need DSMSs to actively find interesting events and useful patterns by processing streams and trigger a sequence of pre-defined actions once interesting events are detected. The current DSMSs focus on the traditional computations provided in DBMSs through a set of predefined operators such as *SELECT*, *JOIN*, *PROJECTION*, *AGGREGATITION*, and so on. It has little support to express complex events and the computation needed for detecting complex events and rule processing. In contrast, event processing in the form of ECA (event-condition-action) rules has been researched extensively from the situation monitoring viewpoint to detect changes in a timely manner and to take appropriate actions. Several event specification languages and processing models have been developed, analyzed, and implemented. Researchers seem to address them as two separate topics. The problems we address is: *1*) how similar and different are these two threads of work and the two models, namely data stream processing model and event processing model? *2*) will an integrated model that synthesizes these two by combining their strengths do a better job than either one of them? *3*) what extensions do we need in a DSMS in order to synthesize an integrated model?

We investigated all above issues in this thesis.

1.2 Summary of Our Contributions

Our main work in this thesis targets building a general DSMS prototype, termed MavStream, as part of the MavHome Project [42, 5] at the University of Texas at Arlington. However, the solutions and algorithms proposed in the thesis are not particular for this project and can be applied to other DSMSs as well.

Continuous Query Modeling: System capacity planning and QoS verification are two important problems in DSMSs to support QoS requirements of continuous queries. System capacity planning has not been studied traditionally in the context of database systems and we have not seen, to the best of our knowledge, any system capacity planning study in DSMSs. The QoS verification is simply done through monitoring output results and verifying QoS individually. To address both problems, we propose a queueing model to analyze QoS metrics such as tuple latency and memory requirement in DSMSs using queuing theory in this thesis. We first study the run-time characteristics of each operator in a continuous query plan by modeling each operator in a select-project-join query as a stand alone M/G/1 queuing system. We study the average number of tuples, average tuple latency in the queue, and the distribution of the number of tuples and tuple latency in the queue under the Poisson arrival of input data streams in our queueing model. We then extend our queuing model to a multiple continuous query processing system. Each operator in the system is modeled as an M/G/1 queuing system with vacation time and setup time, where the vacation time of an operator is the period of time during which it does not gain CPU cycles, and the setup time is the initialization time of an operator before it starts processing the input tuples once it gains the CPU cycles. The whole query processing system is modeled as a network of queuing systems with vacation time and setup time. Under this queuing model, we analyze both memory requirement and tuple latency of tuples in the system under two service disciplines: gated-service discipline and exhausted service discipline. Based on our queueing model, we can answer the question of whether a system in terms of resources can compute a set of given continuous queries with QoS requirements over given data streams. On the other hand, the memory requirement estimation provides useful guideline for the allocation of buffer for each operator effectively and in addition provides useful information for scheduling as well. The estimation of tuple latency provides critical information about when the outputs of the system violate their predefined QoS requirements, which implicitly determines when the system has to activate appropriate QoS delivery mechanisms to guarantee the predefined QoS requirements.

Scheduling Strategies: In a DSMS, different scheduling strategies can be employed under different scenarios (i.e., light load periods, heavy load periods) to satisfy the resource requirements of different queries. it is a challenge to effectively allocate the limited resources (CPU cycles, Memory, and so on) to the queries at run-time. In this thesis, we first present our scheduling model and then propose a family of scheduling strategies for a DSMS. Specifically, the operator path capacity (PC) scheduling strategy schedules the operator path with the biggest processing capacity at any time slot. we prove that the PC strategy can achieve the overall minimal tuple latency. In order to decrease its memory requirement, we further propose the segment strategy in which the segment with the biggest processing capacity is scheduled. Due to the larger processing capacity of the bottom operators in a query plan, the segment can buffer its partially processed tuples in the middle of an operator path. The segment strategy greatly improves the memory requirement with a slightly larger tuple latency. The memory optimal segment strategy (MOS) employs the same segment partition algorithm as the Chain strategy [21] proposed in the literature, but minimizes the memory requirement. However, it schedules an operator segment each time, instead of an operator each time as in the Chain strategy. As a result, it achieves the strictly optimal memory requirement comparing with the near-optimal memory requirement in the Chain. It also achieves better tuple latency and smoother throughput than the Chain strategy. The simplified segment strategy is a variant of the MOS strategy as it partitions an operator path into at most two segments, which preserves the low memory requirement of the MOS strategy and further improves its tuple latency. Finally, the threshold strategy is proposed, which is the combination of the PC strategy and the MOS strategy. It activates the PC strategy to achieve the minimal tuple latency if the memory is not a concern; otherwise, it acts as the MOS strategy in order to decrease the memory requirement. We further discuss how to extend these strategies to a multiple query processing system in which computation sharing through sharing common sub-expressions is essential. All proposed strategies work under a general multiple-CQ DSMS with computation sharing. The techniques of extending our strategies to a multiple CQ processing system can also be applied to extend other strategies that only work under a DSMS without computation sharing such as the Chain strategy and others. The theoretical results are validated by an implementation where we perform experiments on all the strategies. The experimental results clearly validate our theoretical conclusions. Furthermore, the threshold strategy inherits the properties of both the path capacity strategy and the simplified segment capacity strategy, which makes it more appropriate for a DSMS.

Load Shedding: In order to deal with high bursty nature of input data streams of a DSMS, we develop a framework and techniques for a general load shedding strategy by dynamically inserting load shedders into query plans and activating/deactivating existing shedders based on the estimation of current system load. These shedders can drop tuples in either a randomized manner or using user-specified application semantics. Specifically, we first address the problem of the physical implementation of a load shedder with a goal of minimizing the overhead introduced by the shedder itself. Our analysis and experiments show that a shedder should be part of the input queue of an operator in order to minimize the load introduced by the shedder itself and to decrease the memory requirement by discarding tuples before they enter the input queue. We then develop the solution for the problem of predicting the system load through monitoring the input rates of the input streams. The estimated system load implicitly determines when load shedding is needed and how much. The proposed system load estimation technique can be applied to the other load shedding techniques proposed in the literature as well. We also develop algorithms to determine the optimal placement of a shedder in a query plan in order to maximize load shedding and to minimize the error introduced by discarded tuples during high-input periods. Once we determine how much load needs to be shed and where, we propose a strategy for distributing the total number of tuples to be dropped among all shedders with a goal of minimizing the total relative errors in the final query results due to load shedding. Finally, we conduct extensive experiments to validate the effectiveness and the efficiency of the proposed load shedding techniques.

Event and Rule Processing: In order to extend the expressiveness and computation power of a DSMS to process massive complex events and rules, we develop an integrated model with several enhancements to data stream processing model to synthesize the data stream processing and event-processing model. Specifically, we first analyze the similarities and differences between the event and stream processing models and we clearly show that although each one is useful in its own right, their combined expressiveness and computation power is critical for many applications of stream processing, and there is a need for synthesizing the two into a more expressive and powerful model that combines the strengths of each one. We then provide several enhancement to the data stream processing model to support the complex event computation and rule processing over data streams. Those enhancement include:

1) the ability to name CQs,

2) stream modifiers,

3) semantic windows,

4) extended event operators with input queues/buffers,

5) enhanced event expressions,

6) enhanced event consumption modes (or contexts), and

7) extended SQL to support combined specification of events and CQs.

Through the above techniques and enhancements, our integrated model not only supports a larger class of applications, but also provides more accurate and efficient ways for processing CQs and event expressions. All the enhancements proposed for our integrated model do not affect any current stream processing techniques and can be easily integrated into any current data stream management systems. Finally, we discuss the implementation of the integrated model using the MavStream system and the Local Event Detector of Sentinel.

1.3 Thesis Organization

The rest of thesis is organized as follows. We begin by providing an overview of data stream processing and the architecture of proposed QoS driven DSMS and the related work in Chapter 2. We then present our queueing model and how to perform system capacity planning and verify QoS requirements of query results in Chapter 3. We present our proposed scheduling strategies along with detailed theoretical and experimental analysis in Chapter 4. In Chapter 5, we present a set of comprehensive load shedding techniques to handle bursty nature of input data streams of a DSMS. In Chapter 6, we analyze both data stream processing and event processing model and discuss the enhancements to a general DSMS in order to synthesize two models to process massive complex event expressions and rules. Finally, we conclude the thesis by summarize our contributions with a discussion of open research problems in Chapter 7.

CHAPTER 2

DATA STREAM PROCESSING

The computation over data streams is expressed in the form of continuous queries compared with one-time ad-hoc queries in DBMSs. This is due to two major reasons: *1*) queries expressed in SQL in traditional DBMSs have gained much success and been accepted widely. It is a natural solution to use SQL with some extensions to express continuous queries, and *2*) DSMSs have to compute results in a continuous manner and output results in real-time. The ad-hoc queries cannot output results in a continuous manner and hence continuous queries are natural choices. In this chapter, we will first give an overview of continuous query processing and then present the architecture of the proposed QoS-driven DSMS. Finally, we will discuss related work in data stream processing and provide a summary of this chapter.

2.1 Overview of Data Stream Processing

2.1.1 Continuous Query

Continuous queries are modified/extended versions of one-time ad-hoc queries used in traditional DBMSs. A continuous query plan consists of a set of operators, such as *project, select, join*, and other aggregate operators, as one time ad-hoc query plans in DBMSs. However, the operators in DSMSs are different from those in DBMSs in the following aspects:

1. All operators in DSMSs have to compute in a streaming manner or in a *non-blocking* mode. Many operators such as *JOIN*, *SORT*, and some aggregation operators in DBMSs are blocking operators. They are blocking operators in the sense that they

need to have all their inputs before they can output results. However, in the data stream model presented in 1.1.4, it is impossible for operators in DSMSs to have all their inputs before computing their functions since the size of a data stream is assumed to be unbounded. As a consequence, these blocking operators have to be modified to compute in a window-based manner by imposing a window over each input data stream, such as a window-based *symmetric hash join* [122].

2. All operators are computed in a push-paradigm as compared with the pull-paradigm in DBMSs. In traditional DBMSs, a query plan is processed by computing its top operator by obtaining one tuple from each of its child operators. Each of its child operators recursively calls their child operators to get required tuples in order to output one tuple to its parent. However, in DSMSs, the input characteristics of a data source (i.e., an input data stream) are not controllable. If the operators were computed with the pull-paradigm, the operators would be blocked if there was no input from one of its child operator temporarily. On the other hand, a child operator may need to buffer a large number of tuples during bursty input periods as its parent operators do not have any information about the input characteristics of an input stream. To facilitate data stream processing, a push paradigm is used and the tuples are pushed from bottom operators gradually to the top operators.
3. All operators are usually associated with a queue, which is used to store unprocessed inputs and to deal with the bursty nature of stream inputs. As a result, the output of one operator is directly output to the input queue of another operator.

As a result, a continuous query plan in DSMSs consists of a set of basic non-blocking operators and the inter-queues that connect them and a continuous query processing system over data streams can be conceptualized as a data flow diagram. In this diagram, a node represents a non-blocking operator. While a directed edge between two nodes represents the queue (or buffer) connecting those two operators, it also determines the

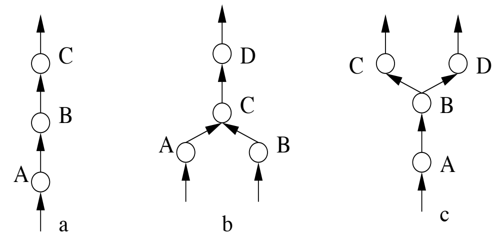

Figure 2.1 Operator Path.

input and output relationship between those two operators. For example, the edge from node $\mathcal{A}$ to node $\mathcal{B}$ indicates that the outputs of operator $\mathcal{A}$ are buffered into the queue $\mathcal{AB}$ that is the input source of operator $\mathcal{B}$. Each source (input stream) is represented as a special source node, while the applications are represented as a root node in this data flow diagram. Therefore, the edges originating from source nodes represent the earliest input queues that buffer the external inputs; while the edges terminating at the root node represent final output queues that buffer final query results. In this diagram, each tuple originates from a source node, and then passes through a series of operators until it reaches the root node or is consumed by an intermediate operator. We refer to the path(s) that a tuple travels from a source node to the root node excluding the source node and the root node as an *operator path*(OP), and the bottom node of an OP as its *leaf node*.

The operator paths can be further classified into two classes in a multiple query processing system as illustrated in Figure 2.1:

1. simple operator path, if there is no sharing among multiple queries as in Figure 2.1-a and2.1-b (one simple path ABC in Figure 2.1-a and two simple paths ACD and BCD in Figure 2.1-b);
2. complex operator path, if two operators share the outputs of one operator as illustrated in Figure 2.1-c (one complex path: ABC/D).

An *operator segment* is defined as a set of connected operators along an operator path. Similarly, there are simple operator segments and complex operator segments in multiple query processing systems. Detailed algorithms are introduced in Section 4.3.3 to partition an operator path into operator segments based on different criteria.

The operators also maintain statistical information such as selectivity, service time[1], and others as we maintain statistical information in a catalog of a DBMS. This information is important and necessary for QoS delivery (i.e., scheduling, load shedding, and others) and query optimization. This information is updated periodically during run-time. The selectivity of an operator is maintained as a function (i.e., moving average) of the old value and the current ratio of the number of input tuples and the number of output tuples during a schedule and the service time of an operator is maintained as a function of the old value and the current ratio of the total service time to the total number of tuples during a batch process (i.e., a schedule run). The overhead of maintaining those statistic information is small because we only update them periodically.

2.1.2 Window Specification

As we mentioned earlier, blocking operators have to be modified to compute in a non-blocking mode by imposing a window. A window is defined as a historical snapshot of a finite portion of a stream at any time point. This window defines the meaningful set of data used by operators to compute their functions. Currently, there are two types of windows are used: tuple-based and time-based. In this thesis, we also introduce the notion of a semantic window [72], which will be discussed later in the chapter 6.

A Time-based window can be simply expressed by ***[Range N time units]*** and a Tuple-based window can be expressed by ***[Row N tuples]***, where N specifies the length of the window.

[1]The CPU time needed for an operator to process one tuple.

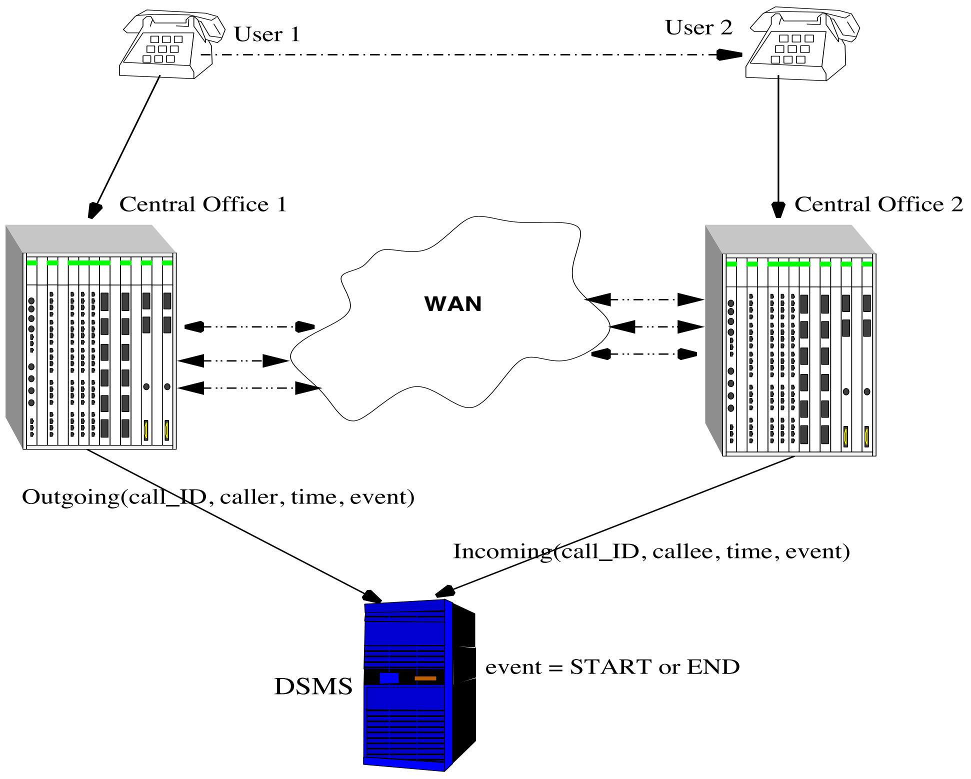
User 1
User 2
Central Office 1
Central Office 2
WAN
Outgoing(call_ID, caller, time, event)
Incoming(call_ID, callee, time, event)
DSMS
event = START or END

Figure 2.2 Data Streams In a Telecommunication Service Provider.

2.1.3 Examples of Continuous Queries

Consider the Call-Detail-Record (CDR) streams from circuit switches of a large telecommunication service provider illustrated in Figure 2.2. CDR records are collected for each outgoing or incoming call at a central office. Four fields, which are *call_ID*, *caller* for outgoing call or *callee* for incoming call, *time*, and *event*, which can be either *START* or *END*, are extracted from each CDR and sent to a DSMS. Various online processing over CDR streams can be done there. The following are three continuous queries over *Outgoing* and *Incoming* streams.

Consider the first continuous query Q_1, which finds all the *outgoing calls* longer than 10 minutes over a 24-hour window (assume that no call lasts longer than 24 hours).

```
Q1:     SELECT   O1.Call_ID, O1.caller
        FROM     Outgoing O1 [Range 24 hours] , Outgoing O2
[Range 24 hours]
        WHERE    (O2.time - O1.time > 10
        AND      O1.call_ID == O2.call_ID
        AND      O1.event == START
        AND      O2.event == END)
```

This query continuously outputs all outgoing calls longer than 10 minutes originating from central office 1 and terminating at central office 2 within a 24-hour time sliding window. The results form a new data stream because of the continuous output of result records.

The second query is a join query, which finds all pairs of caller and callee between two central offices over a 24-hour window.

```
Q2:     SELECT   O.Caller_ID, I.callee
        FROM     Outgoing O [Range 24 hours] , Incoming I
[Range 24 hours]
        WHERE    (O.call_ID == I.call_ID
        AND      O1.event == START
        AND      O2.event == START)
```

The results from Q_2 form a data stream too and the join is done using a 24-hour sliding window on each stream.

Our final example query is an aggregation query, which computes total connection time of each call between two central offices.

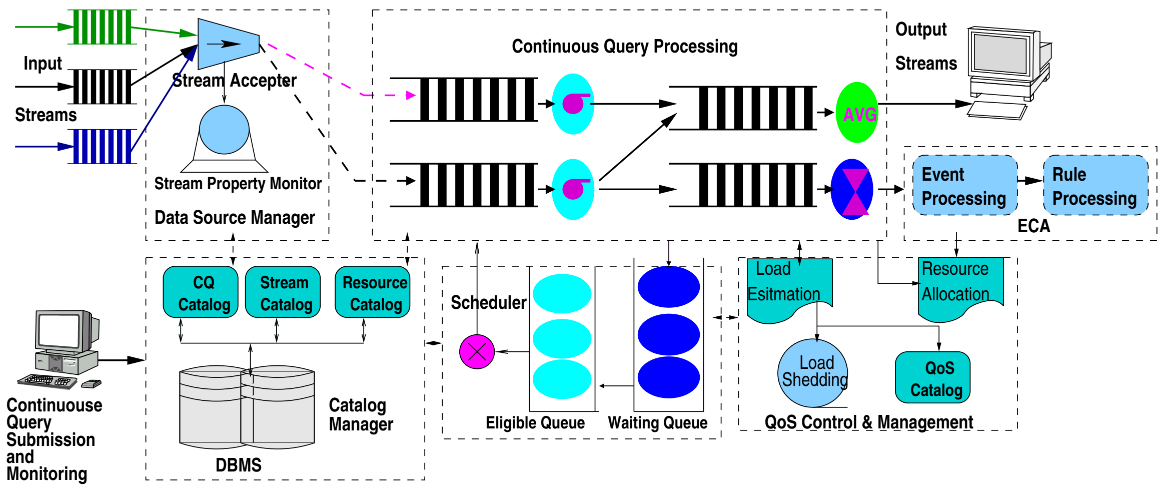

Figure 2.3 Architecture of Proposed QoS-Aware DSMS.

```
Q_3:   SELECT   O_1.Call_ID, SUM(O_2.time - O_1.time)
       FROM     Outgoing O_1, Outgoing O_2
       WHERE    (O_1.call_ID == O_2.call_ID
       AND      O_1.event == START
       AND      O_2.event == END)
```

The results of this query form a data stream too. However, only the most recent result is correct and useful. The application has to update the previous value using the most recent result. The results should be presented to the user in an update-only manner, instead of an append-only manner.

2.2 Architecture of Proposed Data Stream Processing System

Figure 2.3 shows the system architecture of our proposed QoS-aware DSMS. The system consists of six components: `data source manager`, `query processing engine`, `catalog manager`, `scheduler`, `QoS manager`, and `ECA manager`. The `data source manager` accepts continuous data streams and inserts input tuples into corresponding input queues of query plans. It also monitors various input characteristics of a stream and data stream characteristics (i.e., sortedness). Those characteristics provide useful information for query optimization, query scheduling, and QoS management. The `query processing engine` is in charge of generating query plans and optimizing them dynamically. It supports both CQs and one-time ad-hoc queries. The `catalog manager` stores and manages the meta data in the system, including stream meta data, detailed query plans, and resource information. The `scheduler` determines which query or operator to execute at any time slot due to the continuous nature of the queries in the system. Due to the fact that most of stream-based applications have different QoS requirements, the `QoS manager` employs various QoS delivery mechanisms (i.e., load shedding, admission control, and so on) to guarantee the QoS requirements of various queries. In many stream-based applications monitoring changes through continuous queries is necessary and important in order to capture and understand the physical environment. However, it is equally important to react to those changes immediately. The `ECA manager` in our system is used to detect complex events and to respond to those events using predefined actions. We discuss the details of each of these components in the proposed QoS-aware DSMS system in the remaining of this thesis.

2.3 Related Work

In this subsection, we will discuss related work. We will first provide a brief review of data stream processing systems and then related work in areas of continuous query modeling, scheduling strategies, load shedding, and event and rule processing.

2.3.1 Data Stream Processing Systems

Tapestry [115, 61] was an experimental content-based filtering system over email and message streams developed at the Xerox Palo Alto Research Center. It translates filters specified by users based on their preferences and other personalization information into continuous queries internally and processes those continuous queries over an appended-only database. *Alert* system used continuous queries, also called *active queries* in [105], over *active tables*, which are append-only tables, to transform a passive DBMS into an active DBMS without using triggers. It executes active queries over active tables using a cursor with an enhanced fetch operation to keep track of new tuples appended to active tables and the active queries continuously compute their functions over newly returned tuples by the cursor from the active tables. Continuous queries were also used in *Tribeca* [110] to monitor and analyze network traffic streams by using a data-flow oriented query language.

OpenCQ [88], a distributed event-driven continual query system, used incremental materialized-views to support continuous queries processing for monitoring persistent data streams from a wide-area network. Niagara system [41] supports continuous queries over large scale of streams through grouping queries based on their signatures to sharing computation efficiently.

STREAM [6] is a general-purpose data stream management system developed at Stanford University that expresses its queries in a declarative query language similar to SQL. The research group of STREAM system proposed the Chain scheduling strategy [21] that aims to minimize total memory requirement and a load shedding mechanism [30] for aggregate queries. The work of STREAM also includes resource sharing [16, 99], query caching [22], and others [13, 100, 23]. The PC strategy proposed in this thesis aims to minimize the overall tuple latency as compared with the Chain strategy which aims to minimize total memory requirement. The MOS strategy proposed in this thesis strictly minimize the total memory requirement as compared with the near-optimal memory requirement of the Chain strategy. Furthermore, we propose a simplified segment strategy and a threshold strategy which are better-suited for a DSMS. The optimal properties of the PC and the MOS strategy are proved in the context of a multiple-CQ processing system as compared with the proof of the memory optimal property of the Chain strategy within a restricted DSMSs where no sharing computation is allowed.

Aurora [32] proposed a framework for the support of monitoring applications over data streams. Continuous queries in Aurora form a data-flow diagram and new queries are added to the diagram through decorating the diagram. Its work focuses on real-time data processing issues, such as QoS- and memory-aware operator scheduling [33], semantic load shedding for coping with transient spikes in incoming data rates, and other issues. However, the scheduling strategies proposed do not have a theoretical analysis or proof of their properties. Its next generation system, called Borealis [1, 8], is trying to address the problems of dynamic revision of query results, dynamic query modification, and flexible and highly-scalable optimization. At the time of writing this thesis, the Borealis project is in its initial phase.

TelegraphCQ [40, 39] processing engine over data streams. It proposed an adaptive engine based on Eddy [17] in which a router is used to route a tuple to an operator and the outputs of each operator are returned to router to find their next operator. Based on different system load and other factors, the router can change the path of a tuple dynamically and each tuple from the same data source can have a different path.

There are also a group of data stream systems that are built exclusively for sensor networks and applications such as TinyDB [91, 90], Cougar [28, 124]. Those systems have a different focus from our proposed system and other systems discussed above, where a central query processing system is proposed, and they mainly focus on network-related query processing issues such as reducing communication costs, minimizing power consumption, and so on.

MavStream [76, 59, 107], proposed in this thesis, aims to exploit general solutions for QoS related issues and complicated event and rule processing. To the best of our knowledge, the system capacity planning issue and QoS verification issue are first studied and their solutions are developed and discussed in this thesis. Also, the event and rule processing under the context of data stream processing is first studied as well. We propose scheduling strategies and load shedding techniques which aim at efficient QoS delivery in more general DSMSs as compared to the proposed solutions in the literature. The detailed contrast and comparison of the solutions proposed in this thesis with other solutions in the literature are studied and discussed in detail in following subsections.

2.3.2 Continuous Query Modelling

The inputs of a DSMS are highly bursty data streams. This bursty nature impacts the load of a DSMS dramatically. On the other hand, QoS requirements of continuous queries in the system need query results to satisfy those QoS requirements even in the presence of bursty input. This raises two important problems in a DSMS: *1*) system capacity planning. Given a set of continuous queries with their computation and QoS requirements and a set of input data streams, what kind of system (in terms of resources such as CPU cycles, Main Memory, and so on) is needed to compute the queries over the given data streams? In other words, does the given system have enough capacity to support the queries over given data streams? *2*) system capacity and QoS metrics estimation. Given the continuous queries with QoS requirements, the input data streams, and the systems in terms of resources, can we approximately estimate the possible QoS metrics for final query results output by the given system? The estimated QoS metrics provide important information for a DSMS to take corresponding actions including changing scheduling strategies, activate load shedding mechanisms, and others based on the difference between the estimated QoS metrics and required QoS metrics.

System capacity planning [94, 93] and QoS metrics estimation have been extensively studied under the context of the artificial intelligence and computer and system performance evaluation respectively. However, there is little work done in the context of database research mainly because there is little need and little support of QoS requirements in DBMSs. The highly bursty input-patterns and QoS requirements in DSMSs make system capacity planning and QoS metrics necessary and important for DSMSs to support QoS requirements. To the best of our knowledge, our work in modeling continuous query is the first work in system capacity planning and QoS metrics estimation using queueing theory under the data stream processing context.

Although there are a few papers that have studied the performance issues in DSMSs, those papers focus on the study of the various components or operators in DSMS, rather than the whole system as in our work of modeling continuous query. The performance studies of various components or operators in DSMSs include Kang et al. [79] investigated the various multi-join algorithms and studied their performance in continuous queries over unbounded data streams. Golab et al. [60] analyzed incremental, multi-way join algorithms for sliding window over data streams and developed a strategy to find a good join order heuristically. Vilgas et al [120] study how to maximize the output rate of a mulit-way join operator over stream data. In this thesis, we first use queueing theory to analyze the performance issues in stream data management theoretically. We then study the performance metrics of an operator over data streams given its input characteristics. Furthermore, we extend our analysis method to a continuous query in a multiple continuous query processing system, which enables us to estimate some fundamental QoS metrics (i.e., tuple latency, memory requirement) in a DSMS. Those QoS metrics provide critical quantitative information for us to design and plan a DSMS (for example, to estimate minimal hardware configuration and to initially allocate the memory size of an operator or a continuous query). The metrics computed also provide necessary information to help us choose various QoS delivery mechanisms, such as when to start or stop a CPU-saving scheduling strategy or a load shedding strategy.

Regarding the QoS metrics estimation, monitoring and verification in DSMS, there is little study in the literature. However, there are a few mechanisms proposed for QoS delivery. Das et al [44] study the various load shedding strategies for sliding window join operators over streaming data under limited resources. They provide the optimal offline and the best online algorithms for sliding window joins. Nesime et al. [114] propose load shedding mechanism in Aurora system [32] to relieve a system from an overloaded situation. They provide solution for determining when to do load shedding through continuously computing the CPU-cycle time required by each box (operator) or superbox in the system. Once the total CPU-cycles exceed system's capacity, the system begins load shedding. Brian et al. [20] propose a set of load shedding techniques to aggregate queries to guarantee their relative error requirements. The techniques presented in this paper can be used as an alternative approach to determine in advance when load shedding needs to be activated and can also be used as an assistant tool to enhance the system capacity estimation and load shedding activation techniques proposed in the literatures. Since it is a closed-form solution, it is a more efficient and low-cost approach as compared with currently proposed approaches in the literature. Our work is also related to a set of papers [118, 117, 122] that deal with the problems of multiple query optimizations and scheduling. In terms of system analysis, our work is also related to the memory characterization problem discussed in [12], where the authors characterize the memory requirements of a query for all possible instances of the streams theoretically, and the results provide a solid way to evaluate a query within bounded memory. We analyze both memory requirement and tuple latency from a different point of view. Furthermore, the trade-off among storage requirements, number of passes, and result accuracy under a stream data model has been studied in [67][62]. [19] considers various statistics over a sliding window under a data stream model.

Finally, our work is related to a set of papers in network domain that deal with the performance study of various queueing systems, congestion control and avoidance. In the network and telecommunication domain, the queueing models are mostly either multiple-server models [104, 84] along a traffic path from source node to destination or cyclical models within one server [112, 71] such as switching strategies (put a packet from input queue to output queue) of ATM switches. Those models are different from our task-driven vacation queueing model presented in this thesis.

2.3.3 Scheduling Strategies

Various scheduling problems have been studied extensively in the literature. The Chain strategy [29] has been proposed with a goal of minimization of the total internal queue size. As a complement to that, the PC strategy proposed in this paper minimizes the tuple latency. We also propose the Memory-Optimal-Segment (MOS) strategy to minimize the total memory requirement. The MOS strategy uses no more memory than that required by the Chain strategy and we prove that the strategy is an optimal one in terms of total memory requirement in a general DSMS. The proof of optimal-memory requirement of Chain strategy is only for a query processing system without computation sharing through sharing common-subexpression. Also the techniques that we propose to extend the MOS strategy to work under a multiple query processing environment can be applied to the Chain strategy as well. The Chain-Flush strategy introduces techniques for starvation-free and QoS satisfaction with a high cost (tuple-based scheduling). However, the techniques proposed there can be applied to all strategies proposed in this thesis as well. The Aurora project employs a two-level scheduling approach [33]: the first level handles the scheduling of superboxes which is a set of operators, and the second level decides how to schedule a box within a superbox. They have discussed their strategy to decrease the average tuple latency based on superbox traversal without providing any proofs. However, we have used a different approach and have proved that our PC strategy is optimal in terms of overall tuple latency. Although their tuple batching - termed train processing - uses a similar concept used in the Chain strategy and our segment strategies, it is unclear as to how to construct those superboxes. Furthermore, we provide a more practical scheduling strategy – termed the threshold strategy, which has the advantages of both the PC strategy and the MOS strategy. The rate-based optimization framework proposed by Viglas and Naughton [118] has the goal of maximizing the throughput of a query. However, they do not take the tuple latency and memory requirement into consideration. Earlier work related to improving the response time of a query includes the dynamic query operator scheduling of Amsaleg [11] and the Xjoin operator of Urban and Franklin [116].

Other work that are closely related to our scheduling work are adaptive query processing [17, 65, 117, 66], which address the efficient query plan execution in a dynamic environment by revising the query plan. The novel architecture proposed in Eddy [17] can efficiently handle the bursty input data, in which the scheduling work is done through a router that continuously monitors the system status. However, the large amount of the state information associated with a tuple limits its scalability and there is no proof on the optimality of the results.

Finally, the scheduling work is part of our QoS control and management framework for a DSMS and extends our earlier work of modeling CQ plan [73, 75], which addresses the effective estimation of the tuple latency and the internal queue size under a dynamic CQ processing system. Our modeling results can be used to guide a scheduling strategy to incorporate QoS requirements of applications, and to predict overload situations.

2.3.4 Load Shedding

The load shedding problem has been studied under the Aurora and STREAM project. Our work in load shedding shares some of the characteristics of the load shedding techniques proposed in [114]. However, we provide a more general and comprehensive solution in this thesis. The load shedding techniques proposed in [30] are only for aggregated continuous queries in a DSMS, rather than for general continuous queries which is the focus in this thesis.

Our work differs from them in that: *1*) to our best knowledge, we are the first one to discuss the optimal physical implementation and location of a shedder. *2*) the proposed load shedding techniques with our scheduling strategies (i.e., earliest deadline first or EDF) or QoS-aware strategies such as the Chain-Flush can guarantee the tuple latency requirement of an query. *3*) the proposed system load estimation technique provides explicitly the total load in terms of total system capacity, and it does not need to calculate the absolute system capacity as in [114]. And our system capacity estimation technique is useful for other stream processing systems to make decisions on when to activate/deactivate load shedding, and how much to shed. *4*) the number of shedders and the number of active shedders are minimized in the system. *5*) each shedder has a maximal capacity in our system, and its processing cost is minimized.

The work is also directly related to the work of various QoS delivery mechanisms such as scheduling, admission control, etc, and to the work of various QoS metrics prediction technologies. The Chain strategy [29] minimizes total memory requirements of a CQ processing system, while the PC [77] minimizes the overall tuple latency of a CQ processing system. Through those scheduling strategies, it is possible to meet QoS requirements when system load does not exceed its capacity. The work [75, 73] of predicting tuple latency of a query plan in a DSMS provides important techniques to assist a DSMS to avoid violating its predefined QoS requirements.

Finally, The work is related to architecture work of a DSMS from a system point of view. The paper provides general load shedding techniques and system capacity estimation technology for various DSMSs: Telegraph system [89], STREAM system [24], Aurora system [32], Niagara system [41], and Stream processing for sensor environments [76], to name a few here.

2.3.5 Event and Rule Processing

Event detection and rule processing [45, 37, 36, 38, 87, 56, 105, 53, 47, 50, 82, 31] have been studied extensively under the context of active database. However prior study in event and rule processing focuses on the techniques and mechanism of transforming a passive database management system to an active database management system. Most of study are conducted under the context of database management systems and one of the assumptions of those study is that the primitive events are low-frequency updates (such as data manipulation operations, function calls, events from operating systems, and so on). Also event expressions, operators and consumption modes are solely based on timestamps in earlier studies. In this thesis, our study of event and rule processing focus on complicated event and rule processing over the primitive events generated by stream processing, which are typically generated by computing various continuous queries over high-frequency updates and busty external data streams. Also our work is conducted under the data stream model, rather than a database management system as most of prior work. To the best of our knowledge, our work in complex event and rule processing is the first study under the context of data stream processing. Our work enhances both the computation and expressiveness ability of data stream management systems and has no impact on the other components or techniques proposed in data stream management systems.

DSMSs have been studied from a system architecture point of view in many papers [24, 32, 41, 89, 76, 95]. These work mainly focus on the system architecture, CQ execution (i.e., scheduling and various non-blocking join algorithms), and QoS delivery mechanisms for stream processing. The main computation over stream data is limited to the computation of relational operators over high-speed streaming data, and the event and rule processing and the extensions to CQs to enhance their expressive power and computation efficiency are rarely discussed. To the best of our knowledge, this paper is the first to support event and rule processing for stream processing model and to enhance the data stream computation model horizontally (i.e.,semantic window) and vertically (i.e., stream modifier, event and rule processing). The paper [14] proposed a CQ language for event processing, and provided extension for primarily sliding windows. The intent of this paper is not to provide a complete CQ language. Instead we propose a formal and meaningful extension to express and compute a much richer set of computations, which can be used to enhance current CQ languages without changing their syntax and the overall computation model.

Our work [72] in event and rule processing is also closely related to a set of papers [14, 125, 32] that try to enhance the expressiveness power of SQL in a data stream environment. Carlo Zaniolo et al [125] tried to enhance the expressive power of SQL over the combination of relation streams and XML streams by introducing new operators (e.g. continuous UDAs) and supporting sequences queries (e.g., to search for patterns). A. Arasu and J. Widom [14] proposed an enhanced SQL, termed CQL, to instantiate the abstract semantics and to map from streams to relations. However, the notion of semantic window proposed in this paper enhances the expressive power of SQL over streams and improve the computation efficiency through accurate definition of a window, which consequently reduces the number of tuples in the window, thereby reducing the computation requirements of window-based operators. In [32], the window concept is based on an attribute and is specified in the form of ***(size** s**, Advance** i**)***, where s is the size (in terms of values of an attribute) of the window and i is an integer or predicate that specifies how to advance the window when it slides. However, the window is still a static window (by specifying the fixed size of the window) and only one attribute can be used to define a window. The semantic window proposed in this paper can be used to define window based on meaningful information, rather than the size. The semantic window based on SQL can be static or dynamic. More important, the implementation of semantic window is straightforward by taking advantage of the existing SQL processor and its run-time overhead can be low because of the well-developed SQL optimization techniques.

Our event and rule processing work is further related to a set of papers [44, 60, 79, 108], which try to enhance the computation over streaming data. However, those papers mainly focus on relational operators, such as multi-way join algorithms, and approximated join algorithms. These enhancements are not capable of efficiently computing the changes over streaming data, which can be done efficiently through a family of stream modifiers proposed in this paper. These stream modifiers allow a stream processing system to flexibly express and efficiently monitor complicated change patterns for a large group of stream applications. A number of sensor database projects, Cougar [28, 124], TinyDB [91, 90] have also tried to integrate the event processing with query processing under a sensor database environment. However, the event-driven queries proposed in TinyDB is used to activate queries based on events from underlying operating systems. Our focus in this paper is to process large number of high volume and highly dynamic event streams from CQ processing stage for the applications that needs complex event processing and CPU-intensive computation (i.e., CQs) for generating events.

2.4 Summary

In this chapter, we first gave an overview of data stream processing system: how a stream processing system is modeled as a data-flow diagram and how window concept is used to transform blocking operators in DBMSs to non-blocking operators in DSMSs. We then presented the architecture of our proposed QoS-driven DSMS and a brief discussion of its components. Finally, we discussed the related work in the literature that are relevant to various components and techniques proposed in this thesis.

CHAPTER 3

MODELING CONTINUOUS QUERIES OVER DATA STREAMS

QoS is critical to the success of a DSMS. First, most streaming applications have to monitor various events and respond to abnormal or user-defined events in a timely manner. Second, different applications have different QoS requirements. For example, tuple latency may be critical for some applications whereas accuracy of computed results (with a bound on the error) may be important for other applications. Finally, QoS support is a key factor that distinguishes a DSMS from a traditional DBMS. If applications do not have QoS requirements for query processing, a traditional DBMS with trigger mechanism can satisfy most of them. Therefore, QoS related issues have gained much more attention in the context of stream data processing. For example, the Aurora system [32] has taken into consideration the QoS from the beginning.

There are two complementary ways to prevent a DSMS system from violating the QoS requirements of continuous queries. *1*) Given a set of continuous queries over data streams, we need to plan what resources in terms of CPU cycles, main memory, and so on, are required in order to support the set of queries. *2*) Once a query processing system is deployed, a system may experience all kinds of temporary overload periods due to the irregular and bursty input mode of data streams. However, due to the continuous computation characteristic of continuous queries, it is undesirable and even impossible to stop the production system and upgrade the system. This motivates us to develop various QoS delivery mechanisms, given a system with a set of continuous queries over input data streams, to try to satisfy QoS requirements of continuous queries.

Various QoS delivery mechanisms are being exploited and will be discussed in details in the rest of the thesis. *(1)* A scheduling strategy, which will be discussed in Chapter 4, tries to allocate more resources to those queries with more critical QoS requirements, which makes it possible to satisfy the QoS requirements of some or all queries by allocating resources carefully at run time among queries when system load is not too heavy. The path capacity scheduling strategy [77] can achieve an overall minimal tuple latency, while The MOS and the Chain scheduling strategy [29] try to minimize the total memory requirement of queries. However, when system load exceeds a certain threshold, it is impossible to guarantee some QoS requirements by only employing scheduling strategies. *(2)* Load shedding discussed in Chapter 5, which drops tuples (in either a random manner or using some semantics), is a natural choice to relieve system load and guarantee sufficient resources for scheduling strategies to satisfy all predefined QoS requirements. Since some applications have QoS requirements for both tuple latency and precision of final query results, dropping too many tuples can violate the required precision of final query results. In this case, we have to deactivate some queries and guarantee sufficient resources to deliver the QoS requirements of the other queries in the system, instead of violating the QoS requirements of all active queries. Therefore, *(3)* an admission control needs to be used to control the number of active queries in the system, and determine which victim queries have to be deactivated when there is not enough resources to deliver the QoS requirements of active queries. In a general system, all three mechanisms may have to be employed simultaneously and to work collaboratively in order to satisfy all predefined QoS requirements due to the highly dynamic changes in load experienced during stream processing.

A system has to activate different mechanisms and deactivate some of them at different levels of system load because of the highly dynamic system load caused by bursty input rate of a data stream. When system load is not too heavy, scheduling strategies are capable of satisfying QoS requirements. It is not necessary to activate load shedding [114][20] which introduces unnecessary errors in final query results, but we still need to determine which scheduling strategy can achieve better performance as system load fluctuates. When system load exceeds a certain level, both scheduling strategies and the load shedding mechanism have to be employed. When system load further increases, the admission control has to be activated as well. A fundamental problem here is when a QoS delivery mechanism has to be activated or be deactivated, and which one to activate or deactivate such that the active mechanism(s) can guarantee sufficient resource for all per-defined QoS requirements and minimize the extra-errors introduced in final query results.

This chapter focuses on these two fundamental problems related to QoS control and management. Namely: *1*) system capacity planning problem which, given a set of continuous queries with their QoS specifications over a set of data streams, addresses the kind of systems in terms of resources such as CPU cycles, memory size and others needed to compute the given continuous queries and satisfy their QoS requirements; and *2*) QoS metrics estimation problem which, given a system in terms of resources and a set of continuous queries with their QoS specifications over data streams, can estimate various QoS metrics of a continuous query in the system based on current system state (i.e., input rate of a data stream, characteristics of continuous queries). If we can estimate those QoS metrics, we can activate or deactivate corresponding QoS delivery mechanisms based on the difference the estimated QoS metrics from the required QoS parameters.

In this chapter, we develop a queueing model to study the dynamics of a DSMS given a set of continuous queries over a set of given data streams. Through our queueing model, we can estimate various QoS parameters. Through the queueing model and those estimated quantitative QoS parameters, (*1*) we are able to do system capacity planning and provisioning through the quantitative information provided by our queueing model.

That is given the input characteristics of input data streams and the computation (i.e., query plans) and the QoS requirements, we are able to estimate the kind of system, in terms of CPU cycles and the amount of memory, that we need to complete given tasks. (*2*) we are able to activate and deactivate suitable QoS delivery mechanisms based on the difference of the estimated QoS parameters from the defined QoS requirements. (*3*) we are able to verify whether a system satisfies the defined QoS requirements by comparing the estimated QoS parameters with defined QoS requirements.

For our theoretical model, we have to monitor input rate of each input stream. However, various components of DSMSs need to monitor the input characteristics of data streams, such as query optimization, scheduling, load shedding, and other purposes. Also approximate rates are enough for our queueing model. Therefore, it does not introduce any substantial cost to use our queueing model in DSMSs. DSMSs only needs to monitor the input rate of an input stream periodically and the length of the periods varies as the system load changes.

The rest of the chapter is organized as follows. Section 3.1 provides an overview of our queueing model. We formalize our problems in Section 3.2. In Section 3.3, we discuss how to model an individual operator in a DSMS, and provide tuple latency and memory requirement of an individual operator in detail. We further extend our modeling work to an individual query plan in a general DSMS, and provide closed-form solution to estimate QoS metrics (i.e., the overall tuple latency and memory requirement) of the output tuples of a query in Section 3.4. Section 3.5 further presents a set of our quantitative experimental results. The conclusion is presented in Section 3.6.

3.1 Continuous Query Processing

Query processing in a DSMS can be handled using two approaches: a multiple threads approach and a single thread approach. In a multiple threads approach, each

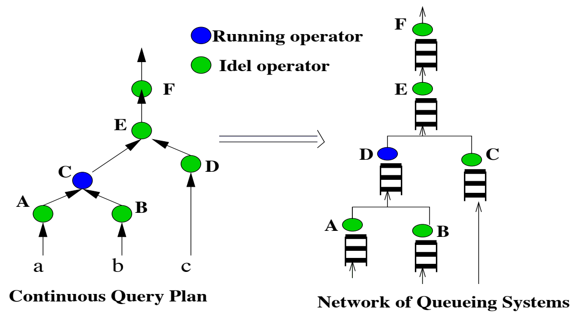

Figure 3.1 Modeling of Continuous Query Processing System.

operator runs as a thread, and the operating system determines which thread to run in any time slot. However, it is hard to control the resource management/allocation in this approach, and the context switch cost can be considerably high. In a single thread approach, all the query plans are registered to a single thread. Various strategies could be used to determine how to allocate system resources, and to schedule which operator or plan to run. The system behavior in a single thread approach is more controllable, and the context switch cost is minimized. For a single thread approach in a multi-processor architecture, the whole set of query plans is partitioned into subsets; each subset of query plans runs on a particular processor as a single thread. Some sharing and interdependent relationships may exist between any two subsets.

We have employed the second approach in this thesis, which enables us to control the various aspects of the system and to manage the QoS requirements associated with each query plan in the best possible way. Furthermore, we have only considered a subset of query plans scheduled on a single processor. A query processing system running on a multi-processor architecture is considered as a set of such independent subsets, where the relationship between any two subsets were ignored in this chapter, which are actually part of our future work.

3.1.1 Modeling

A queueing system is widely used to model various software and hardware systems [92][81]. In our approach, each operator in a given query plan is modeled as the service facility of a queueing system [75], and its input buffer(s) are modeled as a logical input queue with different classes of input tuples if it has more than one input stream. The relationship between two operators is modeled as a data flow between two queueing systems. Eventually, a query plan is modeled as a network of queueing systems as illustrated in Figure 3.1. A multiple continuous query processing system is modeled as a network of queueing systems, which consists of the network of queueing systems for each query plan.

In this queueing model, We call the queueing system whose inputs are the output of another queueing system the *parent queueing system* of its input queueing systems, and the input queueing system as its *children queueing systems*. For each external input stream, there exists an *operator path* from the bottom operator to the top operator; we call this path in our queueing network model as *queueing path*.

Given a subset of query plans scheduled in one processor, at most one operator in this subset query plans is served at any instant of time. Therefore, a scheduling algorithm is necessary to choose an operator to process at each time instant (or time slot), leaving all other operators in the system idle. For an operator in the system, we describe the period when the processor is not available as the *vacation time* for the operator. Furthermore, after the operator gains the processor, it needs some time to setup the environment or to do the initialization (i.e., context switch, scheduling, and so on); we call this period the *setup time*. Therefore, an operator in a multiple query processing system is further modeled as a queueing system with *vacation time* and *setup time*. The time period when the operator is being served is called its *busy period*.

In summary, an operator in our queueing model works as follows: once the operator gains the processor, if its input queue is empty, the operator goes to vacation immediately. Otherwise, the processor needs a setup time and then serves a certain number of tuples determined by a specific service discipline using a first-come-first-served order, and the service of a tuple is non-preemptive. After that, the operator goes to vacation, and the processor goes to serve the other operators or to handle other tasks during its vacation time and then returns for further service. Once again, each operator is modeled as a queueing system with *vacation time* and *setup time*, and a multiple query processing system is modeled as a network of such queueing systems.

3.1.2 Scheduling and Service Discipline

In our proposed queueing model, the vacation time and busy periods are mainly determined by the scheduling algorithm and the service discipline employed in a DSMS, respectively. The scheduling algorithm determines how often an operator is scheduled statistically and how many other operators or non-query processing tasks are scheduled between its two consecutive schedules. Once a scheduling algorithm is employed in a DSMS, we are able to find how many other operators or non-query processing tasks are scheduled between its two consecutive schedules, which also implicitly determines how often we schedule an operator. We can analyze the scheduling algorithm to find these two items. For example, if round Robin (RR) scheduling algorithm is employed in a DSMS with n operators and each operator is scheduled for two time slots once it gains processor, we know that we will schedule one operator again every $2(n-1)$ time slots. Therefore, the vacation time of each operator in such a DSMS is $2(n-1)$ time slots. However, if a non-deterministic scheduling algorithm (in such a scheduling algorithm, an object is scheduled based on run-time information in the system), for example, operator path scheduling [77], Chain [21], and so on, is employed, we collect run-time information to statistically determine how many other operators or non-query processing tasks are scheduled between two consecutive schedules of an operator. For example, we can obtain a distribution of the number of operators or tasks scheduled between two consecutive schedules of an operator.

Just as a scheduling algorithm controls the execution order of the operators in a DSMS, a service discipline controls the number of tuples being served when it gains control of the processor. Takagi [111], [113] gives an overview of various service disciplines. In general, service disciplines can be categorized into two classes: gated-type discipline and exhaustive-type discipline. In the case of the gated-service discipline (GSD), once an operator gains the processor, all tuples waiting in its input queue are served. The tuples which have arrived during the service period are processed in the next round. In the case of exhaustive-service discipline (ESD), once the operator gets the processor, the processor serves all waiting tuples in its input queue as well as the tuples arriving during its service period until its input queue becomes empty. Both service disciplines will be considered in this chapter. Therefore, once the service discipline is employed in a DSMS, given the input characteristics (i.e., input rate) of data streams and the service time of one tuple, we can determine the busy period of an operator.

In summary, the service discipline determines the busy period of an operator and the scheduling algorithm determines the number of operators scheduled between two consecutive schedules of an operator. The service discipline and scheduling algorithm together determine the length of the vacation of an operator. In Section 3.4.4, we discuss in detail how to determine the length of the vacation time for an operator.

3.2 Problem Definition

Tuple latency and the memory requirement of a CQ plan are two fundamental QoS metrics used by most stream-based applications. The problem of estimating both tuple latency and the memory requirement of a CQ plan in this chapter is formally defined as following:

Problem Definition 1. *Given a continuous query plan consisting of m operators $\{O_1, O_2, \cdots, O_m\}$ and m queues* [1] *$\{Q_1, Q_2, \cdots, Q_m\}$ over k data streams $\{S_1, S_2, \cdots, S_k\}$, a scheduling algorithm, and a service discipline in a continuous query processing system in a single processor system, determine:*

1. *the total memory $M(t)$ required by this query in terms of the number of the tuples at time instant t, and*
2. *the tuple latency $R(t)$ at time instant t.*

Both tuple latency and the number of tuples in the above problem are random variables with a continuous parameter (time). It is extremely hard to find the probability distribution function (PDF) for the number of tuples and cumulative distribution function (CDF) for tuple latency in the system. Even if we can find them, the overhead to continuously calculate their values are considerably large. Therefore, we attempt to find approximate mean values of these two performance metrics, which we believe provide sufficient information to manage the QoS requirements of a DSMS. Our experiments show that our approximate solution provides results that are close to what we get from an actual system. Hence, the above problem can be simplified as the following problem that determines mean values.

Problem Definition 2 (Revised Problem). *For the same conditions given in problem 1, determine:*

1. *the total mean memory $E[M]$ required/consumed by this query in terms of the number of tuples, and*
2. *the mean tuple latency $E[R]$ of tuples from this query.*

[1]For those operators that have two or more input queues, we consider all its input queues as one logical queue with different kinds of tuples.

Intuitively, the total mean memory size required by this query plan is the sum of the memory required by each operator in the query plan. So $E[M] = \sum_{i=1}^{m} E[Q_i^s] + \sum_{j=1}^{m} E[O_j^s]$, where $E[Q_i^s]$ is the mean number of tuples in the queue Q_i, and $E[O_j^s]$ is the mean number of tuples maintained by the operator O_j in order to compute the operator over a stream. For non-blocking operators such as *SELECT, PROJECT*[2], $E[O_j^s] = 0$. For *MAX*, *MIN*, $E[O_j^s] = 1$, which is the biggest or smallest value it has seen so far. For a window based symmetric *JOIN* operator [122], and window based aggregate operators, $E[O_j^s]$ is a constant[3], which is the number of tuples in its window(s) under steady state. Therefore, $\sum_{j=1}^{m} E[O_j^s]$ is a constant $\mathcal{C}$. The total memory is reduced to

$$E[M] = \sum_{i=1}^{m} E[Q_i^s] + \mathcal{C} \tag{3.1}$$

To solve the total memory consumed by this query, it is sufficient to find the first component of equation (3.1).

Similarly, the mean tuple latency of an output tuple is the sum of the waiting times at all queues plus the sum of the service times at all operators along its queueing path. The overall tuple latency of a query plan $E[R]$ is the weighted sum of the mean tuple latencies of all queueing paths $E[R_i]$. For a query plan over k data streams, it has k queueing paths or more (e.g., self-join). Therefore,

$$\begin{cases} E[R] = \sum_{i=1}^{k} (\varphi_i E[R_i]) \\ E[R_i] = \sum_{i=1}^{\bar{m}} E[W_i] + \sum_{j=1}^{\bar{m}} E[S_j] \end{cases} \tag{3.2}$$

where φ_i is the queueing path weight, which is the ratio of the number of output tuples from that path to the total number of output tuples from that query plan. $E[W_i]$ is the mean waiting time of a tuple in the queue Q_i, and $E[S_j]$ is the mean processing time of a tuple at the operator O_j. Fortunately, both queueing path weight and processing time

[2]No elimination of duplicated tuples is considered here.

[3]Most cases, its window size is big enough so that a burst in its inputs can be absorbed.

can be learned during query execution by gathering statistics over a period of time. If we expect these values to change over time, we could collect statistics periodically. As a result, we only need to find a solution for the mean waiting time of a tuple in the queues along its path.

3.2.1 Notations and Assumptions

In this chapter, we use the following notations (variables):

λ: finite mean arrival rate of an external input.

U, V, B, S: setup time, vacation time, busy period, and service time distributions, respectively, with their first and second moments $(.)^{(1)}, (.)^{(2)}$, where $(.)$ can be U, V, B, S.

ρ: traffic intensity or utilization with $\rho = \lambda S^{(1)}$.

W: tuple waiting time in the input queue with its first and second moments $W^{(1)}, W^{(2)}$.

C: a cycle which is defined as the period between two successive vacation endings with its first and second moments $C^{(1)}, C^{(2)}$.

Q_n^s: queue length at the end of service of the customer (operator).

Q_n^v: queue length at the end of vacation.

$N^{(.)}$: the number of tuples arriving during the period of $(.)$, where $(.)$ can be U, V, B, C.

$A^*(s)$: the Laplace Stieltjes Transformation (LST) of a distribution A with $A^*(s) = E[e^{(-sA)}]$

$A(z)$: the probability generation function (PGF) of a discrete distribution A with $A(z) = E[z^A]$.

μ: the time of an operator taken to process a tuple.

In our queueing model, we make the following assumptions:

1. The arrival times of tuples from an external input form a Poisson process. Although some reports from the network community show that a self-similar traffic model [109] is closer to the real network traffic data than a Poisson model, a Poisson process is still a good approximation for most applications. As most applications over stream data only require approximate results, the results based on a Poisson process provide sufficient information to manage both QoS and resources.
2. The input queue of an operator has an infinite capacity. Therefore, no tuple is discarded.
3. The *setup time* U and selectivity σ of an operator are known or can be learned by collecting run-time statistics in a DSMS periodically.

3.2.2 Stability and Performance Metrics

In order to solve problem 2, we are interested in the mean queue size and the mean tuple latency. In addition, we need mean cycle time, mean service period, and the mean total number of tuples served during a cycle in order to decide vacation time of an operator. To derive these metrics, we need the whole query processing system in a steady state. As pointed out in [34], the sufficient condition for stability for a single server queue is:

$$\rho + \frac{\lambda}{N^{(1)}}(U^{(1)} + V^{(1)}) < 1$$

where $N^{(1)}$ is the mean number of tuples served in a cycle. For a gated- and exhaustive-discipline, we have $N^{(1)} = \infty$. Hence, $\rho < 1$ is a necessary and sufficient condition for stability.

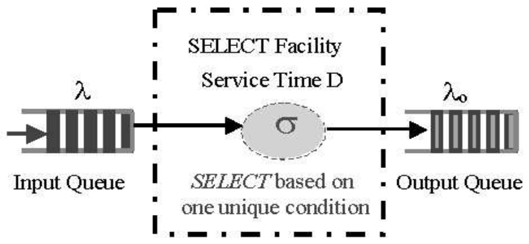

Figure 3.2 Model of *Select* Operator.

3.3 Modeling Relational Operators

Before we derive a closed form solution for tuple latency and memory requirement of a continuous query in a DSMS, it is necessary to identify some characteristics (i.e., waiting time and memory requirement) of an individual operator that encompossed in of a continuous query to determine what factors affect the end-to-end tuple latency and memory requirement of a query. In this section, each operator is modeled as a stand alone queueing system in order to study its characteristics.

In data management systems, *select, project*, and *join* are the most frequently used operators. Once we get the detailed performance metrics of these operators, we will be able to compute the detailed performance metrics of a query, or of an entire query processing system. Some aggregate operators such as *sum* and *average* over a sliding window can be modeled in a similar way.

3.3.1 Modeling SELECT and PROJECT

Both *select* and *project* are single stream processing operators, which have one input and one output queue. The *select* operator works as a filter and evaluates the select condition, which can be either a primitive or a composite condition, over all the input tuples.

3.3.1.1 Select Operator with a Primitive Condition

In this case, every tuple is evaluated by one unique condition, and hence the service time is a constant. So selection over one primitive condition is modeled as a M/D/1 queuing system as shown in Figure 3.2.

Given a *select* operator that takes $1/\mu$ time to process a tuple over a data stream with a mean input rate λ, the mean and variance of the number of tuples in the select subsystem are, respectively,

$$E[\bar{q}] = \rho + \frac{\rho^2}{2(1-\rho)}; \ \ \textit{Where } \rho = \frac{\lambda}{\mu} \tag{3.3a}$$

$$Var[\bar{q}] = \frac{1}{(1-\rho)^2}(\rho - \frac{3}{2}\rho^2 + \frac{5}{6}\rho^3 - \frac{1}{12}\rho^4) \tag{3.3b}$$

Similarly, the mean and variance of the waiting time W of a tuple (waiting time in the queue plus service time) are, respectively

$$E[W] = \frac{1}{\mu} + \frac{\rho}{2\mu(1-\rho)}; \ \ \textit{Where } \rho = \frac{\lambda}{\mu} \tag{3.4a}$$

$$Var[W] = \frac{1}{\mu^2(1-\rho)^2}(\frac{\rho}{3} - \frac{\rho^2}{12}) \tag{3.4b}$$

Note that the second component in (3.4a) is the mean waiting time of the tuple in the queue.

3.3.1.2 Select Operator with a Composite Condition

A composite condition consists of several primitive conditions (a conjunction of disjuncts is assumed). In the worst case, the *select* operator needs to evaluate all of the primitive conditions to determine whether a tuple is output or not. On the other hand, it is also possible to make a decision by evaluating just one of the conditions, in the best case scenario. Therefore, the service time is not a constant, and it depends on the selectivity of each condition.

In an optimized query processing system, the first primitive condition has a relatively small selectivity, and the rest have the same order of selectivity. Thereby, only a small portion of tuples are needed to evaluate more than one condition. The service time can be considered as an exponential distribution because every tuple must be evaluated using the first conjunct, and if it is not applicable, the second conjunct will be used and so on. Apparently, the number of tuples to be evaluated by each successive conjunct decreases exponentially.

THEOREM 1. *Service time of an operator with a composite condition obeys an exponential distribution*

Proof sketch. Without of loss generality, consider a composite condition consisting of n, where $n \geq 2$, primitive conditions $\sigma_0, \sigma_1, \cdots, \sigma_{n-1}$ and service times for primitive conditions are $1/\mu_0, 1/\mu_1, \cdots, 1/\mu_{n-1}$ respectively. A query processing engine decomposes the composite condition during query optimization phase and processes these primitive conditions in an increasing order of selectivity. Without loss of generality, we assume that $\sigma_0 \leq \sigma_1 \leq \cdots \leq \sigma_{n-1}$. Therefore, σ_0 portion of tuples need a service time of $1/\mu_0$, $(1 - \sigma_0)\sigma_1$ (for $n > 2$) portion of tuples need a service time of $(1/\mu_0 + 1/\mu_1)$, $1/\prod_{i=0}^{k-1}(1-\sigma_i)\sigma_k$ (for $1 < k < (n-1)$) portion of tuples need a service time of $\sum_{i=0}^{k} 1/\mu_i$, and $1/\prod_{i=0}^{n-2}(1 - \sigma_i)$ portion of tuples need a service time of $\sum_{i=0}^{n-1} 1/\mu_i$. The number of tuples need more service time decreases exponentially (a factor of $1 - \sigma_i$). As a result, we can safely model the service time of an operator with composite condition as an exponential distribution. □

Hence, the *select* operator with a composite condition is modeled a simple M/M/1 queueing system. For the given situation described in Section 3.3.1.1, the mean and variance of the number of tuples in the select subsystem with a composite condition are, respectively,

$$E[\bar{q}] = \frac{\rho}{1-\rho}; \;\; Var[\bar{q}] = \frac{\rho}{(1-\rho)^2} \; where \; \rho = \frac{\lambda}{\mu} \tag{3.5}$$

The mean and variance of the waiting time are, respectively,

$$E[W] = \frac{1}{\mu(1-\rho)}; \quad Var[W] = \frac{1}{\mu^2(1-\rho)^2} \tag{3.6}$$

3.3.1.3 Project operator

A *project* operator works on a tuple to extract a subset of all the attributes of the tuple. In a traditional DBMS, the *project* operator may also be required to eliminate duplicates. Duplicate elimination, however, may not be applicable to a data stream environment because the output of the project operator is another data stream. Also, elimination of duplicates [4] will introduce blocking, which is undesirable in a stream data environment. Hence this operator is modeled without taking duplicate elimination into consideration, and it works very much like a *select* operator with a primitive condition, and hence is modeled as a M/D/1 system, where the constant service time is the time for extracting the specified subset of attributes of a tuple, and the relative performance metrics for a M/D/1 queueing system as summarized in Section §3.3.1.1.

3.3.2 Modeling Window-based Hash Join

A window-based symmetric hash join algorithm is based on the architecture where an infinite queue is attached to each data stream as illustrated in Figure 3.3. Let Q_1, Q_2 denote the queues attached to left data stream 1 and right data stream 2 respectively, and the output queue Q_O is used to store the results. Typically the size of the reference of a tuple is much smaller than the size of the tuple itself, and hence using references

[4]The duplicate tuples can be eliminated locally by maintaining a hash table based on a time-window.

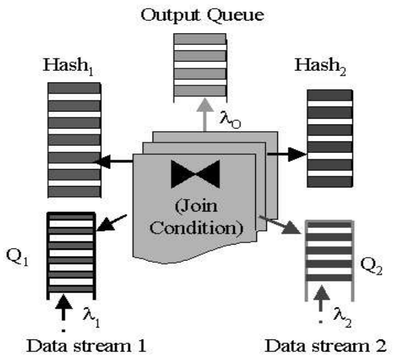

Figure 3.3 Window-based Symmetric Hash Join.

avoids duplicate storage of tuples. Based on this observation, we use a thread to maintain the global tuple pool in a DSMS, including adding new tuples to the pool, passing the reference of the new tuple to the corresponding queue, and deleting an expired tuple and its references in the queues. A tuple expires and is deleted when the difference between its arrival time stamp and current time stamp is greater than the largest time-window required by any operator in the system. For the purpose of this analysis, we define two phases for the above join algorithm: a transition phase and a steady state phase. The transition phase is defined as the period before the first tuple is removed from the last hash table due to its expiration. After the transition phase, the system enters a steady state phase because the mean of the number of tuples in the system fluctuates slightly around a constant if the input rates obey a Poisson distribution and the window size is big enough. Since transition phase is very short in terms of the lifetime of a continuous query, we only analyze the steady state phase in this chapter.

3.3.3 Processing Cost of a Tuple in Steady State

To analyze the computation cost required for processing a new tuple in the steady state, we find that the average number of tuples in each hash table $H_i(n)$ is stable and is given by: $H_i(n) = \lambda_i I_i, i = 1, 2$, where I_i is the predefined time-window for the join operator illustrated in Figure 3.3, where the time-window is the largest time span for a newly arrived tuple to find its match in the corresponding stream. To process one new tuple from stream 1, the join operator needs to do the following operations:

1. insert the tuple into the hash table $Hash_1$;
2. hash the tuple into the hash table $Hash_2$ to find the corresponding bucket;
3. search the corresponding bucket with the join condition;
4. output the matching tuples to the output queue if any.

Mathematically, the total cost for processing one tuple:

$$D_i = 2C_H + C_O + C_E \left(\frac{H_j(n)}{m_j} \right) \begin{cases} j = 2 \textit{ if } i = 1 \\ j = 1 \textit{ if } i = 2 \end{cases} \tag{3.7}$$

where C_H is the cost of hashing, which is a function call in the system, and it is a constant; C_E is the cost of evaluation of the join condition; m_j is the number of buckets in the corresponding hash table. C_O is the cost of output, which is the cost of passing the reference of the matched tuple to the upstream operators or applications for further processing. The output cost is negligible compared to the cost of condition evaluation. Consequently, the total cost D_i of processing a tuple from left data stream is a constant. This holds true for the tuple from data stream 2 as well, though these two costs may be different.

We have not considered the cost of accepting newly arrived tuples and deleting expired tuples so far. It is useful to take a look at how a new tuple is added to the system and an expired tuple is located and deleted before we model the costs of the addition and deletion. In a general query processing system, it is natural to order tuples in a global buffer along a time axis because tuples arrive in an increasing time-stamp order. In our system, a dirty link list maintains the references of all active tuples by adding a newly arrived tuple to its head, a free list links all unused tuples together. Therefore, deleting can be done through periodically checking and truncating the tail of the dirty list. Those expired tuples are linked back to the free list. The cost of both accepting tuples and deleting tuples is a very small portion, say $\alpha\%$, of the system load, and therefore the cost is taken into consideration by multiplying a factor of $1/(1-\alpha)$ to the service time in equation (3.7).

The hash join sub-system can be considered as a typical queuing system with two input queues and a single server that provides the join service. First, we consider the join operator as a service facility with two input queues. Then, we combine those two input queues into a single logical virtual queue with two different classes of tuples, and the service facility provides a constant service time to each class of tuples based on our analysis in Section §3.3.3. Under a steady state, the constant service times may be different for different classes of tuples. Therefore, we model the join subsystem as a $M/(D_1, D_2)/1$ model as illustrated in Figure 3.4, where the logical virtual input queue has a mean arrival rate $(\lambda_1 + \lambda_2)$, and D_1, D_2 are the service times to process a tuple coming from data stream $1, 2$ respectively.

3.3.3.1 Steady State Analysis

Based on above queueing system model, the service times are deterministic for each type of tuple. The queueing system is not a simple Markov chain system because the service time does not obey an exponential distribution and does not satisfy the memoryless property. To fully describe the queue state, we need not only the number of tuples in the queue, but also the service time for which a tuple has already been in the service facility (or the remaining service time prior to its departure from the join service facility). This two-dimensional state system makes queueing analysis much more complicated than for a one-dimensional state system.

The method we present here for finding the mean number of tuples and the mean waiting time in the join subsystem is based on an embedded Markov chain approach [80].

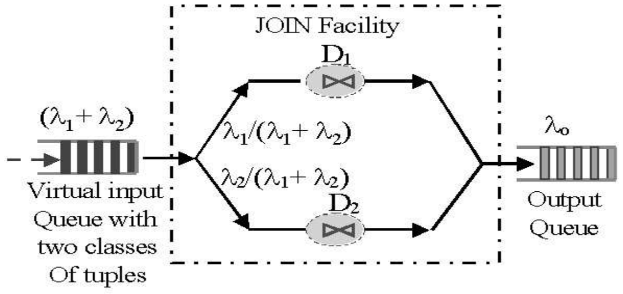
JOIN Facility
D_1
D_2
$(\lambda_1+\lambda_2)$
$\lambda_1/(\lambda_1+\lambda_2)$
$\lambda_2/(\lambda_1+\lambda_2)$
λ_o
Virtual input Queue with two classes Of tuples
Output Queue

Figure 3.4 Queueing Model of Window-based Symmetric Hash Join Operators.

The basic idea behind this method is to simplify the description of the state from a two-dimensional description to a one-dimensional description. The usual way is to sample the two-dimension state information into a special discrete-time point, which explicitly describes the number of tuples in the system, and implicitly contains information about the time spent by a tuple in the join service facility. Usually we select those points at the instant when a tuple in the service facility departs the system. When we specify the number of tuples in the system, we also know that the service time spent for the tuple that just enters the join service facility at that instant is zero. By only considering the point when the tuples leave the join service facility, the state transitions take place only at these points and form a discrete time space. Fortunately, the solution at these embedded Markov points happens to provide a solution for all points in time as well.

In this section, we define the following random variables.

- C_n represents the n^{th} tuple to enter the system.
- τ_n represents the arrival time of C_n.
- $t_n = \tau_n - \tau_{n-1}$ represents the interarrival time between C_n and C_{n-1}.
- X_n represents the service time for tuple C_n.
- q_n represents the number of tuples left behind by the departure of C_n from the service.
- ν_n represents the number of tuples arrives during service time (of C_n).

THEOREM 2. *Given a window-based hash join system over two data streams with input mean rate λ_1, λ_2, and service times D_1, D_2 for the tuples from left stream 1 and right stream 2 respectively, the mean queue size under the steady state is given by:*

$$E[\bar{q}] = \lambda_1 D_1 + \lambda_2 D_2 + \frac{(\lambda_1 + \lambda_2)(\lambda_1 D_1^2 + \lambda_2 D_2^2)}{2(1 - (\lambda_1 D_1 + \lambda_2 D_2))} \tag{3.8}$$

Proof. Considering the time point when C_{n-1} left the system, there are two cases: (*1*) There are no more tuples ($q_n = 0$) in the queue. In this case, the number of tuples in the queue is the number of tuples arriving during the service period X_{n-1} (of C_{n-1}); (*2*) There are q_n tuples ($q_n > 0$) in the queue. Therefore, the number of tuples left in the queue is the number of tuples that arrive during the service period of time X_{n-1} (of C_{n-1}) plus the number of tuples left behind when tuple C_n left the system minus 1. Based on the above analysis,

$$\begin{cases} q_{n+1} = q_n + v_{n+1} - 1 & \textit{if } q_n > 0 \\ q_{n+1} = v_{n+1} & \textit{if } q_n = 0 \end{cases} \tag{3.9}$$

We define the following step function

$$\delta_k = \begin{cases} 1 & \textit{if } k = 1, 2, \cdots \\ 0 & \textit{if } k \leq 0 \end{cases} \tag{3.10}$$

we can rewrite (3.9) as

$$q_{n+1} = q_n - \delta_n + v_{n+1} \tag{3.11}$$

If the system is an ergodic one, and when $n \to \infty$, we have $E[\delta] = E[\bar{v}]$, where $E[\delta] = P(\textit{system busy}) = \rho$. So

$$E[\bar{v}] = \rho \tag{3.12}$$

Since random variables v and q are independent, $E[v_n q_n] = E[v_n]E[q_n]$, *also* $E[q_n \delta_n] = E[q_n]$. Take square and then expectation on both sides of equation (3.11), when $n \to \infty$,

$E[\bar{q}_{n+1}^2] = E[\bar{q}_n^2] + E[\bar{v}_{n+1}^2] + E[\bar{\delta}_n^2] + 2E[\bar{q}_n]E[\bar{v}_{n+1}] - 2E[\bar{q}_n]E[\bar{\delta}_n] - 2E[\bar{v}_{n+1}]E[\bar{\delta}_n]$. Here $E[\bar{\delta}^2] = E[\bar{\delta}]$ because $\delta^2 = \delta$. Then we get

$$E[\bar{q}] = E[\bar{\delta}] + \frac{E[\bar{v}^2] + E[\bar{\delta}]}{2(1 - E[\bar{v}])} \tag{3.13}$$

The only term unknown in (3.13) is $E[\bar{v}^2]$. From the properties of Z-transform, we can get the k^{th} moments easily if we can find the Z-transform function of random variable v. By definition of Z-transform, we have

$$\begin{aligned} V[z] &= \sum_{k=0}^{\infty} P(\bar{v} = k) z^k \\ &= \sum_{k=0}^{\infty} \left(\frac{(\lambda D_1)^k}{k!} e^{-\lambda D1} + \frac{(\lambda D_2)^k}{k!} e^{-\lambda D_2}\right) z^k \\ &= P(D_1) e^{-\lambda D_1 (1-z)} + P(D_2) e^{-\lambda D_2 (1-z)} \end{aligned} \tag{3.14}$$

where $P(D_1) = \frac{\lambda 1}{\lambda_1 + \lambda_2}$ and $P(D_2) = \frac{\lambda 2}{\lambda_1 + \lambda_2}$; taking the first derivative of (3.14), $\frac{dv(z)}{d_z} \|_{z=1}=$ $P(D_1)\lambda D_1 + P(D_2)\lambda D_2 = \lambda_1 D_1 + \lambda_2 D_2 = \rho = E[\bar{v}]$, and taking the second derivative of (3.14), $\frac{d^2 v(z)}{d_z^2} \|_{z=1}= P(D_1)(\lambda D_1)^2 + P(D_2)(\lambda D_2)^2$ We find $E[\bar{v}] = E[\bar{v}] + \lambda(\lambda_1 D_1^2 + \lambda_2 D_2^2)$, substituting it into (3.13), which yields

$$E[\bar{q}] = \lambda_1 D_1 + \lambda_2 D_2 + \frac{(\lambda_1 + \lambda_2)(\lambda_1 D_1^2) + \lambda_2 D_2^2}{2(1 - (\lambda_1 D_1 + \lambda_2 D_2))} \tag{3.15}$$

□

Remark 1. : *1*). The first part (first two items) of the above equation is the mean number of tuples arrived during the service time of one tuple. The second part is the mean number of tuples that are left in the queue before the tuple that is in service has entered the service facility.

2). The mean queue size is invariant when both the processing ability and the input rate are increased to k times, namely $\lambda_1^{'} = k\lambda_1, \lambda_2^{'} = k\lambda_2$ and $D_1^{'} = \frac{D_1}{k}, D_2^{'} = \frac{D_2}{k}$. This can be shown by substituting them into equation (3.8). So the utilization $\rho = \lambda_1 D_1 + \lambda_2 D_2$ is the only factor that affects $E[\bar{q}]$ in the system.

THEOREM 3. *Given a window-based hash join system described in Theorem 2, the probability distribution of the number of tuples in such a system under the steady state is given by*

$$P(N = k) = (1 - \rho)d_k \tag{3.16}$$

where $d_k = \begin{cases} a_k - \sum_{j=1}^{k} b_j d_{k-j} \textit{if } k = 1, 2, \cdots \\ 1 \qquad \textit{if } k = 0 \end{cases}$

$a_k = \frac{c_k}{c_0 k!} - \frac{c_{k-1}}{c_0 (k-1)!} \qquad \textit{for } k = 1, 2, \cdots;$

$b_1 = \frac{c_1 - 1}{c_0}; \textit{ and } b_k = \frac{c_k}{c_0 k!} \qquad \textit{for } k = 2, 3, \cdots;$

$c_k = \frac{\lambda_1}{\lambda} e^{-\lambda D_1} (\lambda D_1)^k + \frac{\lambda_2}{\lambda} e^{-\lambda D_2} (\lambda D_2)^k \textit{ for } k = 0, 1, \cdots;$

$\lambda = \lambda_1 + \lambda_2$

Proof. Take Z-transform on both sides of (3.11),

$$Z^{q_{n+1}} = Z^{q_n - \delta_n + v_{n+1}} \tag{3.17}$$

According to the Z-transform property, the Z-transform of the sum two random variables equals the product of those two random variables' Z transform.

$$Z^{q_{n+1}} = Z^{q_n - \delta_n} Z^{v_{n+1}} \tag{3.18}$$

Where $Z^{q_{n+1}} = \sum_{k=0}^{\infty} P(q_{n+1} = k) Z^{k-1}$ and $Z^{q_n - \delta_n} = P(q_n = 0) + \frac{1}{z}(\sum_{k=0}^{\infty} P(q_n = k) z^k - P(q_n = 0))$. Since$P(q_n = 0) = 1 - \rho$,

$$Z^{q_n - \delta_n} = 1 - \rho + \frac{1}{z}(Z^{q_n} - (1 - \rho)) \tag{3.19}$$

We denote the Z-transform of random variable q_n, v_n by Q(z), V(z) respectively, and simplify (3.18),

$$Q(z) = V(z) \frac{(1 - \rho)(1 - z)}{V(z) - z} \tag{3.20}$$

Substituting (3.14) and $\rho = \frac{\lambda_1 D_1 + \lambda_2 D_2}{\lambda}$ into (3.20),

$$Q(z) = \frac{(1-\rho)(1+\sum_{k=1}^{\infty}(\frac{c_k}{c_0 k!} - \frac{c_{k-1}}{c_0(k-1)!})z^k)}{1+\frac{c_1-1}{c_0}z + \sum_{k=2}^{\infty}\frac{z^k}{c_0 k!}} \tag{3.21}$$

where $c_k = \frac{(\lambda D_1)^k}{k!}e^{-\lambda D1} + \frac{(\lambda D_2)^k}{k!}e^{-\lambda D_2}$; Furthermore, we can rewrite the (3.21) as

$$Q(z) = (1-\rho)\frac{1+\sum_{k=1}^{\infty} a_k z^k}{1+\sum_{k=1}^{\infty} b_k z^k} = (1-\rho)\sum_{k=1}^{\infty} d_k z^k \tag{3.22}$$

where

$a_k = \frac{c_k}{c_0 k!} - \frac{c_{k-1}}{c_0(k-1)!}$

$b_1 = \frac{c_1-1}{c_0};\ \ and\ b_k = \frac{c_k}{c_0(k-1)!}$

$d_k = \begin{cases} a_k - \sum_{j=1}^{k} b_j d_{k-1} & if\ k = 1, 2, \cdots \\ 1 & if\ k = 0 \end{cases}$

□

From Theorem 3, it follows that

$P(N=0) = 1-\rho$

$P(N=1) = (1-\rho)(\frac{1}{\frac{\lambda_1}{\lambda}e^{-\lambda D_1} + \frac{\lambda_2}{\lambda}e^{-\lambda D_2}} - 1)$

$P(N=2) = (1-\rho)(\frac{c_2}{2c_0} - \frac{c_1 + \frac{3}{2}c_0 - 1}{c_0^2})$

Corollary 1. *The variance of the number of tuples in such a join subsystem is given by*

$$Var[\bar{q}] = (1-\rho)\sum_{k=2}^{\infty} k(k-1)d_k + E[\bar{q}] - E[\bar{q}]^2$$

Proof. It follows from Theorem 3 by taking Z-transform, $Q(z) = (1-\rho)\sum_{k=0}^{\infty} d_k z^k$, and then taking the second derivative, $\frac{d^2Q(z)}{d_z^2}\|_{z=1} = (1-\rho)\sum_{k=2}^{\infty} k(k-1)d_k = E[\bar{q}^2] - E[\bar{q}]$. So the variance of number of tuples in the system $Var[\bar{q}] = E[\bar{q}^2] - E[\bar{q}]^2 = (1-\rho)\sum_{k=2}^{\infty} k(k-1)d_k + E[\bar{q}] - E[\bar{q}]^2$. □

THEOREM 4. *Given a window-based hash join system described in Theorem 2, the mean waiting time of tuples in the join subsystem and in the queue under the steady state are given by, respectively,*

$$W = \frac{\lambda_1 D_1 + \lambda_2 D_2}{\lambda_1 + \lambda_2} + \frac{\lambda_1 D_1^2 + \lambda_2 D_2^2}{2(1 - (\lambda_1 D_1 + \lambda_2 D_2))} \tag{3.23a}$$

$$W_q = \frac{\lambda_1 D_1^2 + \lambda_2 D_2^2}{2(1 - (\lambda_1 D_1 + \lambda_2 D_2))} \tag{3.23b}$$

Proof. According to the Little's result, the waiting time of the tuples in the system $W = \frac{\bar{N}}{\lambda} = \frac{E[\bar{q}]}{\lambda}$. Substituting (3.8) into the above equation, we get (3.23a). And the second part of (3.23a) is the mean waiting time in the queue. □

Remark 2. *: If we increase, at the same time, the processing ability and the input rate k times, as we did earlier, the mean waiting times of the tuples in both the system and the queue decrease to $1/k$ of the original mean waiting time. This can be shown by substituting the changed factors into the equations* (3.23a) *and* (3.23b).

From Remark 1 and 2, we clearly show that a bigger system with n units processing power and n units input rate is better than n small systems with 1 unit processing power and 1 unit input rate. Although both the bigger system and all small systems require the same amount of memory, the tuple latency in the bigger system is one n^{th} of that in a small system. Therefore, to process data streams, clustering multiple smaller systems into a bigger system is better than multiple standalone small systems.

Corollary 2. *The variance of waiting time of tuples in such a join subsystem is give by*

$$Var(W) = \frac{1 - \rho)}{\lambda^2} \sum_{k=2}^{\infty} k(k-1)d_k - W^2.$$

Proof. From the generalized Little's formula [92], we know $\frac{d^k Q(z)}{d_z^k} \|_{z=1} = \lambda_k^W$, where W_k is the k^{th} moment of the waiting time of tuples in the system. The variance of W $Var(W) = W_2 - W^2 = \frac{1-\rho}{\lambda^2} \sum_{k=2}^{\infty} k(k-1)d_k - W^2$, where W can be obtained from (3.23a), d_k is given in (3.16). □

THEOREM 5. *Given a window-based hash join system described in Theorem 2, the probability distribution of the waiting time of tuples in the queue under the steady state is given by*

$$W(t) = 1 - (\lambda_1 D_1 + \lambda_2 D_2)e^{(s_0 t)} \tag{3.24}$$

where s_0 is its negative root of equation (3.29)

Proof. For our $M/(D_1, D_2)/1$ queueing model, we have the Cumulative Distribution Function (CDF) of service time

$$B(s) = \begin{cases} 0 & \textit{if } s < D_1 \\ \frac{\lambda_1}{\lambda_1+\lambda_2} & \textit{if } D_1 \leq s < D_2 \\ 1 & \textit{if } s \geq D_2 \end{cases} \tag{3.25}$$

and its probability density function is
$b(t) = \frac{\lambda_1}{\lambda_1+\lambda_2}\delta(t)(t - D_1) + \frac{\lambda_2}{\lambda_1+\lambda_2}\delta(t)(t - D_2)$
where $\delta(t) = \begin{cases} 1 & t \geq 0 \\ 0 & t < 0 \end{cases}$. Then, computing its LST B*(s)

$$B^*(s) = \frac{\lambda_1}{\lambda_1 + \lambda_2}e^{sD_1} + \frac{\lambda_2}{\lambda_1 + \lambda_2}e^{sD_2} \tag{3.26}$$

Also, the LST of the interarrival probability distribution function is given by

$$A^*(s) = \frac{\lambda_1 + \lambda_2}{(\lambda_1 + \lambda_2) + s} \tag{3.27}$$

By substituting (3.26) and (3.27) into the characteristic equation $A^*(-s)B^*(s) = -1$, defined in Lindley's equation [92], we have

$$\lambda_1 e^{-sD_1} + \lambda_2 e^{-sD_2} - (\lambda_1 + \lambda_2) + s = 0 \tag{3.28}$$

Obviously, zero is one of its roots. We can also easily find a numerical solution for the above equation by MatLab or any other mathematical tool. The CDF of the waiting-time for our model

$$W(t) = 1 - (\lambda_1 D_1 + \lambda_2 D_2)e^{(s_0 t)} \tag{3.29}$$

The distribution of the waiting time in the system is the same as that in the queue except we need to add the constant service time D to the above equation. □

3.3.4 Extensions

The approach presented so far is generally true for any window-based symmetric join operator, with the only difference being the value of the service time. If a stream data processing system also accesses a stored database, we may need to join the tuples from a data stream with a data set stored on a disk. On the other hand, the input process may not be a Poisson process; we briefly discuss the impact of more bursty inputs on the performance of these models.

3.3.4.1 One Relation on Local Disks

If the entire relation on local disks can fit in main memory, we can load all the tuples into memory before we start the join algorithm. There is only one input data stream for this case. The join operator works as a single stream processing operator and hence the system can be modeled as a M/D/1 queue with the mean arrival rate λ_1 and constant service time D_1, which is the cost for a tuple from input stream to join with all tuples in the relation. If all the tuples of the relation on local disks cannot fit in memory, the cost of processing a tuple not only includes those costs listed in Section §3.3.3, but also the extra cost of loading tuples from disks and paging out the probed tuples to the disks. However, the service time is still a constant because for each input tuple from input stream, the cost of loading tuples from disks and paging out the probed tuples to disks is the same. Therefore, the system can again be modeled as a M/D/1 queue with a larger service time.

Corollary 3. *The mean number of tuples in such a join system when one relation is on local disks is*

$$E[\bar{q}] = \lambda_1 D_1 + \frac{\lambda_1^2 D_1^2}{2(1 - \lambda_1 D_1)}.$$

It follows from (3.8) by setting $D_2 = 0$ and $\lambda_2 = 0$.

Corollary 4. *the mean waiting time of tuples in such a system and in the queue are* $W = D_1 + \frac{\lambda_1 D_1^2}{2(1-\lambda_1 D_1)}$ *and* $\bar{W}_q = \frac{\lambda_1 D_1^2}{2(1-\lambda_1 D_1)}$ *and respectively.*

Similarly, the above results can be derived by setting $D_2 = 0$ and $\lambda_2 = 0$ in (3.23a) and (3.23b).

The queue size in Corollary 3 and the mean waiting time in Corollary 4 have the same forms as the results of a standard M/D/1 presented in Section §3.3.1.1. This verifies the correctness of our analysis in Section §3.3.3.1.

3.3.4.2 Impact of Bursty Inputs

Although many data streams can be approximately modeled as a Poisson input model, many researchers have recently shown that a class of input data streams are much more bursty than Poisson inputs. This class of input data streams exhibits a so-called *self-similar*[85] property over a wide range of time scale. Partial explanation of this property is that the input data stream itself is a superimposition of many (theoretically, infinite) ON/OFF sub-streams, the distributions of the length of ON/OFF periods are a heavy tailed distribution. The input data streams of a sensor database may demonstrate the *self-similar* property because thousands of sensors send data during ON periods, and they are in OFF (or in sleep mode) in order to save energy; more investigation is needed to prove this.

Due to the modeling difficulties of queueing systems fed by self-similar data streams, we do not intend to do an exact analysis. Instead, we highlight here a few key points of the impact of a bursty input data stream on queueing performance. Under the steady state,

1. Both queue size and tuple latency in the queue have a heavy-tailed distribution, but with different power. The larger queue size and longer tuple latency do not decrease exponentially as they do in a Poisson model; instead, the tails of the distribution are much longer than those in standard Poisson model.
2. The more bursty the input data streams, the longer are both the queue size and tuple latency. The bursty property of the input data streams is mainly dominated by the distribution of the ON periods, not by the departure distribution of the tuples within ON periods.
3. The queueing metrics obtained from a Poisson input process are similar to those obtained from *self-similar* input streams when the system load is not high. However, as the system load increases, the difference based on these two inputs increases dramatically.

3.4 Modeling Continuous Queries

In the previous section, we analyzed the performance metrics of each operator that is used in a query plan based on a single-server queueing model. However, in a multiple query processing system, there is only one server (processor) available for all operators in the system. And the input process of some operators does not form a Poisson distribution because those operators get their inputs from the outputs of other operators. Even if the input process of an operator forms a Poisson distribution, its output does not form a Poisson distribution. Therefore, the end-to-end tuple latency or memory requirement of query plan is not simply the sum of its corresponding parts in those standard queueing systems presented in previous section where each of them requires a dedicated server.

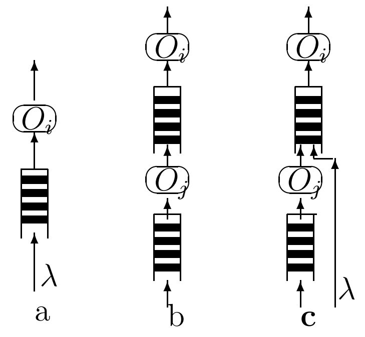

Figure 3.5 Three Classes Of Queueing Models.

In a multiple query processing system, some operators have an Exponential service time while others have a deterministic service time or a more general service time. Therefore, all operators with only external inputs in a general DSMS are modeled as an $M/G/1$ queueing system with a vacation time V and a setup time U as we discussed in §3.1.1, and those queueing systems form a network of queueing systems. In this queueing network, the queueing system for individual operators can be categorized into three classes based on their inputs, as illustrated in Figure 3.5.

1. Queueing system with external input(s) (Figure 3.5-a). This class has only external input(s) from continuous data streams. Whether it is in a vacation period or in a serving period, the input tuple is inserted into its input queue immediately when it arrives.
2. Queueing system with internal input(s)(Figure 3.5-b). This class of queueing system only has input(s) from the output of another operator in the system. The arrival time of an input tuple is the departure time of the output process of another operator. Therefore, this class only has inputs during its vacation period, and no input during its setup time and serve time.
3. Queueing system with external input and internal input(Figure 3.5-c). This class has both internal input and external input, and its inputs are a combination of the above two classes.

The above three classes capture all the alternatives possible for a query plan. In the rest of this section, we will derive the mean queue size and the mean waiting time of each class of queueing model under both gated-service and exhaustive-service disciplines. Additionally, to decide the vacation time and the internal input rate(s) of the queueing models, we have to derive the busy period and the total number of tuples served during a cycle as well. In this section, we assume that the vacation period V of an operator is known. We will discuss how to determine vacation period of each operator in Section §3.4.4.

3.4.1 Queueing Model with External Input(s)

In the model illustrated in Figure 3.5-a, each external input has a dedicated thread in the system which accepts input tuples whenever they arrive. The prototype of the queueing model can be a *SELECT* or a *PROJECT* or aggregate operator over a continuous data stream, or a *JOIN* operator over one (self-join) or two continuous data streams. If the operator has two or more external inputs, we consider the inputs as one logical external input with different class of input tuples and with a mean arrival rate which is the sum of all mean arrival input rates of all inputs of the operator.

3.4.1.1 Exhaustive-service discipline

First of all, we need to compute the length of a busy period of an operator under an exhaustive-service discipline. Using that we can derive the number of tuples it served during its busy period, the output rate, and the cycle time.

Corollary 5. *Given a queueing system with external input(s) and its vacation time V, its setup time U, its mean input rate λ, and service time (to serve one tuple) S, the mean* *busy period at each round this queueing system with an exhaustive-service discipline under the steady state is given by,*

$$B^{(1)} = \frac{\rho}{1-\rho}\left((1-q_0)\,U^{(1)} + V^{(1)}\right) \tag{3.30}$$

Proof. Let N be the random variable representing the total number of tuples arrived during vacation and setup time. We have

$$N = \left\{N^U + N^V\right\} \|_{Q^v>0} \tag{3.31}$$

taking Z transform on both sides

$$N(z) = U^*(\lambda - \lambda z)V^*(\lambda - \lambda z) + q_0(1 - U^*(\lambda - \lambda z)) \tag{3.32}$$

where $q_0 = V^*(\lambda)$ is the probability of an empty queue after one vacation time. Then the busy period B in our $M/G/1$ queueing system with setup time and vacation time consists of N standard busy periods in a standard $M/G/1$ system.

$$B = \sum\nolimits_{i=1}^{N} \bar{B}_i \tag{3.33}$$

where $\bar{B}_i$ is the standard busy period introduced by i^{th} tuple. Taking LST on both sides of (3.33),

$$\begin{aligned} B^*(s) =& N(\bar{B}^*(s)) \\ =& U^*(\lambda - \lambda\bar{B}^*(s))V^*(\lambda - \lambda\bar{B}^*(s)) \\ &+ q_0\left(1 - U^*(\lambda - \lambda\bar{B}^*(s))\right) \end{aligned} \tag{3.34}$$

According to the relationship of the LST of the standard busy period and the LST of service time distribution,

$$S^*(s) = \bar{B}^*(s + \lambda - \lambda S^*(s)) \tag{3.35}$$

From (3.34) and (3.35), we have

$$B^{(1)} = \frac{\rho}{1-\rho}\left((1-q_0)\,U^{(1)} + V^{(1)}\right)$$

□

Remark 3. *By taking the nth derivative from equation* (3.34) *and* (3.35)*, we can get the nth moment of the busy period. In this chapter, as we are only interested in mean values, no higher moment is given. The interested reader can take it further.*

Corollary 6. *Given a queueing system with external input(s) described in Corollary 5, its mean cycle time at each round of this queueing system with an exhaustive-service discipline under the steady state is given by,*

$$C^{(1)} = \left(\frac{1-\rho q_0}{1-\rho}\right)\left(U^{(1)} + \frac{1}{1-q_0}V^{(1)}\right) \tag{3.36}$$

Proof. A cycle consists of a busy period, a setup period, and one or more vacation periods. It has m vacation periods only if the queue is empty after the first $m-1$ vacations, and there is at least one tuple arrival during the m^{th} vacation period. Therefore,

$$C^{(1)} = U^{(1)} + B^{(1)} + E\left[\sum\nolimits_{m=1}^{\infty}\left(mV^{(1)}P(\mathrm{m})\right)\right] \tag{3.37}$$

where $P(\mathrm{m})$ is the probability of that a cycle includes m vacations and $P(\mathrm{m}) = (1-q_0)q_0^{(m-1)}$. The mean length of vacation time $E[\sum_{m=1}^{\infty}(mV^{(1)}P(m))] = \frac{1}{1-q_0}V^{(1)}$. Plug it into (3.37), we have (3.36). □

Corollary 7. *Given a queueing system with external input(s) described in Corollary 5, the mean number of tuples served during each round is given by:*

$$N^{C^{(1)}} = \frac{\lambda}{1-\rho}\left((1-q_0)U^{(1)} + V^{(1)}\right) \tag{3.38}$$

Proof. number of tuples N^C served during one cycle under the exhaustive-service discipline is the total number of tuples N that have arrived during the vacation time and setup time, plus the number of tuples arrived during the busy period. Therefore,

$$N^C = \sum\nolimits_{i=1}^{N} F_i \tag{3.39}$$

where F_i is the total number of tuples introduced by the i^{th} tuple in the input queue during a standard busy period in a standard M/G/1 queue. Taking Z transform on both sides of (3.39), we have

$$\begin{aligned} N^C(z) =& N\left(F(z)\right) \\ =& U^*\left(\lambda - \lambda F(z)\right) V^*\left(\lambda - \lambda F(z)\right) \end{aligned} \tag{3.40}$$

In a standard M/G/1 queue, the Z transform of the number of tuples during a standard busy period has the following relationship with the Z transform of the service time,

$$F(z) = zS^*\left(\lambda - \lambda F(z)\right) \tag{3.41}$$

From (3.40) and (3.41), we get (3.38) by taking derivative. □

THEOREM 6. *Given a queueing system with external input(s) described in Corollary 5, the mean queue size and mean waiting time of tuples in this queueing system with an exhaustive-service discipline under the steady state are given by, respectively,*

$$\begin{aligned} Q^{(1)} =& \rho + \lambda W^{(1)} \\ W_q^{(1)} =& \frac{\lambda S^{(2)}}{2(1-\rho)} + \frac{U^{(2)} + 2U^{(1)}V^{(1)} + V^{(2)} - q_0 U^{(2)}}{2(1-q_0)U^{(1)} + V^{(1)}} \\ W^{(1)} =& S^{(1)} + W_q^{(1)} \end{aligned} \tag{3.42}$$

where $q_0 = V^*(\lambda)$ *is the probability of an empty queue after one vacation time.*

Proof sketch. According to the decomposition property of M/G/1 queue with vacation time [113], and considering the vacation period termination points, we derive the mean queue size $Q^{(1)}$, and waiting time in the queue and in the queueing system. □

With the increase in system load, we can see that the vacation period increases because the busy periods of other operators in the system increase. As a result, both the waiting time in the queue and queue size increase.

3.4.1.2 Gated-service discipline

For a gated-service discipline, the processor only serves the tuples that have arrived before the busy period, while the tuples arriving during the busy period will be processed in the next round.

THEOREM 7. *Given a queueing system with external input(s) and its vacation time V, its setup time U, its mean input rate λ, and service time (to serve one tuple) S, the mean queue size and mean waiting time of tuples in this queueing system with an gated-service discipline under the steady state are given by, respectively,*

$$\begin{aligned} Q^{(1)} &= \rho + \lambda W^{(1)} \\ W_q^{(1)} &= \frac{\lambda S^{(2)} + 2U^{(1)} + 2\rho V^{(1)}}{2(1-\rho)} + \\ & \frac{(1-q_0)(U^{(2)} + 2U^{(1)}V^{(1)} + V^{(2)})}{2((1-q_0)U^{(1)} + V^{(1)})} \end{aligned} \tag{3.43}$$

where q_0 has a different value from that under an exhaustive-service discipline, and

$$q_0 = \frac{1}{D} \prod_{j=0}^{\infty} U^*(\lambda - \lambda\eta_j(0)) V^*(\lambda - \lambda\eta_j(0)) \tag{3.44}$$

with denominator $D = 1 - V^*(\lambda)(1-U^*(\lambda)) - \sum_{k=1}^{\infty}(V^*(\lambda-\lambda\eta_k(0))(1-U^*(\lambda-\lambda\eta_k(0)))\prod_{j=0}^{k-1}U^*(\lambda-\lambda\eta_j(0))V^*(\lambda-\lambda\eta_j(0)))$, where $\eta_j(0)$ is given recursively by

$$\begin{cases} \eta_0(z) = z \\ \eta_{j+1}(z) = S*(\lambda - \lambda\eta_j(z))j = 0, 1, \cdots \end{cases}$$

For the moments of the queue length/waiting time, we use the same approach as in [27]. The results are derived by analyzing the departure point of the nth customer in a service period.

Corollary 8. *Given a queueing system with external input(s) described in Theorem 7, its mean busy period at each round in this queueing system with a gated-service discipline under the steady state is given by,*

$$B^{(1)} = \frac{\rho}{1-\rho}\left((1-q_0)U^{(1)} + V^{(1)}\right) \tag{3.45}$$

Proof. The number of tuples served during a cycle consists of the tuples arrived during last setup period, the tuples arrived during last busy period and the tuples arrived during this vacation period(s). Considering the tuples that are served during busy period, when $n \to \infty$,

$$N^C = (N^U + N^B)\|_{Q^V>0} + N^V \tag{3.46}$$

Therefore, the busy period is the total service time of those N^C tuples. As a result, we have (3.45). $\square$

Although the mean number of tuples under both exhaustive-service and gated-service disciplines are the same, they have different higher moments.

Corollary 9. *Given a queueing system with external input(s) described in Theorem 7, its mean number of tuples served at each round in this queueing system with an gated-service discipline under the steady state is given by,*

$$\begin{aligned} N^{C(1)} &= \lambda(U^{(1)} + \frac{1}{1-q_0}V^{(1)} + B^{(1)}) \\ &= \frac{\lambda(1-\rho q_0)}{1-\rho}\left(U^{(1)} + \frac{1}{1-q_0}V^{(1)}\right) \end{aligned} \tag{3.47}$$

By taking the derivative of both sides of (3.46), we get the above result.

Corollary 10. *Given a queueing system with external input(s) described in Theorem 7, its mean cycle time in this queueing system under gated-service discipline is the same as under an exhaustive service discipline under the steady state, which is given in Corollary 7.*

Proof sketch. As we mentioned earlier, one cycle includes one busy period, one setup period, and one or more vacation times. The number of vacation times are only related to the q_0, therefore, the lengths of vacation times under both gated-service and exhaustive-service are the same. This applies to the mean busy period and consequently holds true for the cycle time, while higher moments are different due to the difference in high moments of the busy periods under these two service disciplines. □

3.4.2 Queueing Model with Internal Input(s)

In this model, the queueing system has only internal input, which is the output of processes of its children queueing systems. Therefore, the input process is neither a Poisson process nor a continuous stream. An operator has outputs, if any, only during its busy period. The number of tuples outputted from an operator is decided by both the total number of tuples N processed during one cycle and the selectivity σ of its child operator. For the *SELECT* operator, the selectivity $\sigma \leq 1$; for the *PROJECT* operator, $\sigma = 1$. If the operator is a *JOIN* operator, its selectivity may be greater than 1, which is decided by the selectivities of its two *SEMI-JOIN* operators σ_L, σ_R. We assume that selectivity of an operator is known; otherwise, it can be learned through collecting run-time statistic information in a DSMS.

To derive similar performance metrics as those in §3.4.1.1, we use the following approximate method: consider the operator, such as *SELECT* or *PROJECT*, which has only one internal input as illustrated in Figure 3.5-b. In the steady state, we assume that the operator j runs once after its child operator i runs k times, which is called the *weight ratio* of operator i in terms of operator j hereafter. Here, $k \geq 1$ because operator j can only enter its busy period with at least one output tuple from its child operator i. The value k is determined by the scheduling strategy. An example well be given in Section 3.4.4 to demonstrate how to determine k.

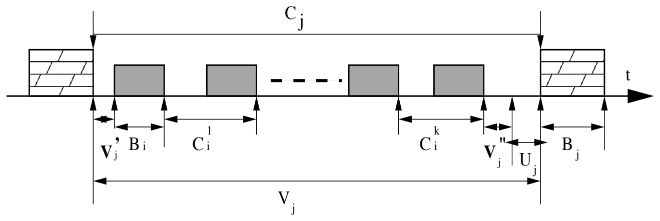

Figure 3.6 Input Process Of Operator (Internal Input).

For this queueing model, there is no input during its busy periods (since all other operators are idle). Therefore, an exhaustive-service discipline behaves exactly like a gated-service discipline. The input process of the operator j is shown in Figure 3.6. After a busy period is over, the operator j waits for a random period $V_j^{'}$, called *pre-input vacation period* during which time some other operators, if the system does not serve its child operator i immediately after it has served operator j, can be served. It then receives its first batch of inputs from its child operator i during its child operator i's busy period. After waiting another cycle time of its child operator i, it receives another batch of inputs. k rounds later, it waits another random time $V_j^{''}$ called the *post-input vacation period* during which time some other operators except its child operators can be served; it finally begins its own service cycle, consisting of its own setup time U_j and busy period B_j to serve all the tuples in its input queue. We consider the input of an operator under this model as a discrete batch of inputs and use the following method to approximate the performance metrics.

Corollary 11. *Given a queueing system with internal input(s) and its vacation time V, its setup time U, its mean input rate λ, and service time (to serve one tuple) S, the mean number of tuples served during one cycle under both gated-service and exhaustive-service discipline is given by:*

For single-input operators,

$$N_j^{C(1)} = k^{(1)} N_i^{C(1)} \sigma_i$$

For two-input operators,

$$N_j^{C(1)} = k_i^{(1)} N_i^{C(1)} \sigma_i + k_{i+1}^{(1)} N_{i+1}^{C(1)} \sigma_{i+1}$$

Proof sketch. The total number of tuples served during a cycle is the sum of k batch inputs. Therefore,

$$N_j^{C(1)} = k^{(1)} N_i^{C(1)} \sigma_i \tag{3.48}$$

where $N_i^{C(1)}$ is the number of tuples at one batch input, which is given in Corollary 7 if the operator i has only external input(s). Otherwise, we have to first compute the number of tuples served during one cycle at its child operator, or its grandchild operator until we reach the bottom operator that only has external input(s).

If the operator j has two internal inputs - one is from its left operator i^{th} and another from its right child operator $(i+1)^{th}$, the total number of tuples served during a cycle time is the sum of the number of tuples from its left child and from its right child.

$$N_j^{C(1)} = k_i^{(1)} N_i^{C(1)} \sigma_i + k_{i+1}^{(1)} N_{i+1}^{C(1)} \sigma_{i+1} \tag{3.49}$$

where k_i, k_{i+1} are the weight ratios of operator $i, i+1$ in terms of operator j respectively.

□

Corollary 12. *Given a queueing system with internal input(s) described in Corollary 11, the mean busy period at each round under both gated-service and exhaustive service disciple is given by*

$$B_j^{(1)} = N_j^{C(1)} S_j^{(1)} \tag{3.50}$$

The busy period is simply the sum of the service times to serve these $N_j^{C(1)}$ tuples.

$$B_j^{(1)} = N_j^{C(1)} S_j^{(1)} \tag{3.51}$$

Corollary 13. *Given a queueing system with internal input(s) described in Corollary 7, the mean cycle time at each round under both gated-service and exhaustive service disciple is given by*

$$C_j^{(1)} = U_j^{(1)} + B_j^{(1)} + \frac{1}{1-q_0} V_j^{(1)}$$

where $q_0 = P(N_j = 0)$.

Clearly, the cycle time in this model has the same form as in the previous model, except the q_0 and $B_j^{(1)}$ have different values.

THEOREM 8. *Given a queueing system with internal input(s) and its vacation time* V*, its setup time* U*, its mean input rate* λ*, and service time (to serve one tuple)* S*, the mean waiting time of tuples at this queueing system under both gated-service and exhaustive service disciple is given by*

$$\begin{aligned} W_q^{(1)} =& \frac{kN_i^C\sigma_i - 1}{2} S_j + \frac{(2-\sigma_i)N_i^C - 1}{2} S_i \\ &+ \frac{(k_i^{(1)} - 1)}{2} C_i^{(1)} + (V_j^{''(1)} + U_j^{(1)}) \end{aligned} \tag{3.52}$$

Proof. Considering the p^{th} tuple arriving during the l^{th} batch (busy period of its child operator) shown in Figure 3.6, the waiting time w_l^p of this tuple at operator j consists of:

1. the sum of service times to serve all the arrived tuples in first $l-1$ batch inputs, and the first $p-1$ tuples in the l^{th}batch. The number of tuples output from its child operator during each batch $N_i^O = N_i^C\sigma_i$.
2. the time for its child operator i to output(serve) the rest $N_i^C - \frac{p}{\sigma_i}$ tuples in l^{th} batch;
3. $k-l$ cycle periods because the operator j will be served after its child operator i has been served k times;
4. the *post-input vacation period* and itself setup time.

In summary,

$$\begin{aligned} w_l^p =&(l-1)N_i^O S_j + (p-1)S_j + \\ &(N_i^C - \frac{p}{\sigma_i})S_i + (k-l)C_i + V_j^{''} + U_j \end{aligned}$$

The total waiting time W^{total} of all tuples arriving in one cycle time of operator j is $W^{total} = \int_{l=1}^{k}\int_{p=1}^{N_i^C} w_l^p \; d_l d_p$. The mean waiting time of a tuple at the operator j is derived by dividing the total waiting time by the total number of tuples,

$$\begin{aligned} W_q^{(1)} =& \frac{kN_i^C\sigma_i - 1}{2}S_j + \frac{(2-\sigma_i)N_i^C - 1}{2}S_i \\ +\frac{(k_i^{(1)}-1)}{2}C_i^{(1)} + (V_j^{''(1)} + U_j^{(1)}) \end{aligned} \tag{3.53}$$

□

Remark 4. *1*) For two internal inputs, the operator j has inputs from both the left child operator i and the right child operator $i+1$. Then we have $W_j^{(1)} = \varphi_i W_j^{i(1)} + \varphi_{i+1} W_j^{i+1(1)}$; where φ_i, φ_{i+1} are the weight of the left input and right input respectively, $\varphi_i = 1 - \varphi_{i+1} = \frac{\sigma_i N_i^{(1)}}{\sigma_i N_i^{(1)} + \sigma_{i+1} N_{i+1}^{(1)}}$. 2) The mean tuple latency in above Theorem 8 can be decreased if we decrease the post-input vacation period. This tells us that we need to schedule an operator immediately after its child operators are scheduled in order to decrease the tuple latency in a DSMS.

Queue size: According to the Little's formula $Q^{(1)} = \lambda_j W_q^{(1)}$, we have the mean queue size $Q^{(1)}$ in a system described in Theorem 8 as follows:

$$Q^{(1)} = \frac{N_j^{(1)}}{C_j^{(1)}} W_j^{(1)} \tag{3.54}$$

3.4.3 Queueing Model with External and Internal Inputs

In this model, the queueing system has at least one internal input and at least one external input. The internal input is neither a Poisson process nor a continuous stream as explained in Section 3.4.2. The prototype of this queueing model is a *JOIN* operator with two inputs; one is the output of an operator, another is an external data stream. We decompose such a queueing system into two sub-queueing systems: (a) the queueing system that only has an internal input, which is modeled in Section 3.4.2; and (b) the queueing system that only has an external input, which is modeled in Section 3.4.1. Under both exhaustive-service and gated-service disciplines, the total number of tuples served during one cycle time is the sum of total number of tuples served during one cycle time in each of the decomposed queueing systems, therefore,

$$N_j^{C(1)} = N_j^{'C(1)} + N_j^{''C(1)} \tag{3.55}$$

Since the busy period is the time to serve those $N_j^{C(1)}$, it has the same form as (3.51). Similarly, the cycle time has the same form as in previous two models. The mean waiting time is the weighted sum of the mean waiting times of the tuples from the two sub-models. Again the weight is the ratio of the number of tuples output from a sub-model to the total number of tuples outputted from the model. The mean queue size can also be derived from mean waiting time through Little's formula.

3.4.4 Scheduling Strategy and Vacation Period

In this section we further discuss how to decide the weight ratio k of an operator in terms of its parent operator, and the lengths of both vacation period V and *post-input vacation period* V'' of an operator given a scheduling strategy.

In a multiple query processing system, various scheduling strategies have been proposed. They can be broadly classified into two categories: hierarchical scheduling and global scheduling. In hierarchical scheduling, the scheduling is done in a hierarchical way. At the top level, a system only needs to schedule the top-level objects such as a query plan, an operator path [77] and so on. Once a top-level object is scheduled, a lower-level scheduling strategy is employed to schedule the objects within a top-level object such as

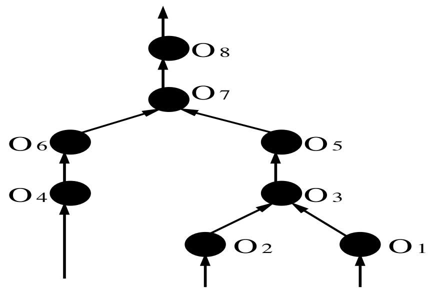

Figure 3.7 Push Up Operator Scheduling.

an operator. In global scheduling, only one scheduling strategy is needed, which schedules all the elementary objects and no composite object that consists of multiple elementary objects can be scheduled. For example, the Chain scheduling algorithm employs a global scheduling approach. Here we use the push up scheduling strategy and the weighted round robin scheduling strategy as examples to show how to derive the vacation periods.

Push up operator scheduling strategy: A push up operator scheduling strategy is a hierarchical scheduling strategy. From Remark 4, we found that the mean tuple latency can be decreased if we can schedule an operator as close as possible to its child operators. Based on this principle, a push up operator scheduling strategy employs a round robin strategy at the top level to schedule the query plans registered in the system. Therefore, the number of operators scheduled between an operator and its child operators is limited the number of operators in a query plan. To further limit the number of operators scheduled between the child operators of an operator and the operator itself, It employs a bottom-up approach to schedule all the operators of a query plan during each round. The operators at the bottom are scheduled first, followed by its parent operators. For a query plan with multiple operator paths, the operators at the same level are scheduled from right to left.

For example, the execution order of the operators of the query plan Ψ in Figure 3.7 is $\{O_1, O_2, O_3, O_4, O_5, O_6, O_7, O_8\}$. The weight ratio k is 1 under the scheduling strategy, and the vacation time for the j^{th} operator in the system, which is dedicated to query processing, is

$$V_j^{(1)} = \sum\nolimits_{i=1}^{m} (U_i^{(1)} + B_i^{(1)}) - (U_j^{(1)} + B_j^{(1)}) \tag{3.56}$$

where m is the total number of query plans registered in the system. From Section §3.4, we know that $B_j^{(1)}$ in all cases is a function of $V_j^{(1)}$, and we have m such equations with m unknown variables $V_j^{(1)}; j = 1, 2, \cdots, m$. These m equations are linearly independent, so we can simultaneously solve this set of equations to find $V_j^{(1)}$s.

Similarly, the *post-input vacation* $V_j^{''(1)}$ of an operator can be derived from (3.56), except that m is the total number of operators served during the period between the end point of the last input batch and the starting point to serve its tuples. In some scenarios, a processor has to take α percent[5] of its time to do non-query processing tasks, such as scheduling or system maintenance routines, so we have to multiply the equation (3.56) by a factor $\frac{1}{1-\alpha}$.

Weighted Round Robin scheduling strategy: A weighted round robin scheduling strategy is considered as a global scheduling strategy. The whole query processing system is treated as a pool of individual operators, and a weight is assigned to each operator to decide how often the operator is scheduled in the system compared with other operators. By considering a query processing system with m operators along with their weights[6] $\beta_1, \beta_2, \cdots, \beta_m$ respectively, statistically, we can infer that the operator j, which has its child operator i in the system, will be scheduled once whenever the operator i is

[5]which is not negligible. Only if a system is a dedicated query processing system, this α percents is very small and is negligible.

[6]The weight of each operator can be learned from the system through analyzing the scheduling strategy or collecting statistics from an actual system.

scheduled $\frac{\beta_i}{\beta_j}$ times, which is equivalent to the weight ratio k_j of operator j. The mean vacation time of the operator O_j is

$$V_j^{(1)} = \sum_{k=1}^{m} \frac{\beta_k}{\beta_j}(U_i^{(1)} + B_i^{(1)}) - (U_j^{(1)} + B_j^{(1)}) \tag{3.57}$$

There are m equations with m unknown variables, and the same approach used earlier to solve the equations defined by (3.56) can be used here. The approach is also applicable to find *post-input vacation time* $V_j^{''(1)}$ of the operator j.

The analysis that we have presented so far is generally applicable for a large family of scheduling strategies, such as Chain scheduling strategy, Path capacity strategy, round robin Operator strategy, and so on. The weight ratio k can be learned through collecting statistics or through analyzing the scheduling strategy.

3.4.5 Total Mean Queue Size and Total Mean Response Time

Once we find the vacation time and the post-input vacation time of each operator in a query plan, we are able to compute the mean queue size of each queue, and the mean tuple latency of each queueing path of each query execution plan in the system. Substituting these values into (3.1) and (3.2), the total memory requirement and the mean tuple latency of a query plan are obtained.

3.4.6 Discussion

We highlight some intuitive aspects of our observations below.

Tuple latency: *There are two ways to decrease the tuple latency in a DSMS without changing configuration of the system:*

1) clustering n, where $n > 1$, small systems into one larger system;

2) decreasing the number of operators scheduled between the child operators of an operator and the operator itself.

Although it can not decrease the memory requirement by clustering n smaller systems into one bigger system for stream processing, the mean tuple latency in the bigger system is only one n_{th} of that in small systems as shown in Remark 2. Remark 4 shows that the mean tuple latency decreases as the post-input vacation time decreases. One way to decrease post-input vacation time is to decrease the number of operators scheduled between the child operators of an operator and the operator itself.

Service discipline: *By using an exhaustive-service discipline, the query can achieve a better tuple latency than using a gated-service discipline under the non-null vacation queue situations.*

From our analysis of both the exhaustive-service discipline and the gated-service discipline in Section §3.4.2, we found that the queueing system has the same cycle time and the same number of tuples on average are served for these two service disciplines, but the tuple latencies under these disciplines are different. Under the non-null vacation queue situation[7] where $q_0 = 0$, the mean tuple latency of an operator under the exhaustive-service disciplines is $\frac{U^{(1)}+\rho V^{(1)}}{1-\rho}$ less than that under the gated service discipline. It is a positive value, and the difference increases dramatically when the system load increases or its vacation time increases.

Similarly, a service discipline in which the larger number of tuples is processed in busy period, the less of tuple latency is. It is also means that for a N-limited service discipline, a limit of processing 100 tuples in its busy period introduces a better tuple latency than a limit of processing 50 tuples in each busy period.

In general, an exhaustive-service discipline is better than any kind of gated-service discipline such as N-limited service discipline in terms of mean tuple latency. However, in some cases, an exhaustive-service discipline may introduce a long cycle time, which

[7]In a moderately loaded query processing system, the probability that the input queue of an operator is empty after it returns from vacation is very small, and can be ignored.

causes a long vacation time for some query plans. To solve this problem, we need to assign a weight to each query, and a N-limited service discipline is used for a query plan which has a long cycle time. Although the tuple latencies of the other query plans decrease, the mean tuple latency of the query plan with long cycle time increases.

Scheduling Algorithm: *The hierarchical scheduling approach with a push up operator strategy is better than any global scheduling approach under the exhaustive- or gated-service discipline.*

Consider the i^{th} query plan in a total of m query plans processing system. The total service time of all query plans except the i^{th} query plan is S during the steady state, and the total service time of all the operators in i^{th} query plan is S_i. In general, $S \gg S_i$. If a hierarchical scheduling approach (a query plan, or an operator path is treated as a schedulable object) with a push up operator strategy is used, the implication is that no operator from other query plans is interleaved with any operator in i^{th} query plan. (a) If the operator k has only one child operator, it is scheduled right after its child operator j is served. Therefore the *post-input vacation time* of the operator k is zero. (b) If an operator has two child operators, the *post-input vacation time* of one of its sub queueing system is not zero because only one operator can be scheduled at one time. However, the *post-input vacation time* is minimized. If a global scheduling approach (that is, an operator is a schedulable object) is used, it is highly possible that some operators, say p operators, are served between the service periods of two consecutive operators j, k of a queueing path of the i^{th} query plan. Consequently, the *post-input vacation time* of the operator k is the total service time of those p operators, which is larger than zero or at least equals to the service time of one of its child operators for an operator with two children. The vacation times under these two service disciplines are same. From (3.2), we can conclude that the overall tuple latency of the i^{th} query plan is larger under the global scheduling approach than under the hierarchical scheduling approach with a push up operator scheduling strategy.

In general, any operator path or query plan based scheduling strategy can achieve a better tuple latency than any scheduling strategy which uses an operator as a schedulable object.

Query Plan: *Among all the different implementations of a general query, the query plan which has a minimal total service time is better than the one which has minimal peak memory requirement but longer total service time in a multiple query processing system.*

If a query plan with a longer service time but a minimal peak memory requirement is chosen, all the other query plans in the system will have a longer vacation time due to its longer service time. The longer vacation time causes a longer tuple latency and increases the backlog of the tuples of all query plans registered in the system. The total increase of the backlog of all the query plans in the system may significantly larger than the memory size it saves.

Input Rate: *Linear increase of all the input rates in a query processing system can decrease the overall performance of the system dramatically (faster than a linear decrease).*

From §3.4, we know that with a linear increase in input rate of a queueing path, the service times of all the operators along that path increase linearly, which causes the vacation time of all the other query plans in the system to increase. As a result the service times of those operator paths increase due to a higher number of tuples arriving during their vacation periods. Consequently, the vacation time of the queueing path that increased its input rate rises, which causes its service time to rise further. Therefore, the overall tuple latency increases with a rate faster than linear, and the same is true for the queue size.

Similarly, the above statement holds for both selectivity and the number of query plans (system load) as well. Linearly increasing these factors causes the overall performance of the query processing system to decrease at a speed faster than the linear speed. Our experiments in the following section clearly testify to these statements.

3.5 Experiments

We conducted two sets of experiments to validate our theoretical analysis of both relational operators and continuous queries presented in this chapter. All experiments were run on an alpha-based dual-processor computer with an OFS1 (Tru Unix) V5.1 operating system, and 2Gb RAM. We were able to keep the tuples within the predefined window completely in main memory. Also there are no other user applications in the system when performing these experiments.

Input data streams: All data streams used in our experiments are synthetic network traffic data streams and consist of 7-field tuples : *(int sequenceId, int hostAdd, int networkAdd, int portNumber, int packSize, int protocolId, int serviceType).*

Once a tuple enters our system, a stream identifier, an arrival time-stamp, a departure time-stamp, and expired time-stamp are added for each tuple and the reference of the tuple is passed to the input queues of operators in the system for further processing. The arrival time stamp is the time of the tuple entering the system; the departure time-stamp is assigned when the tuple is outputted from the system; and the expired time stamp is assigned when the tuple enters the system based on the largest time window of all queries registered in the system.

The input data streams that we used are either a bursty data stream generated from a Poisson distribution or a more bursty stream that has so called self-similarity property. All data streams have a different mean input rate.

The input data streams with self-similarity property are highly bursty streams, which we believe resembles the situation in real-world applications. Each input stream is a super imposition of 64 or 128 flows. Each flow alternates ON/OFF periods, and it only sends tuples during its ON periods. The tuple inter-arrival time follows an exponential distribution during its ON periods. The lengths of both the ON and the OFF periods are generated from a Pareto distribution which has a probability mass function $P(x) = ab^a x^{-(a+1)}, x \geq b$. We use $a = 1.4$ for the ON period and $a = 1.2$ for the OFF period. For more detailed information about self-similar traffic, please refer to [85]. In our experiments for validating query plans, we use 5 such self-similar input data streams with different mean input rates.

Experimental operators and query plans: The operator used in validating our operator models is a symmetric hash-join operator. All of our queries used for validating query plan models are CQs consisting of *select, project*, and *symmetric hash join* operators. To be more close to a real application, we run 16 actual CQs with 116 operators over 5 different data streams in our system. The selectivity of each operator is widely distributed ranging from 0 to 1. Both the selectivity and the processing capacity of each operator are determined by collecting statistical information periodically during run time. This set of queries and their detailed properties are presented in Appendix A.

Tuple latency is measured by computing the difference between the arrival time stamp and the departure time stamp from either a queue for validating operator models or the DSMS system for validating query plan models. A large window size in our experiments is used in order to accurately measure tuple latency of a tuple in a DSMS. In OFS and most of current operating systems, the finest level of time unit is 10^{-3}seconds

Table 3.1 Delay & queue size (same window size)

Item	Exp1	Exp2
Input Rate(#/sec)	100	100
Utilization	0.9551	0.8595
Mean Delay(Theory)	0.05080	0.01257
Mean Delay(Exp)	0.05867	0.01539
Mean QueueSize(Theory)	10.6380	2.6313
Mean QueueSize(Exp)	10.6385	2.9399

or higher. If the actual tuple latency is less than one 10^{-3}seconds due to a small window size (also means small service time), the underlying operating system provide us nothing to measure the tuple latency and we will get a 0ms tuple latency. However, this limitation does not affect our analysis and our validation of our analysis.

3.5.1 Experiments for validating operator models

We conduct our first set of experiments to verify the theoretical analysis of hash-join operators presented in this chapter. In this set of experiments, the delay of a tuple, shown in Tables and Figures, is equivalent to the delay of tuples in the queue. We have conducted a wide range of experiments by varying the processing rates, window sizes, and input rates.

Same window size for two streams: This set of experiments measures the mean and CDF of the waiting time of a tuple in the queue under the same data stream input rates, and the mean number of tuples in the queue. In this case, we used two stream generators that send tuples to the system using the Poisson distribution with the same mean value. The window sizes for two data streams are one million tuples (about 10000 seconds), and the processing rates for both data streams are 209.4 tuples per second. The results presented here are obtained from the log file by deleting the first 2M records because the first 2M records are logged during the transition phase.

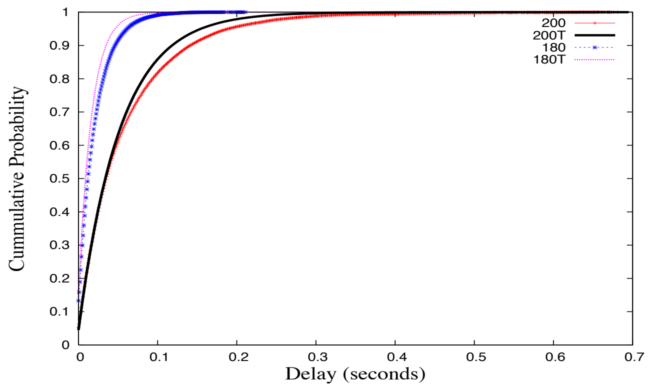

Figure 3.8 CDF Of Delay In Queue (Same Window Size).

Table 3.1 shows that the mean queue sizes from our experiments are very close to the theoretical results obtained from the second part of the equation (3.8) in Section §3.3.3.1. The mean delays are a little larger than the result calculated from equation (3.23b) because of the overhead associated with the recording of the departure time, and passing the reference to the hash join facility after dequeueing from the input queue. Figure 3.8 shows that the CDF of tuple latency in the queue is very close to our analysis results that were obtained from equation (3.24). The 3 digit numbers in the legend represent the total input rates (number of tuples per second), and the graphs with a 3 digit number following a letter 'T' represent the results of our theoretical analysis; otherwise they represent our experimental results.

Different window size for two streams: The following sets of experiments are done with different window sizes. We have 0.8 million tuples in hash table 1, and 0.4 million tuples in hash table 2. So the service time for a tuple that comes from the right stream is twice of the service time for a tuple that comes from the left stream. Specifically, the processing rates are 440 tuples and 220 tuples per second for left steam and right stream respectively.

Table 3.2 Delay & queue size (different window size)

Item	Exp1	Exp2	Exp3
Input Rate1(#/sec)	200	180	150
Input Rate2(#/sec)	100	90	75
Utilization	0.9091	0.8182	0.6812
Mean Delay(Theory)	0.01549	0.06276	0.00249
Mean Delay(Exp)	0.01931	0.00781	0.00291
Mean QueueSize(Theory)	5.1130	2.0710	0.8217
Mean QueueSize(Exp)	5.3817	2.1054	0.8438

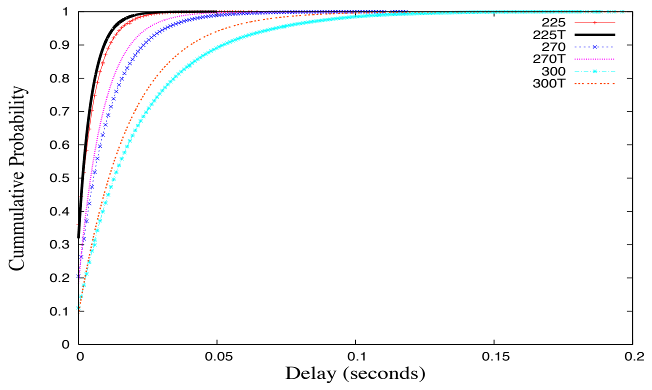

Figure 3.9 CDF Of Delay In Queue (Different Window Size).

Table 3.2 shows that the mean queue size as well as the mean delay is decreasing when the system utilization decreases. Also, the mean delay decreases when the service decreases with the same utilization. This indicates that we can either increase the processing ability of the system or decrease the input rates in order to achieve a better tuple latency. In our experiments, the longest delay of a tuple in the input queue can be more than 7 seconds if the utilization is very high and the service time is as low as 1/400 second. Another observation is that the delay of a tuple in the system primarily depends on the waiting time in the input queue if the utilization is high, and it depends on the service time of one tuple in the service facility under low utilization. Figure 3.9 shows the CDF of delay of tuples in the input queue under different window sizes as well as different input rates. The results show that there is a small probability for a tuple to wait in queue for as long as 10 times the mean waiting time. So it may not be difficult to meet the tuple latency for most tuples, but it is hard to achieve a 99% or higher confidence interval.

From our analysis and experimental results, we believe that to design a data stream processing system - especially for those that have a response time requirement - accurate estimation of the waiting time in the queue and service time is very important to determine the computational capability and the system capacity. The numbers of tuples in the input queue and in the service facility are negligible in terms of the number of tuples in the hash table, and the memory size mainly depends on the sliding time-window size.

3.5.2 Experiments for validating query plan models

In this set of experiments, we run an actual continuous query processing system, which consists of 16 actual queries with 116 operators over 5 different data streams on a dual-processor Alpha machine, where one processor is used exclusively for query processing and another is used for collecting data. Each experiment is started with a 3 hours transition phase[8], following a parameter collection phase in which we collect the various parameters for each operator such as processing rate, selectivity, setup time, weight ratio and so on. After that, the experiment enters a normal query processing phase which lasts about 5 to 8 hours. The results showed in this section are the mean values of various performance metrics we measured under a gated-service discipline. The results under an exhaustive-service discipline are slightly less than those under a gated-service discipline in terms of overall tuple latency of the whole query processing system. However, it does

[8]During a transition phase, the number of tuples in the window increases until the window is full, which causes the service time of a tuple increases; therefore, its processing rate is decreasing as the increase in the number of tuples in its buffers.

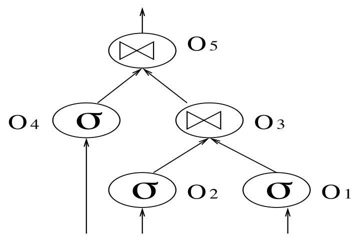

Figure 3.10 Target Query Plan.

Table 3.3 Parameters for Target Query Plan

Operator name	Processing rate(#/s)	Left selectivity	Right selectivity	Setup time
O_5	5894.1	0.308514	0.307712	7.4224E-4
O_4	28461.9	0.36534	-	1.1502E-5
O_3	5285.81	0.30703	0.300177	6.9477E-4
O_2	21861	0.487534	-	1.9599E-5
O_1	41684.7	0.24288	-	4.8044E-5

not give the system any choice to determine how many tuples it serves each round. In most query processing system, we have to control the service time allocated to each query or operator, therefore, a gated-based service discipline is employed in most of continuous query processing system, and the results under an exhaustive-based service discipline have a similar tendency as those under a gate-based service discipline presented in this subsection. And all results reported in this section are collected from our target query plan illustrated in Figure 3.10, through 16 CQs are active in the system. The results for other query plans share a similar tendency as those from our target query plan. From Little's formula, the number of tuples in a queueing system has a linear relationship with the mean tuple latency. Therefore, we only need to validate the tuple latency. The related parameters about this query plan are listed in Table 3.3.

Scheduling approach: Our first group of experimental results not only validates our analytical results, but also compares the performance of two different scheduling approaches in a multiple query processing system. The results presented in Table 3.4 are for a hierarchical scheduling strategy where we schedule all the queries in the system in a round-robin manner, and a push-up strategy is employed to schedule the operators of a chosen query plan. The results presented in Table 3.5 are for a global scheduling strategy in which we schedule one operator from our target query plan, and then schedule 3 other query plans. Both Table 3.4 and Table 3.5 present the theoretical results for our target query plan and the two sets of experimental results that we chosen from a set of experiments under different scheduling strategies. One represents the best results that we got, another represents the worst results. The theoretical results are derived by solving 116 linear equations that we get from Section §3.4.4.

The system loads in the experiments that presented in both tables are about 90% of its maximal capacity. First, these results show that our experimental results are close to our theoretical results because the difference is less than $\frac{0.00016}{0.1527} = 0.1\%$ in the best case, and no more than $\frac{0.04093}{0.19204} = 21.3\%$ in the worst case under a hierarchical scheduling strategy. The difference under a global scheduling strategy is less than $\frac{0.038356}{0.285155} = 13.5\%$ in the worst case. The average difference for all our experiments is less than 9.5%. When system load is less than 85% of its maximal capacity, the differences from these experiments are much better and are less than 3%. Second, the overall tuple latency in a hierarchical scheduling strategy is much less than that in a global scheduling strategy though the service times of one cycle under both strategies are same(they process the same number of tuples). The reason for that is the operators in our target query plan are scheduled in an interleaved manner, which causes all operators in our target query plan have a much larger post-input vacation time. However, the tuple latency for other query plans should be the same because all the other query plans are scheduled in a hierarchical

Table 3.4 Tuple Latency (seconds) Under Hierarchical Scheduling Strategy

OperatorName	O_1	O_2	$O_3(10^{-3})$	O_4	$O_5(10^{-3})$	Query Plan Latency(s)
Theoretical	0.14690	0.147114	2.3277	0.146988	2.64285	**0.15111**
Experiment A	0.14868	0.149151	2.4561	0.14665	2.5963	**0.15127**
Difference	0.00178	0.002037	0.1284	-0.000338	0.1284	**0.00016**
Experiment B	0.18849	0.188792	3.18175	0.186408	3.2960	**0.19204**
Difference	0.0019	0.041678	0.85405	0.03942	0.65315	**0.04093**

Table 3.5 Tuple Latency (seconds) Under Global Scheduling Strategy

OperatorName	O_1	O_2	O_3	O_4	O_5	Query Plan Latency(s)
Theoretical	0.147114	0.146114	0.079383	0.14690	0.075269	**0.246832**
Experiment A	0.152936	0.149876	0.083768	0.15308	0.076646	**0.257601**
Difference	0.005822	0.003762	0.004385	0.00618	0.001377	**0.010769**
Experiment B	0.169569	0.162786	0.092675	0.169223	0.084885	**0.285188**
Difference	0.022455	0.016672	0.013292	0.02233	0.009616	**0.038356**

scheduling strategy. Therefore, a hierarchical scheduling strategy is generally better than any kind of global scheduling strategy in which operators are scheduled in an interleaved manner.

System load: This group of experimental results shows how system load impacts the overall performance of a query plan. The system load can be increased by either increasing the number of queries in the system or increasing input rates of data streams. In this set of experiments, we increase our system load by increasing mean input rates of all data streams. The maximal capacity of the system is to process 485 tuples/per second on average. The system load is considered as $\frac{total_input_rates}{maximal_capacity}$. From Figure 3.11, we can see that the tuple latency of our target query plan increases slowly when system load is small. However, when system load reaches 95% of its maximal capacity, the tuple

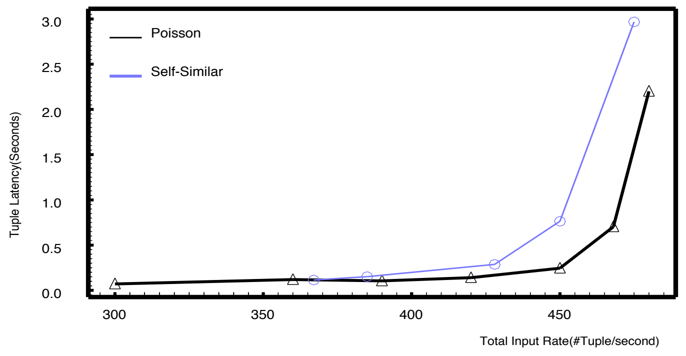

Figure 3.11 Tuple Latency vs System Load.

latency increases sharply as the increase in input rates of all data streams. As the system load approaches to 1, the tuple latency increases to infinity.

Bursty input data: This set of experiments are conducted to show how bursty input streams impact the system performance (tuple latency). The bursty input streams we used are self-similar input streams.

In this set of experiments, we use 5 self-similar input data streams with different mean input rates. The results in Figure 3.11 show that the difference between the tuple latencies from our Poisson input streams and the tuple latencies for self-similar input streams are very small when the system load (measured by input rates) is not too high (less than 95% of its maximal capacity). The results showed here are mean values. However, the maximal tuple latency and the variance of the tuple latency from self-similar inputs are much higher than those from our Poisson input streams. Only when the system load closes to its maximal capacity, the tuple latency that we get from our Poisson model is optimistic for a highly bursty input such as a self-similar input.

3.6 Summary

In this chapter, we addressed the problem of prediction of QoS metrics of continuous queries in a general DSMS. Those QoS metrics provide fundamental quantitative information and basis for a DSMS to manage, control, and deliver its QoS requirements. For example, the predicted QoS metrics can be used to assist a DSMS to determine when to activate or deactivate different QoS delivery mechanisms. Also the model and results presented in this chapter are general and are useful for any DSMS. Although our modeling and analysis work are based on relational operators in a DSMS, our approach can be extended to any user-defined operators as long as we can learn the processing capacity and selectivity of those user-defined operators. We can always learn these parameters through collecting run-time statistics.

By modeling individual operators on a dedicated server, we analyzed, in detail, the mean number of tuples, the mean waiting time of tuples in the queue and the distribution of the number of tuples, the distribution of waiting time of tuples at that operator. Furthermore, by modeling a general *Select-Project-Join* query over streaming data using a queueing network, we analyzed both memory requirement and tuple latency in the system under both gated- and exhaustive-service disciplines. We further discussed how the scheduling strategy, system load, operator selectivity, and input rates impact the performance of a general query processing system. The experiments based on our implementation of a query processing system clearly validate the accuracy and effectiveness of our analysis.

As part of our QoS management framework, we are investigating how the estimation techniques presented in this chapter can be combined with scheduling strategies and approximation techniques to guarantee the predefined QoS requirements in the system. In addition, we are extending our analysis to bursty input rates such as Long-Range Dependence (LRD) process to see how bursty input impacts the performance of query plans. Other problems that are currently under investigation include optimizing the overall performance of multiple queries, and system capacity planning and provisioning.

CHAPTER 4

SCHEDULING STRATEGIES

In this chapter, we focus on run-time resource allocation (i.e., scheduling strategy) problem for stream data processing, which determines the order in which operators or operator paths are scheduled at each time slot in a multiple CQ processing system.

The long-running characteristic of CQs, in contrast to one-time ad-hoc queries in DBMSs, makes resource allocation – especially scheduling – necessary in a multiple CQ processing system over data streams. The infinite input-size characteristic of input data streams further makes the resource allocation problem in DSMSs an on-line problem since it has to make decisions before seeing all of its inputs. The irregular and bursty input characteristics of data streams and the near real-time response requirements from stream-based applications further require DSMSs to carefully allocate the limited resources in the system. Improper resource allocation can cause DSMSs o fail in handling temporal bursty inputs in data streams and in providing on-time responses. As we will show in this chapter, improper resource allocation can cause delayed responses which are not acceptable for some applications, and cause the required maximal memory in a DSMS to exceed its physical available amount of memory, which can cause a system crash in the worst case. However, these failures can be avoided with proper resource allocation mechanisms. It is clear that the resource allocation mechanisms – one of which is judicial scheduling – is necessary in DSMSs and it is critical for the successful application of a DSMS.

The scheduling problem in a DSMS is as complicated as it is important. First, it has significant impact on the performance metrics of the system, such as tuple latency, maximal memory requirement, and system throughput. We will define these performance metrics in the next section. Second, although the resources such as memory size and CPU speed are fixed, scheduling can be highly dynamic. Third, various predefined QoS requirements for a query add additional constraints to an already complex problem. Finally, the issue is a complicated one because the problem of finding the schedule that only minimizes the memory required is NP-complete as shown in [21]. On the other hand, a desirable scheduling strategy in a DSMS should be able to: *1*) achieve the maximal performance within the fixed amount of resources; *2*) be aware of the unexpected overload situations, and take corresponding actions in a timely manner; *3*) guarantee user- or application-specified QoS requirements for a query, if any; and *4*) be implemented easily, and run efficiently with a low overhead.

A single scheduling strategy may not be able to satisfy all of the above properties, as there are trade-offs among these performance metrics and usages of the limited resources. Several scheduling strategies have been proposed for minimizing the maximal memory requirements in the literature. For stream applications, tuple latency is another important measure that is used in QoS (Quality of Service) specifications. In this chapter, we develop several scheduling strategies: *1*) the *path capacity* (PC) strategy to achieve the best overall tuple latency; *2*) the *segment* strategy to achieve lower maximal memory requirement than the PC strategy and better overall tuple latency than all operator-based strategies, such as an operator-level Round Robin strategy, the Chain strategy [21], and others; *3*) the *memory optimal segment* (MOS) strategy, which achieves the optimal memory requirement and improves upon the memory requirement of the Chain strategy, which has a near-optimal memory requirement; *4*) the *simplified segment* (SS) strategy, which requires slightly more memory but much smaller tuple latency than the segment strategy; *5*) the *threshold* strategy, which is a hybrid of the PC strategy and the MOS strategy. These suite of strategies provides sufficient choices to allow one to choose a strategy that is appropriate for an application. They provide reasonable overall performance although they do not meet all the desirable properties. The predefined QoS requirements of a query have not been incorporated into these strategies, which is part of our future work.

The rest of the chapter is organized as follows. Section 4.1 provides a detailed discussion of our scheduling model, and a summary of notations used in this chapter and remaining chapters. Section 4.2 introduces some preliminary scheduling strategies and shows the impact of a scheduling strategy on performance (i.e., maximal memory requirement, tuple latency, throughput, and so on) of a DSMS. In Section 4.3, we first propose the PC strategy, and then compare the PC strategy with the Chain strategy. We then discuss the segment strategy, its variants (the MOS strategy and the SS strategy), and the threshold strategy. Section 4.4 provides discussion related to the introduced strategies. Section 4.5 presents our quantitative experimental results, detailed analysis, and comparison with theoretical results. We summarize our scheduling work in Section 4.6.

4.1 Scheduling Model, Assumptions, and Notations

In a DSMS, a CQ plan is decomposed into a set of operators (as in a traditional DBMS) such as *project, select, join*, and other aggregate operators and a sequence of those operators form operator paths or segments. Therefore, from a scheduling point of view, a query processing system over data streams consists of a set of basic operators, or operator segments, or operator paths. Detailed algorithms are introduced in Section 4.3.3 to partition an operator path (OP) into operator segments based on a set of given criteria.

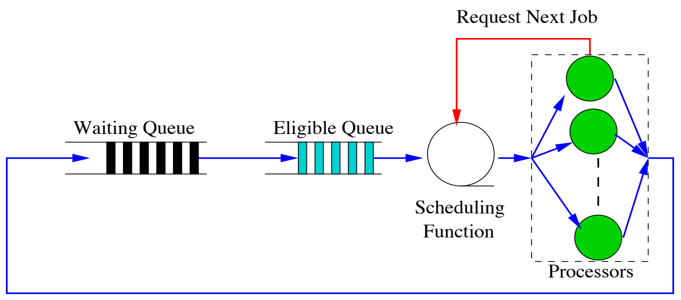

Figure 4.1 Scheduling Model.

4.1.1 Scheduling Model

As mentioned earlier, a multiple query processing system consists of a set of schedulable objects, which can be query plans, operator paths, operator segments, operators, and others, a scheduling function, and a set of job executors (i.e., processors/processes/threads), as illustrated in Figure 4.1, from a scheduling point of view.

In order to minimize the overhead introduced by scheduling itself, we employ an event-driven and preemptable scheduling model. In this scheduling model, we try to minimize the frequency of calls to the scheduling function. Although the cost of calling the scheduling function is small, frequent calls of the scheduling function can introduce considerable overhead in a DSMS. This is because: *1*) the total cost of frequent calls to the scheduling function can be high; *2*) the context switching cost can also be high due to instruction cache miss [126]. For example, if an operator only processes one tuple each time it is invoked, many instruction cache miss will occur, which results in high context-switching overhead [126, 46].

A schedulable object is initially in the waiting queue and it is moved to the eligible queue only when some events occur (i.e., the number of waiting tuples in the input queues of an object in the waiting queue exceeds a predefined threshold value, an object in the waiting queue has been waiting longer than a predefined threshold value, the eligible queue is empty and one processor is idle, or others). The scheduling function schedules only the objects in the eligible queue and it is called to determine next executable job when either of the following events occurs:

1. when a new eligible job is inserted into the eligible queue from the waiting queue. In this case, the scheduling function selects one object to execute among all objects in the eligible queue and the running objects such that the selected object has the lowest property (i.e., priority, processing capacity, and others). If the selected object is the current running one, no change is made; otherwise, the selected object is scheduled and the running object is preempted. The preempted object is placed back to the eligible queue and the scheduling function is not called to select an object to execute only if the left unprocessed job in the input queue satisfies the criteria of moving an object from the waiting queue to the eligible queue; otherwise, it is placed back to the waiting queue. Notice that an object can be preempted only when the current processing tuple is processed completely by the running object, it cannot be preempted in the middle of processing a tuple.
2. when a processor finishes processing all inputs of its job and the eligible queue is not empty; In this case, the finished object is placed back to the waiting queue and the scheduling function is called to select one object to execute. If the eligible queue is empty and no objects in the waiting queue satisfy the criteria of moving an object from the waiting queue to the eligible queue, some objects in the waiting queue (i.e., the objects with largest number of tuples waiting in the input queues, the objects with oldest age in the waiting queue, or others) are moved to the eligible queue directly without checking moving criteria.

Even in a multiple processor architecture, no more than one processor will call the scheduling function to request the next executable object at the same instant. If more than one processor calls the scheduling function simultaneously, we process them serially in an arbitrary order. This can be done through a lock mechanism on the scheduling function.

Once an operator within the scheduled object is scheduled, the tuples waiting in its input queues can be processed in a different order. There is a need for another scheduling strategy to determine which tuple is processed first, we refer this scheduling strategy within an operator as operator-inner-scheduling strategy. If the operator is computed over a tuple-based window, the operator-inner-scheduling strategy does not affect the amount of memory needed to maintain synopsis of an operator in order to compute the operator correctly. However, if the operator is computed over a time-based window or semantic window, the operator-inner-scheduling strategy does affect the amount of memory needed for synopsis.

For a two or multiple-way join operator, a different order to process the input queues can cause different memory requirement to maintain synopsis information. For example, considering a join operator over a 5-minute sliding window over streams A and B. Suppose there is a burst of tuples that arrive on stream B. Then the synopsis of B that is needed to compute the join with A will be large, until A tuples that arrive 5 minuets after the burst has passed the join operator. A scheduling that chooses to schedule the join of A with B as early as possible will free up memory from B's synopsis as early as possible, whereas a strategy that choose to delay scheduling the join will result in the large synopsis staying around for a longer period of time, which possibly cause a larger maximal memory requirement.

In our model, we employ a First-In-First-Out (FIFO) scheduling strategy for operator-inner-scheduling strategy[1] since we would like to preserve the order of tuples during the processing. For an object with one input stream, this is straightforward. For an object

[1]How different operator-inner- scheduling-strategy affect the maximal memory requirement and other metrics is beyond the scope of this work.

with two or more input streams, we maintain one input queue with different classes of tuples and tuples from different input streams are placed in the input queue based on their arrival time stamps. When the object is scheduled, the tuples in its input queue are processed based on their orders in the queue and the time stamp of an output tuple is the time stamp of the tuple from the input queue, not the tuple from synopsis.

Based on our FIFO operator-inner-scheduling strategy and the same assumption in [21], we can assume that the runtime state or synopsis information stored by each operator is fixed in size and, therefore, the variable portion of the maximal memory requirement is derived from the sizes of the input queues to operators. We also assume that the root node consumes its inputs immediately after they enter queues. Therefore, there is no tuple waiting in the input queues of the root node, and the root node is simply treated as a sink in this chapter.

4.1.2 Notations

To facilitate our analysis, we use the following notations.

- *Maximal memory requirement:* the maximal amount of memory consumed by the tuples waiting in the input queues of all operators at any time instant in the system. The memory requirement in this chapter means the maximal memory requirement unless specified.
- *Tuple latency:* the length of time an output tuple stays in the CQ processing system after it enters the system. The tuple latency for an output tuple that only involves unary operators is straightforward. For a join operator or multiple-way operator, as we mentioned earlier, such an operator is computed by getting a tuple from its input queue and then computing the operator with the stored synopsis information. The arrival time stamp of an output tuple from such an operator is derived as the arrival time stamp of the tuple from the input queue. Since we maintain a virtual queue

for a join operator or multiple-way join and employ a FIFO scheduling strategy for the operator-inner-scheduling strategy, the time stamp of the tuple from the queue is also the latest time stamp comparing with those in the stored synopsis. The tuple latency is computed as the difference between the departure time stamp of an output tuple and its arrival time stamp. Although different systems may use different ways to compute the tuple latency, the scheduling strategies proposed in this thesis do not depend on the way of computing tuple latency. The overall tuple latency is the weighted average of the tuple latency of all output tuples in the system. The tuple latency in this chapter means the overall tuple latency unless specified otherwise.

- *Throughput:* the number of final tuples output from the query processing system per time unit.
- *Operator processing capacity* $C^P_{O_i}$: the number of tuples that can be processed within one time unit at operator O_i. Inversely, the operator service time is the number of time units needed to process one tuple at this operator. A join operator or k-way operator is considered as two or k semi-operators. Each of them has its own processing capacity, selectivity, and memory release capacity.
- *Operator selectivity* σ_i: it is the same as in a DBMS except that the selectivity of a join operator is considered as two semi- join selectivities.
- *Operator memory release capacity* $C^M_{O_i}$: the number of memory units such as bytes, pages that can be released within one time unit by operator O_i.

$$C^M_{O_i} = C^P_{O_i}(InputTupleSize - T_i\sigma_i) \tag{4.1}$$

where T_i is the size of the tuple output from operator O_i; the input tuple size from each input stream of a join or k-way operator is considered as the input tuple size of its corresponding semi- operator.

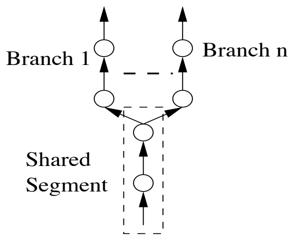

Figure 4.2 A Complex Operator Path/Segment.

- *Operator path processing capacity* $C^P_{P_i}$: the number of tuples that can be processed within one time unit by the operator path P_i. Therefore, the operator path processing capacity depends not only on the processing capacity of an individual operator, but also on the selectivity of these operators and the number of operators in the path.

  For a simple operator path P_i with k operators, its processing capacity can be derived from the processing capacities of the operators that are along its path, as follows:

$$C^P_{P_i} = \frac{1}{\frac{1}{C^P_{O_1}} + \frac{\sigma_1}{C^P_{O_2}} + \frac{\sigma_1\sigma_2}{C^P_{O_3}} + \cdots + \frac{\prod_{j=1}^{k-1}\sigma_j}{C^P_{O_k}}} \tag{4.2}$$

  where $O_l, 1 \leq l \leq k$ is the l_{th} operator along P_i starting from the leaf node. The denominator in (4.2) is the total service time for the path P_i to serve one tuple. The general item $(\prod_{j=1}^{h}\sigma_j)/C^P_{O_k}$ is the service time at the $(h+1)^{th}$ operator to serve the output part of the tuple from the h^{th} operator along the path, where $1 \leq h \leq k-1$.

  For a complex operator path P_i with k operators along its shared segment and with m branches with m_k operators along its branches as illustrated in Figure 4.2, its processing capacity can be derived from the processing capacity of the shared segment and the processing capacities of its m branches. The processing capacity of the branch B_i with i_k operators and its sharing segment can be computed from (4.2) by considering each of them as a simple path as follows:

$$C_i^B = \frac{1}{\frac{1}{C_{O_1}^P} + \frac{\sigma_1}{C_{O_2}^P} + \frac{\sigma_1\sigma_2}{C_{O_3}^P} + \cdots + \frac{\prod_{j=1}^{i_k-1}\sigma_j}{C_{O_{i_k}}^P}} \tag{4.3}$$

Therefore, the processing capacity of the complex operator path P_i is derived as follows:

$$C_{P_i}^P = \frac{1}{\frac{1}{C^{SharingSegment}} + \left(\prod_{j=1}^{k-1}\sigma_j\right) * \sum_{i=1}^{i<=m}\frac{1}{C_i^B}} \tag{4.4}$$

In above equation, the $\frac{1}{C^{SharingSegment}}$ part is the total processing time needed by the sharing segment to process one input tuple and the part $\left(\prod_{j=1}^{k-1}\sigma_j\right) * \frac{1}{C_i^B}$ is the total processing time needed by branch B_i to process what the sharing segment outputs by processing one input tuple.

Recursively, for a complex operator path with branches and branches having their own sub-branches, we first compute the processing capacity of any sub-branch, and then compute the processing capacity of each branch as (4.4) by considering each branch as a complex operator path, and finally compute the processing capacity of the complex operator path as (4.4).

- *Path memory release capacity* $C_{P_i}^M$: the number of memory units that can be released within one time unit by the path P_i. Again, in this chapter, we assume that all the output tuples from a query are consumed immediately by its applications. Therefore, no memory is required to buffer the *final* output results and the memory release capacity is simply what the operator path consumes per time unit, which is shown in (4.5).

$$C_{P_i}^M = C_{P_i}^P * InputTupleSize \tag{4.5}$$

From equation (4.5), we know that the processing capacity and the memory release capacity of an operator path differs with only a constant factor, which is the input tuple size. Therefore, we assume that the partial order between the processing capacities of two paths is the same as the partial order between their memory release capacities. We believe that this assumption is reasonable under a data stream processing environment. For instance, a CQ processing system that is used to analyze Internet traffic has the same tuple input size, where the tuples are the header of the Internet IP packets. Although the sizes of all input tuples from some applications may not be exactly the same, their differences are not large enough to change the relative partial orders of their operator paths. Hereafter, we use the path capacity to refer to both the processing capacity and the memory release capacity.

- *Segment Processing Capacity* $C^P_{S_i}$: the number of tuples that can be processed within one time unit by the operator segment S_i. And the processing capacity of a simple segment or a complex segment, illustrated in Figure 4.2, has the same definition as that given in (4.2) or (4.4) respectively.

- *Segment Memory Release Capacity* $C^M_{S_i}$: the number of memory units can be released within one time unit by the segment S_i.

  For a simple segment, its memory release capacity $C^M_{S_i}$ is defined as:

$$C^M_{S_i} = C^P_{S_i}\left(InputTupleSize - S_o * \prod_{i=1}^{k}\sigma_i\right) \tag{4.6}$$

  where S_o is the size of the output tuple from segment S_i. For the last segment, the size of the output tuple is zero because of the assumption that the output can be consumed by its applications immediately.

  For a complex operator segment S_i with k operators along its shared segment and with m branches with m_k operators along its branches as illustrated in Figure 4.2, its memory release capacity is derived as:

$$C^M_{S_i} = C^P_{S_i}\left(InputTupleSize - \left(\prod_{i=1}^{k}\sigma_i\right) * \sum_{j=1}^{j<=m}\left(S^j_o * \prod_{i=j_1}^{i<=j_k}\sigma_i\right)\right) \tag{4.7}$$

In the above equation (4.7), the item $\left(\prod_{i=1}^{k} \sigma_i\right)$ is the portion of one tuple output by the shared segment with m operators in Figure 4.2 by processing one input tuple; the item $\left(S_o^j * \prod_{i=j_1}^{i<=j_k} \sigma_i\right)$ is the total output size outputted by the j^{th} branch by processing one tuple waiting at the beginning of the branch. Therefore, the product of these two items is the total output size outputted by each branch by processing one input tuple. Similarly, the memory release capacity of a complex segment with branches and some branches have their sub-branches can be computed recursively.

4.2 Preliminary Scheduling Strategies

A DSMS has multiple input streams and if the input rate of each stream is trackable (i.e., it is known as to how many tuples will be arriving in future time slots), we can find an optimal scheduling strategy that can achieve the best performance with respect to a metric[2] by using the minimal resources. However, in most cases, the input of a data stream is unpredictable, and highly bursty, which makes it hard, or even impossible to find such a feasible, optimal scheduling strategy. In practice, heuristics-based or near-optimal strategies are usually used. And these strategies have different impact on the performance and the usages of the system resources. The Chain strategy [21] is a near optimal scheduling strategy in terms of total internal queue size. In addition to the memory requirement, tuple latency is another important metric for a stream query processing system. Both are especially important for a DSMS where its applications have to respond to an event in a near real-time manner. In the rest of this section, we use the FIFO strategy described below and the Chain strategy to show how a strategy impacts

[2]We mean tuple latency, throughput, etc in this chapter.

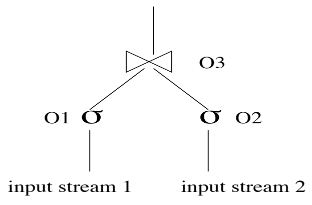

Figure 4.3 A Query Execution Plan.

Table 4.1 Operator Properties

Operator Id	1	2	3
Selectivity	0.2	0.2	0.8
Processing capacity	1	1	0.2

the internal queue size (memory requirement), the tuple latency, and the throughput of a CQ processing system.

FIFO Strategy: *Tuples are processed in the order of their arrival. Once a tuple is scheduled, it is processed by the operators along its operator path until it is consumed by an intermediate operator or output to the root node. Then the next oldest tuple is scheduled.*

Chain Strategy: *At any time, consider all tuples that are currently in the system; of these, schedule a single time unit for the tuple that lies on the segment with the steepest slope in its lowest envelope simulation. If there are multiple such tuples, select the tuple which has the earliest arrival time.*

The slope of a segment in the Chain strategy is the ratio of the time it spends to process one tuple to the memory size changed of that tuple. The tuple in the Chain strategy refers to a batch of tuples, instead of an individual tuple. For further details, refer to [21].

Table 4.2 Performance (F:FIFO, C:Chain)

Time	Input		Queue Size		Tuple Latency		Throughput	
	1	2	F	C	F	C	F	C
1	1	0	1.0	1.0	-	-	0	0
2	0	1	1.2	1.2	-	-	0.0	0
3	1	1	3.0	2.4	2	-	0.16	0
4	0	0	2.2	1.6	-	-	0	0
5	0	0	2.0	0.8	3	-	0.16	0
6	0	0	1.2	0.6	-	5	0	0.16
7	0	0	1.0	0.4	4	5	0.16	0.16
8	0	0	0.2	0.2	-	5	0	0.16
9	0	0	0	0	6	6	0.16	0.16
10	1	0	1.0	1.0	-	-	0	0

Let us consider a simple query plan illustrated in Figure 4.3, which is a common query plan that contains both *select* and *join* operators. The processing capacity and selectivity of the operators are listed in Table 4.1. The input streams are assumed to be highly bursty to accurately model stream query processing systems. Table 4.2 shows the total internal queue size, tuple latency, and throughput of the query plan under both the FIFO strategy and the Chain strategy for the given input patterns.

The results clearly show that the Chain strategy performs much better than the FIFO strategy for the total internal queue size. However, it performs much worse than the FIFO strategy in terms of tuple latency and produces a much bursty and irregular throughput than the FIFO. Clearly, The FIFO strategy maintains its entire backlog of unprocessed tuples at the beginning of each operator path. It does not take the inherent properties of an operator into consideration such as selectivity, processing rate, which causes the total internal queue size to be larger under the FIFO strategy than under the Chain strategy. In contrast, as the Chain strategy pushes the tuples from the bottom, it inherits the bursty property of the input streams. If the input streams are highly bursty in nature, the output of the CQ processing system under the Chain strategy demonstrates highly bursty property too, but the bursty input property in its output is partially determined by the selectivity of the operators and the system load. Therefore, FIFO has a smoother throughput and a better tuple latency than the Chain strategy.

Considering an input stream with a bursty period of 1M tuples and its input rate faster than the processing rate, how long do we have to wait to get the first result in a CQ processing system which has the maximal ability to completely process[3] 10000 input tuples per second? Under the Chain strategy, it is almost 100 seconds! Of course, the total internal queue size is much less than 1M tuples, which is the difference between the input rate and processing rate times the length of the bursty period without considering the changes of tuple size in the system. Based on this observation, we develop the PC strategy, which takes the tuple latency and the throughput as its primary priorities, the total internal queue size as its secondary priority.

4.3 New Scheduling Strategies

In this section, we first present the PC strategy, and then provide a thorough comparison with the Chain strategy. To overcome the non-optimal memory requirement of the PC strategy and to preserve its minimization of tuple latency property, we further propose the segment strategy and its variants – the MOS strategy and the SS strategy. Finally, we discuss the threshold strategy, which is a hybrid of the PC strategy and the MOS strategy.

4.3.1 Path Capacity Strategy

From 4.2, we observe that the FIFO strategy has two promising properties: reasonable tuple latency and throughput. But it does not consider the characteristics of an

[3]We mean the computation from reading in the input tuples to output the final results.

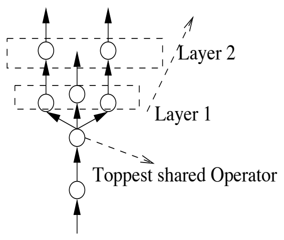

Figure 4.4 Bottom-up Scheduling Strategy.

operator path such as processing capacity and memory release capacity. Also as it schedules one tuple each time, the scheduling overhead is considerably high. This motivates us to develop the PC strategy which improves upon the high memory requirement of the FIFO strategy, and has better tuple latency and smoother throughput than the FIFO strategy.

Path Capacity Strategy: *At any time instant, consider all the operator paths that have input tuples waiting in their queues in the system, schedule a single time unit for the operator path with largest processing capacity to serve until its input queue is empty or there exists an operator path which has a non-null input queue and a larger processing capacity than the currently scheduled one. If there are multiple such paths, select the one with the largest processing capacity. If there are multiple paths with the largest processing capacity, select one arbitrarily* [4]*. The following bottom-up operator scheduling strategy is used to schedule the operators of the chosen path.*

Bottom-up operator scheduling strategy: Once an operator path is chosen, a bottom up approach [5] is employed to schedule all the operators along the chosen path. For

[4]Due to the high cost to keep track of the oldest tuple in the queues, we do not schedule the operator path with the oldest tuple in its input queue when there exists multiple operator paths with the largest processing capacity.

[5]this thesis only discusses bottom-up scheduling strategies. Work on pipelined strategies is being investigated.

a simple operator path, the order of operators scheduled by this bottom up scheduling strategy is straightforward. For a complex operator, illustrated in Figure 4.4, in a multiple query processing system, the order of the operators along the sharing segment of the complex OP is scheduled from bottom to up. For the operators along the branches of the complex OP, those operators are categorized into layers based on their distances to the topmost shared operator as illustrated in Figure 4.4. The operators on a lower layer are scheduled earlier than those in a higher layer. For the operators in the same layer, the operator with largest processing capacity is scheduled first and an arbitrary one is scheduled if two operators have the same processing capacity.

Once an operator path (OP) is scheduled, it will finish processing all tuples in its input queue or be preempted by another OP with larger capacity. The PC strategy is a static priority scheduling strategy. The priority of an operator path is its processing capacity, which is completely determined by the number of operators along its path, and the selectivity and the processing capacity of each individual operator. The priority of an operator path does not change over time until we revise the selectivity of operators. Therefore, the scheduling cost is minimized and can be negligible. Most importantly, the PC strategy has the following two optimal properties that are critical for a multiple CQ processing system.

THEOREM 1. *The path capacity strategy is an optimal one in terms of the total tuple latency or the average tuple latency among all scheduling strategies.*

Proof. First, operator path-based scheduling strategies, where an operator path is the finest schedulable object (for example, operator path- based round robin strategy) have a better tuple latency than those which do not use an operator path as a scheduling unit. In [73], we showed that if the operators of two query plans or operator paths are scheduled in an interleaved manner, the overall tuple latency becomes worse. The operators of two operator paths under the path-based strategy are never scheduled in an interleaved manner, therefore, the PC strategy has a better tuple latency than any non-path based scheduling strategy. Second, the PC strategy has a minimized tuple latency among all path-based scheduling strategies. At any time instant, consider k operator paths $p_1, p_2, \cdots, p_k$, which have $N_i \geq 1, i = 1, 2, 3, \cdots, k$ tuples in their input queues in the system, with their capacities $C_1, C_2, \cdots, C_k$ respectively. Without loss of generality, we assume that $C_1 \geq C_2 \geq \cdots \geq C_k$. The PC strategy has a schedule of $p_1, p_2, \cdots, p_k$, that is to serve the N_1 tuples of operator path p_1, following the N_2 tuples of operator path p_2 , and so on. In the simplest case where $N_i = 1, i = 1, 2, 3, \cdots, k$, the total tuple latency $T = k\frac{1}{C_1} + (k-1)\frac{1}{C_2} + \cdots + (k-g+1)\frac{1}{C_g} + \cdots + (k-h+1)\frac{1}{C_h} + \cdots + \frac{1}{C_k}$, where $(k-i)\frac{1}{C_i}, i = 0, 1, \cdots, k-1$ is the total waiting time of all the tuples in the system due to processing the tuple at operator O_i. If we switch any two tuples (two paths), say g, h, where $g < h$, in the PC strategy, then the total tuple latency $T' = k\frac{1}{C_1} + (k-1)\frac{1}{C_2} + \cdots + (k-g+1)\frac{1}{C_h} + \cdots + (k-h+1)\frac{1}{C_g} + \cdots + \frac{1}{C_k}$. The difference of two tuple latency$\Delta = T - T' = (h-g)\left(\frac{1}{C_g} - \frac{1}{C_h}\right) \leq 0$ because of $g < h$ and $C_g \geq C_h$. Similarly, for the general case, by switching any two tuples in two input queues of these k operator paths, we still have $\Delta \leq 0$. Therefore, any other scheduling strategy causes at least the same total tuple latency or mean delay as the PC strategy causes. □

THEOREM 2. *Any other path-based scheduling strategy requires at least as much memory as that required by the PC strategy at any time instant in the system.*

Proof. At any time instant, the PC strategy schedules the tuples waiting in the input queue of the operator path which has the largest capacity among all the paths with non-empty input queues in the system. Within one time unit, the path scheduled by the PC strategy consumes the maximal number of tuples because it has the largest capacity. Any other path based scheduling strategies (for example, path-based round robin) which do not schedule the tuples waiting in the input queue of the operator path with the largest capacity at that time instant consume less number of tuples. Therefore, any other path based scheduling strategy requires at least the same amount of memory required by the PC strategy. □

Theorem 1 clearly shows that the PC strategy is the optimal one in terms of total tuple latency and it performs much better than the FIFO strategy and the Chain strategy for tuple latency. Theorem 2 shows that the PC strategy performs better than any other path-based strategy, but not as well as the Chain strategy in terms of the memory requirement.

4.3.2 Analysis of Scheduling Strategies

Both the PC strategy and the Chain strategy have their optimal properties as well as shortcomings. In this section, we will present a comprehensive comparison and show how these two scheduling strategies impact the various performance metrics of a CQ processing system. A quantitative experimental study for these two scheduling strategies will be presented in section 4.5.

Tuple latency & throughput: The PC strategy can achieve the optimal tuple latency as compared to any other scheduling strategy and it also has a much smoother output rate than other strategies. The main reason for the large tuple latency in the Chain strategy is that the leaf nodes usually have a much larger capacity than other nodes of a query plan, which causes the Chain strategy gradually to push all the tuples from the leaf nodes toward the root node, and a large number of tuples are buffered along an operator path. This situation becomes even worse during a temporary overload period in which the input rates temporarily exceed the processing capacity of the system. All the computational resources are allocated to these operators at the bottom of a query plan during the bursty input periods[6] and there is almost no throughput from the system. On the other hand, the throughput is surprisingly high immediately after the highly bursty period because there are not too many tuples waiting at leaf nodes, and most of the computation resources is available for the operators in the upper part of the query plans where a large number of partially processed tuples wait. As a result, the Chain strategy not only has a bad tuple latency, but also a bursty output rate if the input streams are bursty. The bursty output rates may negate part of its saved memory because the consumed rates of applications cannot keep up with the bursty output rates, which causes portion of the results to backlog in the query processing system for a while.

Memory requirement: Both strategies have an optimal property in terms of the memory requirement. But the optimal property of the PC strategy is a relative one among all path-based scheduling strategies, while the near optimal property of the Chain strategy is a global optimal property. Under non-overload conditions, the amount of memory required by these two strategies is similar, and there are not too many tuples buffered in the queues. However, during bursty input periods, the Chain strategy performs better than the PC strategy because the PC strategy buffers the unprocessed tuples at the beginning of an operator path.

Starvation: Both strategies have the starvation problem in which some operators or operator paths may never be served because both of them depend on a set of static priorities. Under heavy load situations, the Chain strategy spends most of its computation resources on the bottom- side operators of an operator path; as most of operators in the upper side of an operator path (closer to the root) have lesser capacity, they are likely to starve. On the other hand, as the PC strategy spends most of its computation resources on the operator paths with a larger path processing capacity, the operator paths with less capacity are likely to starve. One significant difference is that during heavy load

[6]Especially, the bursty period is relatively long if the input streams have self similar property.

situations, the Chain strategy has very small or even no throughput at all , whereas the PC strategy still has reasonable throughput. The starvation problem in the Chain has been overcome in its improved version, called Chain-Flush strategy [21], with the cost of introducing a possible larger maximal memory requirement and larger overhead. The technique introduced in the Chain-Flush can be applied to the PC strategy as well as other strategies with the starvation problem. In addition, we also proposed techniques to overcome the starvation problem, which will be discussed in §4.4.

Scheduling overhead: Clearly, both strategies have very small scheduling overhead because both are static priority strategies. But the scheduling overhead incurred by the PC strategy is less than that incurred by the Chain strategy because the number of operator paths in a system is less than the number of operators. In our query processing model, the number of OPs is equal to the number of input data streams. Although the cost of scheduling one operator or operator path is very small, the cost to process one tuple is even smaller than that. Therefore, the number of tuples served by each schedule has significant impact on the performance of a CQ processing system as we discussed in Section §4.1.1. If a small number of tuples is processed when an operator is invoked, (*1*) the ratio of the scheduling cost to the cost of processing actual tuples can be large, which causes the system to spend considerable portion of resources on scheduling; (*2*) the context switch cost can also be high due to many cache instructions miss as reported in [126].

To overcome the overhead introduced by scheduling itself, a number of techniques are proposed in the literature and can be applied to the scheduling strategies proposed in this thesis. (*1*) Considering a tuple as a fixed memory unit such as page, rather than an individual tuple as proposed in [21]; (*2*) An event-driven scheduling model as we discussed in Section §4.1.1 is used, instead of making a scheduling decision at every time unit. In addition, we are studying a non-preemptive scheduling model to further decrease the need for making a scheduling decision and to increase the number of tuples processed once an operator is invoked. In this non-preempted scheduling model, an object is eligible to be scheduled/invoked only when the number of tuples exceeds a predefined minimal value, say its processing capacity. Once the object is scheduled/invoked, the object is non-preempted until all jobs have been processed or the time units used have exceeded a predefined maximal number of time units. In a practical system, we can either an exhaustive service discipline [73] or a gated service discipline in order to further decrease the overall scheduling overhead.

Context switching overhead: When each operator is implemented as a single thread, the context switching cost incurred by a scheduling algorithm is considerably high. The performance of the system will degrade dramatically as the number of operators increases. It is beneficial to implement the entire query processing as a single thread or as a few threads over a multiple-processor system to keep the switching overload low. Indeed we have implemented the whole query processing system as a single thread, and hence the cost of switching from one operator path to another is just a function call, which is quite low in a modern processor architecture. As a batch of tuples, rather than one tuple, is scheduled for each scheduling round, the cost of making a function call is negligible as compared with the cost of processing a batch of tuples.

4.3.3 Segment strategy and its variants

Although the PC strategy has optimal memory requirement among all path- based scheduling strategies, it still buffers all unprocessed tuples at the beginning of an operator path. In a CQ processing system with a shortage of main memory, a trade-off exists between the tuple latency and the total internal queue size. Therefore, we develop the segment strategy which has a much smaller total internal queue size requirement as compared to the PC strategy, and a smaller tuple latency than the Chain strategy.

Furthermore, we introduce two variants of the segment strategy: the MOS strategy, which achieves the strict minimization of maximal memory requirement theoretically comparing with the near-optimal memory requirement property of the Chain strategy, and the SS strategy (a special case of the segment strategy) which further improves tuple latency with a slightly larger memory requirement than the segment strategy.

The segment strategy and its variants employ an idea that allows us to improve upon the PC strategy in terms of maximal memory requirement. Operator scheduling and path scheduling can be seen as two extremes of the spectrum, whereas segment strategies cover the points in between. Instead of buffering the unprocessed tuples at the beginning of an operator path, we partition an operator path into a few segments, so that some partially processed tuples can be buffered at the beginning of a segment. This allows the system to take advantage of the lower selectivity and fast service rate of bottom side operators of a query execution plan. The processing capacity and the memory release capacity for a segment are defined in Section §4.1.2.

The segment strategy employs the same scheduling model as the PC strategy. It schedules an operator segment, rather than an operator path like the PC strategy.

Segment Scheduling Strategy: *At any time instant, consider all the operator segments that have input tuples waiting in their input queues. Schedule a single time unit for the operator segment which has the maximal memory release capacity to serve until its input queue is empty or there exists another operator segment which has a non-null input queue and a larger memory release capacity than the currently scheduled one. If there are multiple such segments, select the one with the largest memory release capacity. If there are multiple segments with the largest memory release capacity, select one arbitrarily. The bottom-up operator scheduling strategy (described earlier) is used to schedule the operators of the chose segment.*

The key component of the segment strategy and its variants is the algorithm to partition an operator path into segments. We proposed three algorithms to partition an OP into segments. Those algorithms work for both simple operator paths and complex operator paths. We focus on discussing the details of our segment construction algorithms when we present our segment strategy and its variants in the following subsections.

4.3.3.1 Segment Strategy

The segment strategy, also termed greedy segment strategy, is a static priority driven strategy and it employs the following greedy segment construction algorithm to partition an OP into segments.

The construction algorithm shown in Algorithm 1 consists of two main steps. First, it partitions an operator path into a few segments in the first 14 lines of the algorithm. Second, it prunes the global segment link list, which is initially empty, due to the join or multiple-way operators and the sharing of two or more query plans and then adds the new segments into the list.

For each operator path in the system, we repeat the following procedure: Consider an operator path with m operators $O_1, O_2, \cdots, O_m$ from leaf to root. Starting from O_1, a segment of the operator path is defined as a set of consecutive operators $\{O_k, O_{k+1}, \cdots, O_{k+i}\}$ where $k \geq 1$, such that $\forall j, k \leq j < k+i, C^M_{O_j} \leq C^M_{O_{j+1}}$. Once such a segment is constructed, we start the construction procedure again from O_{k+i+1} until all the operators along the operator path have been processed. In the pruning procedure, a new segment is added to the segment link list only if: *i*) any of its subset has already been in the list, then we remove all its subsets from the segment list and then add the new segment into the list; *ii*) none of its supersets has been in the list, then we add it to the list; otherwise, the new segment is discarded.

Algorithm 1: Greedy Segment Construction Algorithm

```
INPUT: the operator path $p$, the global operator segment link $\mathcal{GSL}$
OUTPUT: the updated global operator segment link
/*$p$ consists of a list of operator references along the path from left to root */
1  $tempList\mathcal{S} \leftarrow NULL$;
2  $seg \leftarrow NULL$;
3  while $p \neq NULL$ do
4      if $seg == NULL$ then append p−>operator to $seg$ else
5          if the processing capacity of p−>operator is no less than that of the last
           operator of the segment seg then append p−>operator to $seg$;
6          else
7              add $seg$ to $tempList\mathcal{S}$;
8              $seg \leftarrow NULL$;
9              append p−>operator to $seg$
10         end
11     end
12     $p \leftarrow$ p−>next
13 end
14 if $seg \neq NULL$ then add $seg$ to $tempList\mathcal{S}$;
/*the pruning procedure; */
15 foreach segment $s$ in $tempList\mathcal{S}$ do
16     needAdd ← TRUE;
17     foreach segment $s_0$ in global segment list $\mathcal{GSL}$ do
18         if $s$ is a subset of $s_0$ or $s == s_0$ then
           /*no need to add s to the global segment list; continue to process next
             segment in tempList */
19         needAdd ← FALSE;
20         break;
21         else
22             if $s$ is a superset of $s_0$ then delete $s_0$ from $\mathcal{GSL}$;
23         end
24     end
25     if (needAdd == TRUE) then add segment $s$ to $\mathcal{GSL}$
26 end
```

For a simple operator path, the above greedy segment construction is straightforward. However, for a complex operator path with branches, as shown in Figure 4.2, the algorithm can not be applied directly due to the branches of the operator path. In the bottom-up operator scheduling strategy, proposed as an internal scheduling strategy within the PC strategy, the operators in the shared segment of a complex operator path is scheduled from bottom to up. The operators along the branches are scheduled from a low layer to a high layer and for the operators in the same layer, those with larger processing capacity are scheduled first. Based on their execution order, we transform a complex operator path into a simple operator path, then apply the above algorithm to partition a complex operator path into segments.

The order in which we partition an operator path does not matter because the final segment list is the same for a given query plan, and the order of a segment in the segment list does not affect its priority. We only need to execute the above algorithm once in order to construct the segment list. Later on, when a new query plan is registered into the system, we need to execute the algorithm for the operator paths of the newly registered query plan. When a query plan is unregistered from the system, we have to delete all the segments belonging to that query plan. In a multiple query processing system, as one segment may be shared by two or more query plans, we have to add a *count* field to each operator of a segment to indicate how many query plans are using it. Once a segment is deleted from the system, we decrease the value in the *count* field by one for each operator that belongs to the segment. When the *count* value of an operator reaches zero, it is deleted from the segment.

Since CQ plans in a stream processing system are long-running queries, the number of queries that will be registered with a system or unregistered from a system is not likely to be too large (typically no more than a few per hour). Therefore, the cost of the algorithm has very little impact on system performance.

The segment strategy shares the same operator segment concept used by the Chain strategy [21] and the Train strategy [33] in Aurora. However, it is unclear how a segment (superbox) is constructed in [33]. On the other hand, the segment strategy is different from the Chain strategy in that: *i*) the segments used in these two strategies are different. The segments used in the Chain strategy have steepest slope in its lower envelope, while the segments used in the segment strategy consist of a consecutive operators that have an increasing memory release capacity. Therefore, the Chain strategy can achieve the near optimal internal queue size requirement, while the segment strategy has a slightly larger internal queue size requirement, but it achieves better tuple latency, *ii*) it clusters a set of operators as a scheduling unit, and hence there are no partially processed tuples buffered in the middle of an operator segment at the end of each time unit as in the Chain strategy, and *iii*) it has a smaller scheduling overhead than the Chain strategy. The Chain strategy is an operator-based strategy where all the operators along a segment have the same priority, while the segment strategy is a segment- based strategy. In a general query processing system, as the number of segments is less than the number of operators, the scheduling overhead is lower for the segment strategy.

4.3.3.2 The MOS Segment Strategy

In order to achieve the optimal memory requirement, we propose the MOS strategy, which achieves the optimal memory requirement by employing the memory-optimal segment construction algorithms illustrated in Algorithm 2 for a simple operator path and Algorithm 3 for a complex operator path.

For a simple operator path, the Algorithm 2 works perfectly. It partitions a simple OP into segments through finding the segment with the largest memory release capacity among all possible segments that begin with the leaf operator of an OP or the remaining of an OP.

Algorithm 2: Memory-Optimal Segment Construction Algorithm For A Simple Operator Path

```
   INPUT: the simple operator path $p$, the global operator segment link $\mathcal{GSL}$
   OUTPUT: the updated global operator segment link
1  $tempList\mathcal{S} \leftarrow NULL$;
2  $seg \leftarrow NULL$;
3  $startOpOfSeg \leftarrow$ p−>operator;
4  $endOpOfSeg \leftarrow NULL$;
5  while $startOpOfSeg \neq NULL$ do
6      $potentialEndOp \leftarrow startOpOfSeg$;
7      maxCapacity $\leftarrow$ 0;
8      while $potentialEndOp \neq NULL$ do
9          form the segment $seg$ by all operators from $startOpOfSeg$ to
           $potentialEndOp$;
10         $tempCapacity \leftarrow$ compute the processing capacity of the segment $seg$;
11         if $maxCapacity \leq tempCapacity$ then
12             maxCapacity $\leftarrow$ tempCapacity ;
13             $endOpOfSeg \leftarrow potentialEndOp$;
14         end
15         $potentialEndOp \leftarrow potentialEndOp$−>next;
16     end
17     form the segment $seg$ by all operators from $startOpOfSeg$ to
       $endOpOfSeg$;
18     add $seg$ to $tempList\mathcal{S}$;
19     $startOpOfSeg \leftarrow endOpOfSeg$−>next;
20 end
   /*the pruning procedure is the same as in the Algorithm 1;                 */
```

For a complex operator path, the approach used in the greedy segment construction algorithm can not achieve the optimal memory requirement. Instead, we introduced the Algorithm 3 to partition a complex operator path into segments. There are three main steps in this algorithm: (*1*) transform a complex operator path into a list of possible simple operator paths; (*2*) apply Algorithm 2 to each simple operator path constructed in the first step and then select the segment with the largest processing capacity; (*3*) for the remaining part of the complex operator path (excluding those in the segment constructed in second step), recursively apply the step 1 and 2 until there is no operator left.

The first step is to transform a complex operator path to a list of possible simple paths (Line 3 in Algorithm 3). The main idea behind this step is to enumerate all possible execution orders, but preserving their orders along the path, of the operators along a complex operator path. The operator A that takes the direct or indirect outputs of another operator B can not be scheduled before B is scheduled. For example, the operators along the shared segment must scheduled according to their order along the segment, and the operators along a branch must scheduled according to their order along the branch too, but not necessarily in a consecutively order, the operators from other branches can be scheduled between two operators. For example, for the complex operator illustrated in Figure 4.5-a, we list some possible execution orders in Figure 4.5- b. The second step is to find the segment with the largest processing capacity among all segments from all possible simple operator path constructed in first step (from Line 4 to Line 26 in Algorithm 3). Notes: Only the first segment (Line 7 to Line 21 in Algorithm 3) is constructed from a simple path because the second one has less processing capacity than the first one; If it is not, we can reconstruct the first segment by combining the first and the second segment into a new segment which must have a bigger processing capacity than the first one. The third step is to repeat the step 2 for the remaining part of the simple operator path that is used in step 2.

The difference between the MOS segment construction algorithms and the Greedy segment construction algorithm lies in the way segments are constructed. The segments constructed by the Algorithm 1 consist of consecutive operators as long as the processing capacity of an operator in the segment is no larger than that of its next operator. On the other hand, the segments constructed by the Algorithm 2 consist of a list of consecutive operators as long as the processing capacity of the segment formed by the list of operators is no less than that of any segment formed by the first operator in the list to each upstream operator along the *operator path*, instead of *operator segment*. An example is shown in

Algorithm 3: MOS Construction Algorithm For A Complex Operator Path

```
INPUT: the complex operator path $p$, the global operator segment link $\mathcal{GSL}$
OUTPUT : the updated global operator segment link $\mathcal{GSL}$
1  $TempList \leftarrow NULL$;
2  while $p \neq NULL$ do
       /*STEP ONE: emulatePossibleSimpleOperatorPaths($p$) list all possible
         execution orders of the operators along the complex operator path $p$ and those
         execution orders do not violate their order along the path.                  */
3      $EmulationList\mathcal{E} \leftarrow$ EmulatePossibleSimpleOperatorPaths($p$);
       /*STEP TWO: find the segment with the largest processing capacity among all
         segments from any simple path in $EmulationList\mathcal{E}$.                  */
4      $OptimalSeg\mathcal{S} \leftarrow NULL$;
5      $SimplePath \leftarrow NULL$;
6      foreach $\mathcal{SP}$ in $EmulationList\mathcal{E}$ do
           /*find the segment with the largest processing capacity among all segments
             starting from the first operator of the simple path $\mathcal{SP}$.      */
7          $seg \leftarrow NULL$;
8          $startOpOfSeg \leftarrow \mathcal{SP}{-}{>}$operator;
9          $endOpOfSeg \leftarrow NULL$;
10         $potentialEndOp \leftarrow startOpOfSeg$;
11         maxCapacity $\leftarrow 0$;
12         while $potentialEndOp \neq NULL$ do
13             form the segment $seg$ by all operators from $startOpOfSeg$ to
               $potentialEndOp$;
14             $tempCapacity \leftarrow$ compute the processing capacity of the segment $seg$;
15             if $maxCapacity \leq tempCapacity$ then
16                 maxCapacity $\leftarrow$ tempCapacity ;
17                 $endOpOfSeg \leftarrow potentialEndOp$;
18             end
19             $potentialEndOp \leftarrow potentialEndOp{-}{>}$next;
20         end
21         form the segment $seg$ by all operators from $startOpOfSeg$ to $endOpOfSeg$;
           /*Select the segment with the largest processing capacity and the simple
             path to which it belongs;                                                 */
22         if (OptimalSeg == NULL) OR (OptimalSeg->processingCapacity <
           seg->processingCapacity) then
23             optimalSeg = $seg$;
24             SimplePath = $\mathcal{SP}$;
25         end
26     end
       /*STEP THREE: remove all operators on the $seg$ from the $SimplePath$ and
         repeat STEP ONE and STEP TWO;                                                 */
27     $TempList \leftarrow \{TempList + optimalSeg\}$;
28     $p \leftarrow \{SimplePath$ - $seg\}$;
29 end
   /*STEP FOUR: add all segments in $TempList$ to global list $\mathcal{GSL}$ by starting the
     pruning procedure in the Algorithm 1;                                             */
```

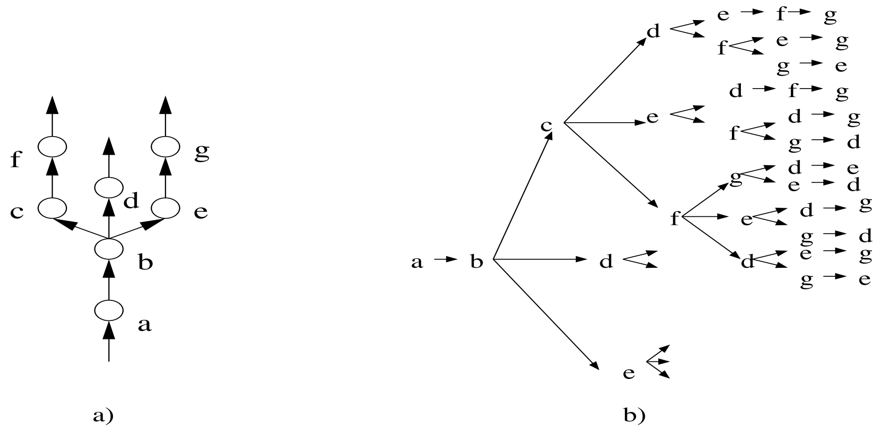

Figure 4.5 a) A Complex OP; b) Some Execution Orders of The Complex OP.

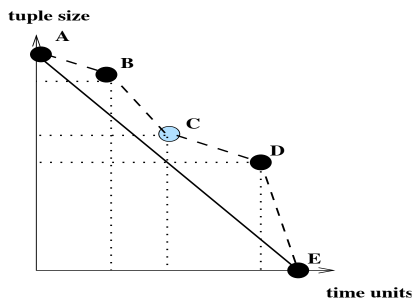

Figure 4.6 Example of Segmentation.

Figure 4.6 in which the Algorithm 1 partitions the path into 2 segments, which are ABC and DE, while the Algorithm 2 partitions the path into 1 segment, which is $ABCDE$.

In our segment strategy, the memory requirement (internal queue size) of the query processing is minimized by using the segments constructed by Algorithm 2 and 3. The segment strategy that employs the MOS construction algorithms are termed as MOS strategy. The MOS strategy has a number of advantages over the Chain strategy although both use the similar algorithm to construct the segments in the system (Notes: the progress chart construction in the Chain strategy is only for simple operator paths. We extend it to the complex operator paths in this thesis). First, it works under both single query (no sharing computation) and multiple query (with sharing) processing systems. Second, it achieves the strictly optimal memory requirement theoretically, instead of the near optimal memory requirement as in the Chain strategy. Third, it achieves better tuple latency and smoother throughput than the Chain strategy. Finally, it has lower overhead than the Chain strategy.

THEOREM 3. *The MOS strategy minimizes the memory requirement of a CQ processing system with multiple queries. It achieves the optimal memory requirement.*

Proof. We assume that there exists an optimal algorithm ALG_{opt}, which always schedules the object that could be an operator, an operator segment, an operator path or a bigger schedulable object (i.e., a query plan) to minimize the memory requirement in the system at any time point. Considering each of these three cases: an operator, an operator segment, an operator path or bigger object, we prove the memory required by the MOS strategy is no more than that required by the ALG_{opt}.

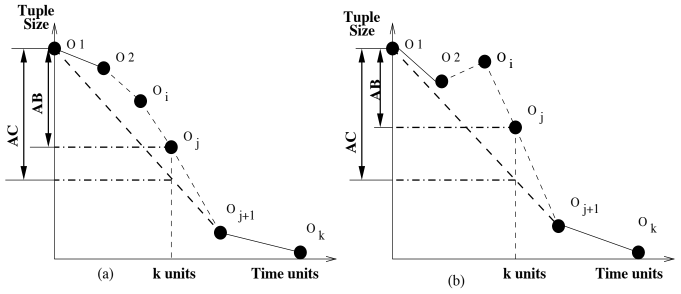

Figure 4.7 (a)$C^M_{O_i} \leq C^M_S$; (b)$C^M_{O_i} > C^M_S$.

Case one (operator): at the time slot t_i, the ALG_{opt} schedules an operator for one time unit and achieves the minimal memory requirement so far. Without loss its generality, we can assume that the operator is the O_i along a k-operator segment $\mathcal{S}_k$, which consists of $< O_1, O_2, \cdots, O_i, O_{i+1}, \cdots, O_k >$. There are two subcases: one is shown in Figure 4.7-a where the memory release capacity of the operator O_i is no larger than that of the segment $\mathcal{S}_k$; another is shown in Figure 4.7-b where the memory release capacity of the operator O_i larger than that of the segment $\mathcal{S}_k$. For either case, the inputs of the operator O_i are the outputs of the operator O_{i-1}. Therefore, the ALG_{opt} must have scheduled the operators from O_1 to O_{i-1} along that segment at least once so far in order to be able to schedule the operator O_i. At each time, when the ALG_{opt} schedules one of these operator, we have an algorithm ALG_{seg} to schedule the segment $\mathcal{S}_j$ once. The segment $\mathcal{S}_j$ consists of the operators $< O_1, O_2, \cdots, O_j >$ along the segment $\mathcal{S}_k$ and its the memory release capacity is no less than that of the segment consisting of $< O_1, O_2, \cdots, O_i >$. In the worst case, we have a segment consisting of $< O_1, O_2, \cdots, O_i >$. The total amount of memory released by ALG_{opt} is AB shown in Figure 4.7. The amount of memory released by ALG_{seg} is AC shown in Figure 4.7. It clearly shows that the amount of memory released by ALG_{seg} is no less than that released by ALG_{opt}. According to the definition of MOS strategy, we know that MOS always schedules the segment with the largest memory release capacity. Therefore, the amount of memory released at each time unit by MOS strategy is no less than that released by ALG_{seg}. In general, the amount of memory released by the MOS strategy is no less than that released by ALG_{opt} if it schedules an operator at any time slot in the system. In other words, the MOS strategy requires no larger amount of memory than the ALG_{opt}.

Case two (segment): at any time slot, if the ALG_{opt} schedules the operator segment S_{opt} for one time unit and achieves the minimal memory requirement in the system, the amount of memory required by S_{opt} within one time unit is its memory release capacity $C^M_{S_{opt}}$. Since the segment scheduled by MOS strategy has the largest memory release capacity in the system and has a memory release capacity no less than $C^M_{S_{opt}}$. We can conclude that the MOS strategy requires no more than the amount of memory needed by the ALG_{opt} for this case.

Case three (operator path): at the time slot t_i, the ALG_{opt} schedules an operator path or a bigger object (i.e., query pan) for one time unit and achieves the minimal memory requirement in the system. Since the memory release capacity of an object (i.e., a query plan) that is larger than an operator path is no larger than the largest memory release capacity of all operator paths encompassed in the object, if the ALG_{opt} can achieve the minimal memory requirement by scheduling an object bigger than operator path for one time unit, we can always construct an algorithm ALG_{path} to schedule the operator path that has the largest memory release capacity among all operator paths encompassed in the object for one time unit. The amount of memory required by ALG_{path} is no less than that released by ALG_{opt}. Similarly, the memory release capacity of an operator path is no larger than the largest memory release capacity of all segments encompasses that operator path. If ALG_{path} can achieve the minimal memory requirement by scheduling an operator path one time unit in the system, we could construct an algorithm $ALG_{s\bar{e}g}$ to schedule the segment with the largest memory release capacity among all segments encompasses in that operator path. The memory required by $ALGs\bar{e}g$ is no more than that required by ALG_{path}. Since the MOS strategy schedules the segment with the largest memory release capacity the system, the memory release capacity of the segment scheduled by the MOS strategy is no less than that of the segment scheduled by $ALG_{s\bar{e}g}$. Therefore, the memory required by the MOS strategy is no more than that required by ALG_{opt} for this case.

In conclusion, the MOS strategy requires no more memory required by the ALG_{opt} for any of the cases. It is an optimal strategy in terms of memory requirement. □

The advantages of the MOS strategy as compared to the Chain strategy are due to the following facts:

- The segment construction algorithms used in MOS strategy work for query processing systems with simple operator paths, but they also work in multiple query processing systems with computation sharing (i.e., sharing sub-common expressions) where one input tuple can output multiple tuples from different paths. However, the progress chart in the Chain strategy only works for query plans without any computation sharing (i.e, only for simple paths). Our segment construction algorithm for a complex operator path can also be applied to extend the progress chart in the Chain strategy to work in a multiple query processing system. The proof of minimization of memory requirement in [21] is only for the query plans without join or multiple-way operators and without computation sharing through sharing sub-common expression. In this chapter, we proved that the MOS strategy minimizes the memory requirement for a multiple CQ processing system with general query plans.

- There is one tuple buffered in the middle of the segment for the Chain strategy and no tuple buffered in the middle of a segment for the MOS strategy. That is the reason for the Chain to be near optimal strategy in terms of memory requirement and the MOS strategy to be an optimal one. Although both of them partition an operator path into a set of segments, the Chain assigns the same priority to all the operators in that segment and then it schedules operators based on priority. For the operators with the same priority, it schedules the operator with the oldest tuples, which also cause higher overhead due to keep tracking of the oldest tuple. The MOS strategy, on the other hand, schedules the whole segment as one object and for segments with the same priority, the MOS strategy schedules one arbitrarily. At any time instant, the Chain schedules the operator with the highest priority for one time unit (or one tuple) while the MOS strategy schedules the operator segment on which the operator scheduled by the Chain lies, one time unit (or one tuple).
- The MOS strategy preserves the order of the operators with the same priority by placing them along a segment. Therefore, no other operators can be scheduled in an interleaved manner with those operators along one segment. The Chain strategy

may schedule the operators of these segments in an interleaved manner if two or more segments in the system have the same priority (memory release capacity) and the tuples with these segments have the same age. This, in turn, would cause longer overall tuple latency. For example, given two segments $ABCD$ and $EFGH$, the Chain strategy may schedule those operators in the order $AEBFCGDH$ (one of the interleavings), while MOS strategy schedules operators in order of segment $ABCD$ and segment $EFGH$ or vice versa. In either case, the MOS strategy achieves a better tuple latency than the Chain strategy and the same or better memory requirement.

- As we discussed earlier, the PC strategy has a smoother throughput than the Chain strategy. For the same reasons, the MOS strategy has a smoother throughput than the Chain strategy, but is a little bit more bursty in throughput than the PC strategy.
- The MOS strategy has a lower overhead than the Chain strategy. This is because i*)* the number of segments in the system is less than the number of operators; ii*)* for the operators with the same priority, the Chain strategy needs to keep track of the ages of the tuples. Due to the highly dynamic input characteristics of data streams and the frequent calls to the scheduling function, the overhead can be high even though the number of operators with the same priority is small. The MOS strategy schedules the segments with the same priority arbitrarily, it does not have this overhead.

4.3.3.3 Simplified Segment Strategy

The (greedy) segment strategy and the MOS strategy decrease the memory requirement as compared to the PC strategy, but they still cause longer tuple latency than the PC strategy because they separate one operator path into multiple segments. Chapter

3 shows that the overall tuple latency of the tuples from an operator path increases significantly if other operators are scheduled in an interleaved manner with the operators along the operator path. In order to decrease the interleaving of the operators of two segments, we propose the SS strategy by partitioning an OP into at most two segments.

The SS strategy differs from the segment strategy and the MOS strategy in that it employs a simplified MOS segment construction algorithm. In a practical multiple CQ processing system, we observe that: *i*) the number of segments constructed by the segment construction algorithm is not significantly less than the number of operators presented in the query processing system and *ii*) the leaf nodes are the operators that have faster processing capacities and less selectivity in the system; all the other operators in a query plan have a much slower processing rate than the leaf nodes. Based on these facts, we partition an operator path into at most two segments, rather than a few segments. The first segment includes the leaf node and its consecutive operators that come from the operator path by using the MOS construction algorithm (Alg. 2). The remaining operators along that operator path, if any, forms the second segment.

Although the memory requirement of the SS strategy is only slightly larger than the segment strategy because the first segment of an operator path releases the maximum amount of memory that can be released by the operator path, it has the following advantages: *i*) the tuple latency significantly decreases because the number of times a tuple is buffered along an operator path is at most two, *ii*) the scheduling overhead significantly decreases as well due to the decrease in the number of segments, and finally *iii*) it is less sensitive to the selectivity and service time of an operator because there exist at most two segments for an operator path, which makes it more useful.

4.3.4 Threshold Strategy

The threshold strategy is a dynamic strategy and is a hybrid of the PC strategy and the MOS strategy. The principle behind it is that the PC strategy is used to minimize the tuple latency when the memory is not a bottleneck; otherwise, the MOS strategy is used to decrease the total memory requirement. Therefore, this one combines the properties of these two strategies, which makes it more appropriate for a DSMS.

Threshold Strategy: *Given a CQ processing system with a maximal available queue memory*[7] $\mathcal{M}$*, the maximal threshold* $\mathcal{T}_{max}$ *and the minimal threshold* $\mathcal{T}_{min}$*, where* $\mathcal{T}_{min} < \mathcal{T}_{max} < \mathcal{M}$*, at any time instant, when the current total queue memory consumed* $\mathcal{M}_c \geq \mathcal{T}_{max}$*, the system enters its memory saving mode in which the MOS strategy is employed. The system transits from the saving mode to the normal mode in which the PC strategy is employed when* $\mathcal{M}_c \leq \mathcal{T}_{min}$*.*

The values of the maximal threshold $\mathcal{T}_{max}$ and the minimal threshold $\mathcal{T}_{min}$ mainly depend on the load of the system and the length of the bursty periods; and they can be obtained heuristically or experimentally. Given that the mean total queue memory consumed by a CQ processing system is $\bar{\mathcal{M}}$ memory units, we define the values of these threshold parameters in our system as

$$\begin{cases} \mathcal{T}_{max} & = min\left(\frac{1+\alpha}{2}\mathcal{M}, \beta\mathcal{M}\right);\ \alpha = \frac{\bar{\mathcal{M}}}{\mathcal{M}} \\ \mathcal{T}_{min} & = min\left(\bar{\mathcal{M}}, \beta\mathcal{T}_{max}\right);\quad 0.5 < \beta < 1 \end{cases} \tag{4.8}$$

In (4.8), β is a safety factor that guarantees a minimal memory buffer zone between the normal mode and the saving mode, which prevents a system from frequently oscillating between the memory saving mode and the normal mode. A smaller value of β causes a longer tuple latency. Therefore, its value need to be in the range of 0.5 to 1.0. α is used

[7]The queue memory here refers to the memory available for input queues, not including the memory consumed for maintaining the status information of an operator.

to adjust the threshold values as the system load changes. The mean total queue size increases as the system load increases, which causes α increases. When α approaches 1, the $\frac{1+\alpha}{2}\mathcal{M}$ factor approaches the maximal available queue memory $\mathcal{M}$. That is why we need $\beta\mathcal{M}$ to guarantee that there is a minimal buffer between the $\mathcal{T}_{max}$ and $\mathcal{M}$. We use $\beta = 0.9$ in our system. Our experiments show these parameters work well in general.

In a practical system, we have to monitor the current queue memory in order to determine when to switch the mode. But the cost to this is small because: *1*) each queue in our system maintains its current queue size and the tuple size, and the current queue memory is the sum of the queue memory occupied by each queue; *2*) instead of computing the current queue memory by the end of each time, we compute it by the end of each time interval. The length of time interval is dynamically determined based on current queue memory. If the current total queue memory size is far away from the total available memory size, a long time interval is used. Otherwise, a shorter interval is used. Therefore, the overall overhead incurred by the threshold strategy has very little impact on the system performance. As the mean load of the system increases, the period for which the system stays under the saving mode increases and the overall tuple latency becomes worse. When there is no more memory available for the internal queues under the saving mode, the load shedding techniques have to be used to relieve the system from suffering from a shortage of memory.

4.4 Discussion

In this section, we briefly discuss how different execution plans of a CQ impact the performance of a system under the proposed scheduling strategies, and then discuss how to avoid the starvation problem while using these scheduling strategies.

4.4.1 Continuous Query Execution Plan

A logical CQ can be implemented as different physical query execution plans. Based on our analysis of the scheduling strategies, we find that the following points are helpful for choosing the right physical execution plan in order to improve the system performance.

Push the select and the project operators down: Both the PC strategy and the segment strategy can benefit from the lower selectivity of a leaf operator and from the earlier project operator. From (4.2), we know that: *i*) the lower selectivity of a leaf operator dramatically increases the processing capacity and the memory release capacity of an operator path or segment; *ii*) the down-side project operators can decrease the output size of the tuples earlier and the released memory can be quickly reused. Therefore, both tuple latency and memory requirements of an operator path or segment can be optimized by pushing the selection and the project as far down as possible in a physical query plan.

Make the operator path short: The processing capacity of an operator path or segment depends not only on the selectivity of the individual operator, but also on the number of the operators. It may not increase the processing capacity of an operator path or segment to make the operator path short because the service time of an individual operator may increase. For instance, by incorporating a project operator into a select operator, we can shorten the operator path, but this does not increase the processing rate of the path. However, the number of times an output tuple is buffered decreases, which can decrease the tuple latency and the scheduling overhead of a scheduling strategy as well. In addition, fewer number of operators in a path makes it much easier to control or estimate the tuple latency.

4.4.2 Starvation Free Scheduling Strategy

All the scheduling strategies discussed so far are priority driven strategies. Under an overloaded system some paths/segments may have to wait for a long period to be scheduled or even not scheduled at all theoretically. To overcome the starvation problem, we discuses two simple solutions here, and we are still investigating other solutions for this problem.

The solutions we present here are applicable to all strategies discussed in this chapter. Furthermore, an operator path and an operator segment are used interchangeably in this subsection.

Periodically schedule the path with the oldest tuple in its input queue: A straight-forward solution to the starvation problem is to periodically schedule the path with the oldest tuple in its input queues in our proposed strategies. The length of the period to schedule the oldest operator path depends on the load of a system and the QoS requirement of its applications.

Dynamic Priority: Another solution is to change the priority of the strategies periodically. The total waiting queue size and the age of the oldest tuple of an operator path characterize its activities, such as the mean input rate, schedule frequency, and so on. In order to avoid the starvation problem, we consider the total waiting queue size and age of the oldest tuple as two additional factors of the priority of an operator path. And we define the priority factor f_i of the operator path i as:

$$f_i = \tau_i Q_i$$

where τ_i is the normalized waiting time of the oldest tuple in the input queue of the path i; Q_i is the normalized total current queue size of that operator path. Therefore, the new capacity of an operator path $\hat{C}_i^P = C_i^P f_i$. Evidently, as the age increases, the queue size increases as well, which makes the priority factor f_i increase exponentially. During a highly bursty input period, its priority factor increases too. Eventually, the oldest path will be scheduled.

Although the above solutions can solve the starvation or long waiting problem due to the temporary overload (the overall input rate of the system is greater than its overall processing rate), they cannot reduce the system load. Once the load of a system is beyond its maximal capacity it can handle, load shedding [114, 74] or sampling techniques have to be used to relieve the load, which is beyond the capability of a scheduling strategy. However, a scheduling strategy can be aware when these techniques have to be used. We are currently investigating this problem.

4.5 Experimental Validation

We have implemented the proposed scheduling strategies as part of the prototype of a QoS aware DSMS – MavStream [76, 75]. In this section, we discuss the results of various experiments that we have conducted in order to compare the performance of these scheduling strategies.

4.5.1 Setup

We begin with a brief description of our experimental setup. The scheduling strategies we have implemented in MavStream include: the PC strategy, the Chain strategy, the MOS strategy, the SS strategy, the Threshold strategy, and various round-robin strategies. We use the following data streams and CQs in our experiments. The scheduling model used for the experiments is the model discussed in §4.1.1. An object is moved from the waiting queue to the eligible queue if the number of waiting tuples exceeds a threshold value which is the number of operators in that object times 100.

We only need the strategies that do a better job during high-load situations in the system. A strategy that can do a much better job during light-load periods is not useful if it can not do a better job during high- load periods. Therefore, the performance reported in this chapter is the performance of each strategy from the same periods and during those periods, the system is overloaded at least for a periods of time.

Input data streams: The input data streams we used are highly bursty streams and the input rate of each stream is controlled by a global bursty factor and a local bursty factor. We think only the strategies that do a better job during heavy load periods are useful since nobody cares the performance of a strategy during light load periods. Therefore, only the performance data during the heavy load periods, which is defined, in this chapter, the system is overloaded during the high-input phase of our 3-phase periods discussed in the following, are reported and compared.

The global burst factor is a 3-phase period. Each stream is generated by repeating the 3-phase period with an increasing standard mean rate. Each phase lasts about 20 minutes. In the first phase, the mean input rate is 2 times of the standard mean rate. In the second phase, the standard mean rate is used. In the last phase, a much lower input rate is used, which is one quarter of the standard mean rate. In order to test how strategies react to the overload situations, we continuously increase, by 5 percent or so, the standard mean input rate each time when we repeat the 3-phase pattern until the system is overloaded (When the maximal memory requirement is larger than 2M bytes, we regard the system is overloaded).

The local burst factor is a so called self-similarity factor. Given the mean input rate of each phase of each period, at each time unit (second in this thesis), the input rate is controlled by the so called self-similarity property of each stream. This self-similarity property is due to that each input stream is a superposition of 64 or 128 flows. Each flow alternates ON/OFF periods, and it only sends tuples during its ON periods. The tuple inter-arrival time follows an exponential distribution during its ON periods. The lengths of both the ON and the OFF periods are generated from a Pareto distribution which has a probability mass function $P(x) = ab^a x^{-(a+1)}, x \geq b$. We use $a = 1.4$ for the ON period and $a = 1.2$ for the OFF period. For more detailed information about self-similar traffic, please refer to [85]. In our experiment, we use 5 such self-similar input data streams with different mean input rates.

Experimental query plans: All of our queries are CQs that consist of *select, project*, and *symmetric hash join* operators. To be more close to a real application, we ran 16 actual CQs with 116 operators over 5 different data streams in our system. The selectivity of each operator is widely distributed ranging from 0 to 1. Both the selectivity and the processing capacity of each operator can be determined by collecting statistical information periodically during run time. The details of the list of queries and their properties are presented in Appendix A.

The prototype is implemented in C++, and all the experiments were run on a dedicated dual processor Alpha machine with 2GB of RAM. One of the processors was used to collect experiment results while another processor was used for query processing.

4.5.2 Performance Evaluation

Due to the fact that the CQs are also long running queries and that the scheduling strategies demonstrate different performance during different system load periods, we ran each experiment for more than 24 hours (including the statistics collection period), and each experiment consists of multiple phases. In each phase, we intentionally increased the average input rates of data streams in order to study and validate the performance characteristics of a scheduling strategy under different system loads. We only present a portion of our experimental data (from a few phases), rather than a full range of results due to limited space. For threshold strategy, we set the maximal threshold $\mathcal{T}_{max}$ to 10M bytes, which means it employs the PC strategy when its total queue size is less

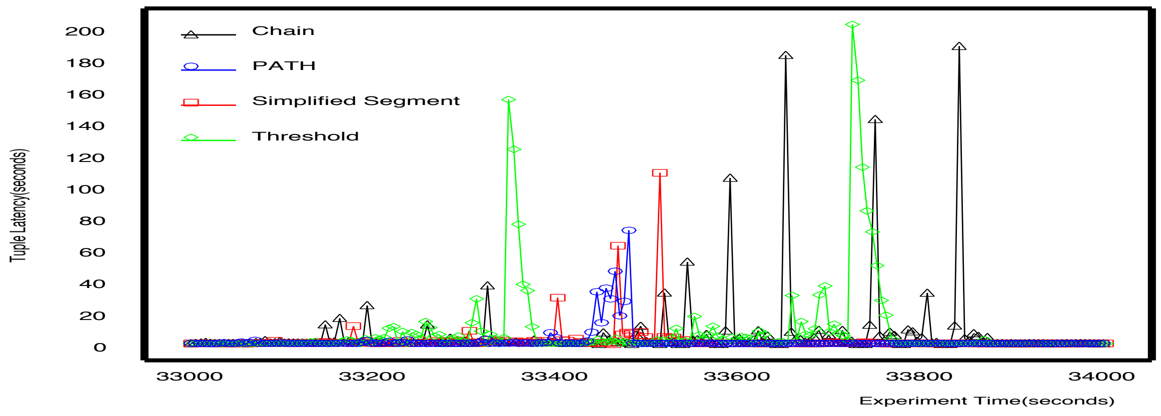

Figure 4.8 Tuple Latency vs. Time.

than 10Mbytes. Otherwise, it employs the MOS strategy to decrease total queue size requirement.

Tuple latency: The tuple latency of an output tuple is computed by taking the difference of its arrival time-stamp and its departure time- stamp when it leaves the query processing system. We presented two sets of our experiments for the proposed scheduling strategies in Figure 4.8 and 4.9 respectively. The tuple latencies shown in both figures are the average tuple latency of all output tuples within every 1 second.

From the results in Figure 4.8, we observe that the overall tuple latency is much better under the PC strategy than under the SS strategy and the Chain strategy. The Chain strategy performs worst among them. Furthermore, the overall tuple latency increases as the system load increases, but the difference among them becomes much sharper as the system load increases. The threshold strategy has a tuple latency as good as the PC strategy when total queue size is less than 10M bytes (i.e., from 33200 to 33300). However it performs as bad as the MOS strategy during heavy bursty periods (i.e., from 33600 to 33800). It is worth noting that during light load periods (i.e., the first 200 seconds), all of them have a reasonable tuple latency except that the Chain has a few spikes. When the system load increases, the tuple latency increases sharply during

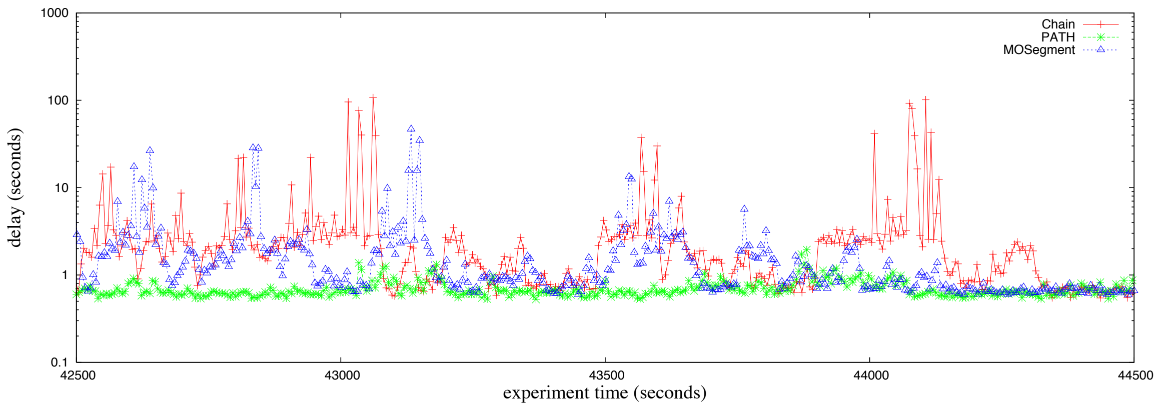

Figure 4.9 Tuple Latency vs. Time.

the highly bursty input periods for both the SS strategy and the Chain strategy. As we explained earlier, the high tuple latency under the Chain strategy and the SS strategy contributes to their buffered tuples in the middle of an operator path. The SS strategy performs better than the Chain strategy because it buffers less number of times of a tuple along an operator path than the Chain strategy.

The tuple latency shown in Figure 4.9 further confirms the conclusions that we derived from the first set of experiments: the PC strategy achieves a much better tuple latency than the Chain strategy. The figure also shows that the MOS strategy achieves a better tuple latency than the Chain strategy, but a longer tuple latency than the PC strategy. This is due to the fact that MOS is a kind of segment strategy.

Throughput: The total throughput of a CQ processing system under any scheduling strategy should be the same because it should output the same number of output tuples no matter what scheduling strategy it employs. However, the output patterns are likely to be dramatically different under different scheduling strategies. Figure 4.10 and 4.11 show the output patterns under different strategies. In order to clearly show the difference, we use the logarithm scale in y axis in Figure 4.11.

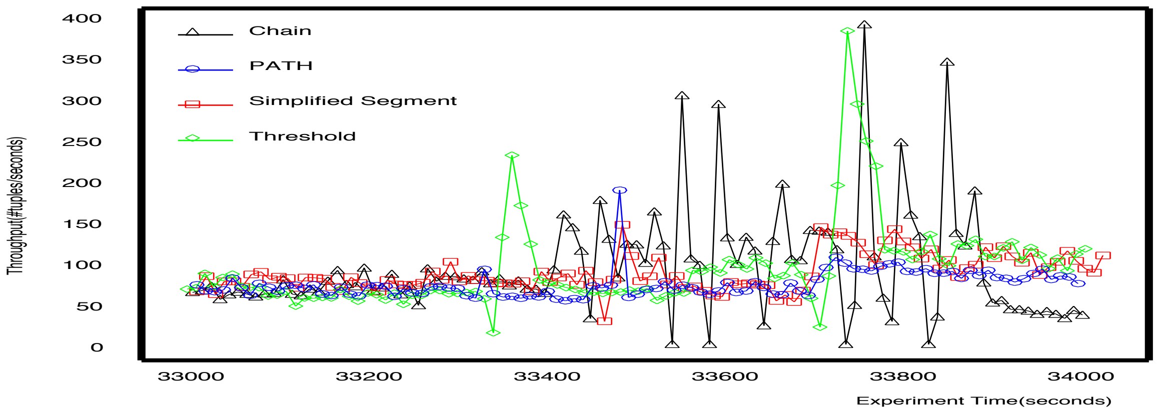

Figure 4.10 Throughput vs. Time.

In Figure 4.10, the PC strategy and the SS strategy have a much smoother output rate than the Chain strategy, and the threshold strategy has the metrics of both the PC strategy and the MOS strategy. The PC strategy performs best among them in terms of the bursty output. The output rate under all four strategies increases as the input rates increase when the system load is moderate, which is the first 300 seconds, and their output patterns do not differ with each other too much. After the system enters the high load periods, the PC strategy and the SS strategy have a much smoother throughput than the other two during the high bursty input periods which are the periods of the 10400 second to 11000 second and from the 12400 second to 12800 second. In contrast, the Chain strategy has a very low throughput, even no throughput during heavy load periods. On the other hand, it has a surprisingly high throughput immediately when system load decreases. Its highest output rate is almost 4 times its average output rate. The situation becomes worse when the system load or the length of the highly bursty input periods increases. This highly bursty output rate is not desirable because of the amount of partial results that have to be buffered in the system temporarily, which consumes unnecessary memory.

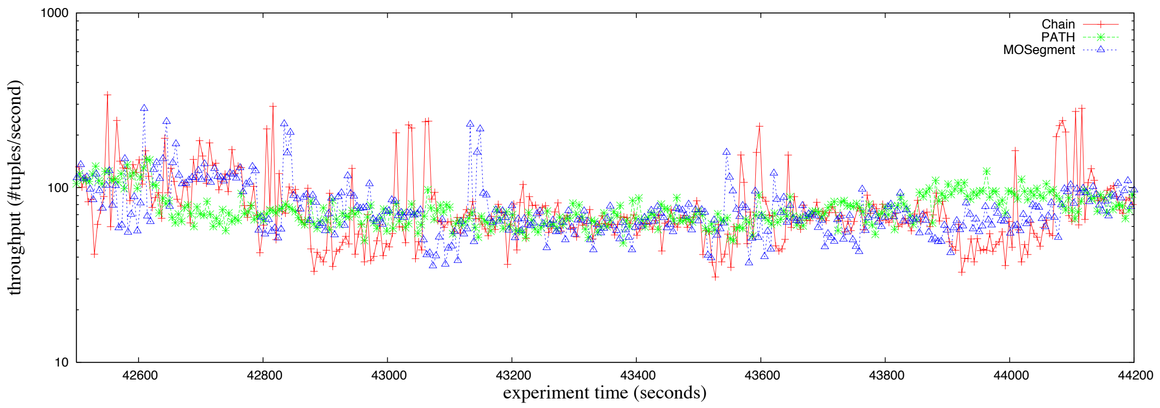

Figure 4.11 Throughput vs. Time.

The throughput patterns shown in Figure 4.11 further confirm that the PC strategy has a much smoother throughput pattern than the Chain strategy. The figure also shows that the MOS strategy has a smoother throughput than the Chain strategy during the high load periods. During the low load periods, all strategies have similar throughput patterns.

Memory requirement: We study the total memory requirement of a CQ processing system under different scheduling strategies given an input pattern. The amount of memory consumed by the query processing system is measured by calculating the memory consumed by all input queues every one second. The memory consumed by each input queue is the total number of tuples waiting in the queue times the size of the tuple. The amount of memory consumed by the scheduling strategies proposed in this chapter is presented in Figure 4.12 and 4.13. The y axis in Figure 4.13 is the logarithm of the amount of memory consumed by the query processing system.

From the results presented in Figure 4.12, we observe that the Chain strategy performs better than the others. The Chain strategy and the SS strategy can absorb the extra memory requirement during the bursty input periods when system load is not high. Although the PC strategy has the capability to absorb the temporary high bursty input,

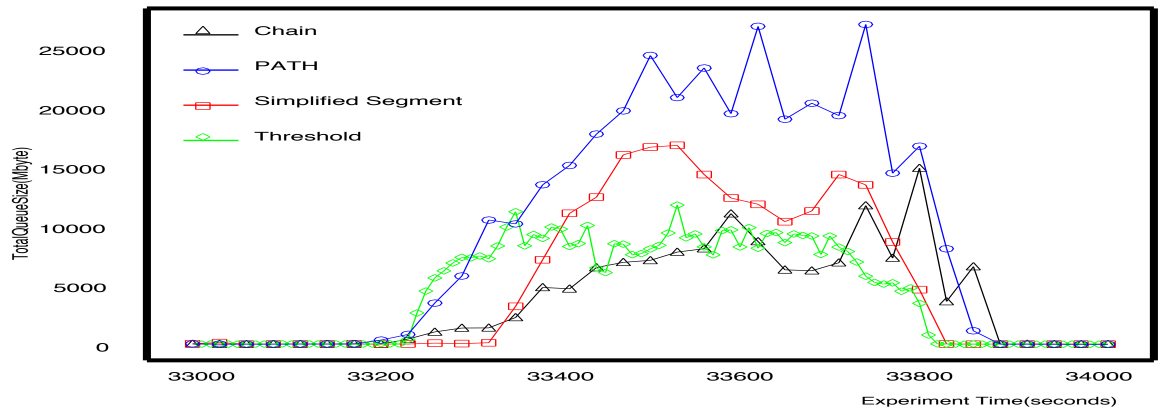

Figure 4.12 Total Memory Requirement vs. Time.

its ability is much less than the other two strategies. The reason, as we mentioned earlier, is that the leaf nodes have a much larger memory release capacity than the other nodes of a query plan and that the memory release capacity of an operator path is much less than that of a leaf node. As the system load or the length of a bursty period increases, the PC strategy requires much more memory to temporarily buffer the unprocessed tuples than the other two strategies. The SS strategy requires a little bit more memory than the Chain strategy during the highly bursty periods. This is because it only takes benefit of the larger memory release capacities of the leaf nodes that are major part of the operators with a larger memory release capacity in a CQ processing system, but not all of them behave like the Chain strategy. The threshold strategy has a similar memory requirement as the PC strategy during the first 300 seconds. It then switches to the Chain strategy and maintains its total memory requirement around 10M bytes, which is similar to the Chain strategy.

From the results shown in Figure 4.13, we can see that the MOS strategy has a smaller memory requirement than the Chain strategy and the PC strategy has the largest memory requirement. This is due to that i) the MOS strategy does not buffer tuples in the middle of a segment, which is unavoidable for the Chain strategy; ii) the overhead

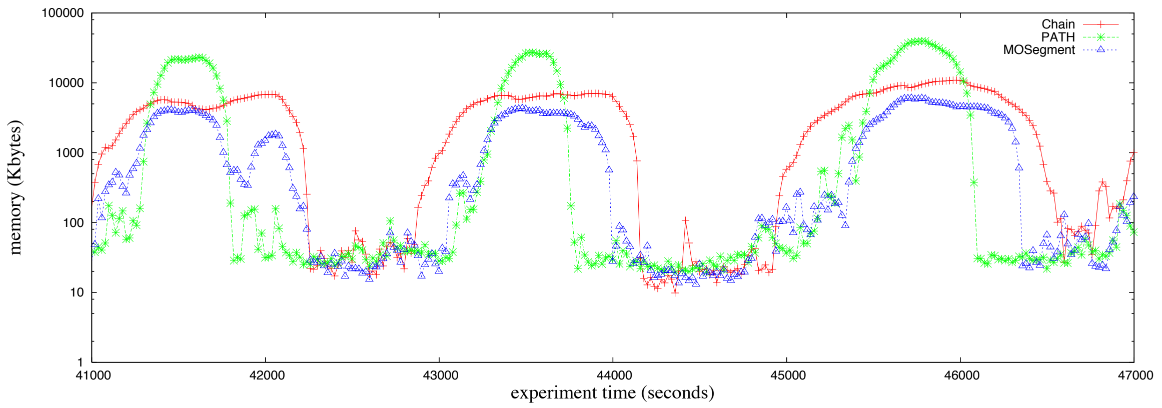

Figure 4.13 Total Memory Requirement vs. Time.

introduced by MOS strategy is smaller than the Chain strategy. The PC strategy requires much more memory than the MOS and Chain strategy because it does not take advantage of the higher memory release capacity of the operators at the bottom side of an operator path.

4.6 Summary

In this chapter, we have proposed a family of scheduling strategies for a DSMS and investigated them both theoretically and experimentally. We showed how a scheduling strategy impacts/affects the performance metrics such as tuple latency, throughput, and memory requirement of a CQ processing system. We proved that the PC strategy can achieve the overall minimal tuple latency. We also proved that the MOS strategy has the strictly optimal memory requirement, which further improves the near optimal memory requirement of the Chain strategy. Both the MOS strategy and the SS strategy demonstrate a much better tuple latency and throughput than the Chain strategy. The theoretical results were validated by an implementation where we performed experiments on all the strategies. The experimental results clearly validate our theoretical conclusions.

Furthermore, the threshold strategy inherits the properties of both the PC strategy and the MOS strategy, which makes it more applicable to a DSMS.

As part of ongoing work, we are considering to extend both the PC strategy and the MOS strategy to incorporate the predefined QoS specifications of the applications. Another problem we are investigating is the effective system capacity estimation of a CQ processing system over data streams, and how to use the estimation to guide a scheduling strategy to be aware of the overload situations and to further take effective actions such as load shedding and sampling to bring the system back to a normal state.

CHAPTER 5

LOAD SHEDDING

In this chapter, we focus on a fundamental problem that is central to a DSMS. Namely, we investigate the problem of `load shedding` during a temporary overload period. The load shedding is not a problem in a traditional DBMS because: (*1*) its queries are one-time queries; (*2*) its data sources are static data sets; (*3*) there is no support for QoS requirements for query processing. However, multiple CQs in a DSMS are active simultaneously for long periods of time. Theoretically, a CQ can be active in the system for ever. The input rates of streams are typically uncontrollable and highly dynamic in most stream-based applications. This highly dynamic input rate can prevent the system from keeping up with the tuple processing rate during a high input rate (or bursty) period. As a result, a large amount of unprocessed or partially processed tuples can be backlogged in the system, and tuple latency can increase and may not be bounded. Due to the predefined QoS requirements of a CQ, the query results that violate their QoS requirements are useless or even cause major problems. Therefore, we have to limit the number of tuples buffered in the system so that all final query results satisfy their predefined QoS requirements. A feasible and a viable solution to limit the number of buffered tuples in the system is to gracefully drop a portion of its unprocessed or partially processed tuples during high input periods, which consequently relieves system load, and makes it possible to satisfy all predefined QoS requirements. This graceful dropping process is called `load shedding`. It should also be noted that the accuracy of final query results is degraded as well due to the loss of tuples in a load shedding process. Fortunately, most stream-based applications can tolerate some inaccuracy in final query results if we are able to guarantee an upper-bound on the inaccuracy. Generally speaking, the `load shedding` problem is a problem of preventing final query results from violating predefined QoS requirements by discarding a portion of unprocessed or partially processed tuples gracefully. The predefined QoS requirements considered in this chapter mainly include the most-tolerable tuple latency (MTTL) and the most-tolerable relative error (MTRE) of a CQ in its final query results.

In this chapter, we propose a framework, and techniques for a general load shedding strategy by dynamically activating `load shedders` in query plans or deactivating existing load shedders based on the estimation of current system load. These load shedders drop tuples in either a randomized manner or using user-specified application semantics. Specifically, *1*) we exploit the optimal physical implementation of shedders in DSMSs with a goal of minimization of computation overhead and memory-consumption of a shedder. *2*) We then develop techniques to estimate the system load which implicitly determines when load shedding is needed and how much to shed. *3*) We develop algorithms to compute the optimal placement of a load shedder in a query plan. *4*) We also develop algorithm to determine how to distribute the total number of tuples (in terms of percentage) to be dropped among all load shedders with the goal of minimizing the total relative errors in the final query results due to load shedding. Finally, *5*) we conduct extensive experiments to validate the effectiveness and the efficiency of proposed load shedding techniques.

The rest of the chapter is organized as follows. Section 5.1 provides a formal definition of the load shedding problem. Section 5.2 discusses the detailed physical implementation of shedders. Section 5.3 describes our load estimation and load shedding algorithms. Section 5.4 presents a prototype implementation and the experimental results. Section 5.5 summarizes the chapter.

5.1 Problem Definitions

In a multiple query processing system over streaming data, various scheduling strategies have been proposed either to minimize the maximum memory requirement [29, 77] and tuple latency [77] or to maximize the throughput [119]. A scheduling strategy can improve the usage of the limited resources in such a system. However, it can never improve the maximal computational capacity of a system, which is inherently determined by its fixed amount of resources such as CPU cycles, size of RAM, and so on. When total load of active queries in a system exceeds the maximal computation capacity of the system, the query processing system has to either temporarily backlog some unprocessed or partial processed tuples that cannot be processed immediately in queues (or buffers) or discard them immediately. However, if the tuples are temporarily backlogged in queues, it causes a longer tuple latency, which is theoretically unbounded (as the queue size can grow indefinitely). This longer tuple latency is unacceptable in many stream applications where a near real-time response time requirement is critical. Fortunately, they can tolerate approximate results. Therefore, discarding extra tuples in the system is a natural choice to avoid a longer tuple latency.

In this thesis, we specifically address how to limit the number of tuples that can be backlogged in the system to satisfy both tuple latency and approximation requirements of final query results based on predefined QoS specifications. We call the scenario in which the system has to discard the tuples in order to prevent the system from violating the predefined QoS requirements of its queries as `query processing congestion`. The query processing congestion problem is very similar to the concept of `network congestion` used in the network community. The network congestion problem has been extensively researched in the computer network field. The techniques used to deal with the network congestion problem can be classified into two categories: (*a*) `congestion avoidance` techniques [102], which are used to prevent the system from entering a congestion situation and (*b*) `congestion control` techniques [123][70], which are used to bring a system back from a congestion situation once the system enters the congestion situation. These two techniques are complementary and are usually used together.

In this chapter, we present a general approach to `query processing congestion avoidance` for a DSMS to prevent the system from entering query processing congestion through dynamic activation of load shedders. This approach is different from those query congestion control techniques, which drop tuples only after they find the predefined QoS requirements have been violated through monitoring final query results. Specifically, we formulate the load shedding problem as follows:

Problem Definition 1. *Given a multiple query processing system with k active queries $\mathcal{Q} = \{Q_1, Q_2, \cdots, Q_k\}$ over n streams $\mathcal{I} = \{I_1, I_2, \cdots, I_n\}$, and each query with its predefined QoS requirement specified in terms of its most-tolerance tuple latency and its most-tolerance relative error of query results, the load shedding problem is to guarantee minimal computation resources, which make it possible to satisfy all predefined QoS requirements in the system by gracefully dropping some tuples, and at the same time, minimizing the relative errors in the system introduced by dropping tuples.*

The load shedding problem is an optimization problem, and consists of three sub problems, namely: (*1*) how to efficiently and effectively estimate the current system load, which implicitly determines when we have to do load shedding in order to avoid a query processing congestion. It also implicitly determines how much load we have to shed in order not to violate the predefined QoS requirements in the system; (*2*) how to find the best position for a potential load shedder (or shedders) that we have to insert into queries so that the introduced relative error in final query results can be minimized and at the same time the load saved can be maximized by discarding one tuple; (*3*) how to find an optimal allocation strategy for allocating the total shedding load among all non-active load shedders with a goal of minimizing the overall relative errors among all queries.Additionally, the problem of how to implement shedders so that the overhead introduced by the shedders themselves is minimized is also very important to minimize the impact of the load shedding process on system performance. We will first discuss what a shedder should be in order to minimize the overhead introduced by itself and then present our algorithms for the subproblems of the load shedding problem.

5.2 Load Shedders

Load shedders are fundamental elements in any load shedding system. Their main task is to shed load by discarding a number of tuples from the system. However, a load shedder itself has a significant impact on both system performance and consequently the accuracy of final query results. It affects system performance mainly because: *a*) the system load is very high when a system activates the load shedding mechanism. Any additional overhead introduced at this time affects both tuple latency and peak memory requirement. *b*) the actual load is shed through a number of load shedders collaboratively. The total load (or overhead) introduced by the load shedders can be substantial. By decreasing the overhead introduced by a load shedder itself, it is not only possible to increase effective system usage, but also improve the accuracy of final query results (less number of tuples will be discarded). Also different tuples may have different contribution to the final query results. It introduces more errors in final query results if it discards more important tuples. Therefore, it is desirable for a load shedder to shed as few tuples as possible which in turn reduces the errors in final query results. In the following, we discuss how a load shedder itself can achieve this desirable property. We will discuss how a load shedder can benefit from its placement along an operator path to achieve its desirable property in Section 5.3.2.

A load shedder can save more tuples by decreasing the additional overhead introduced by itself. Currently, a load shedder is considered as a `drop` operator in many

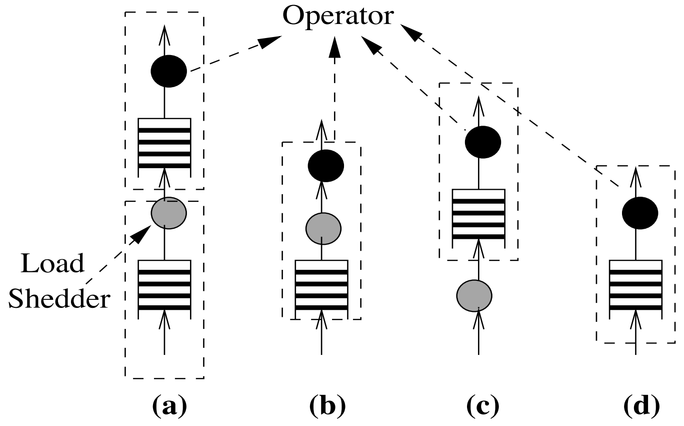

Figure 5.1 Load Shedders.

DSMSs. It is inserted into the current query network and deleted from the network dynamically according to the system load. This case is illustrated in Figure 5.1-a or 5.1-c. In case (a), the drop operator is considered a normal operator, its functionality is to determine whether to drop a tuple or not; its input queue is used to buffer its inputs because of bursty input modes in data streams. In this case, for each input tuple, we have to make a function call to determine whether to drop it or not. First, the total cost of hundreds (or thousands) of such function calls is likely to be high although the cost of each function call in itself is very small in current operating systems. Second, the extra memory required by its input queue increases the peak memory requirement. Finally, the scheduling cost increases as well because of the increase in the number of operators. In case (c), the drop operator is considered as a special operator. Since it does not have its own input queue, it cannot be scheduled as a normal object. It has to process its inputs whenever the inputs arrive, which means it has to be run as a thread. Although it does not require any extra memory, its processing cost increases due to the context switch cost of a thread. Therefore, it may not be a good choice to shed load by inserting a drop operator into the query network dynamically.

We consider shedding as one of functions of a normal operator in the system. There are two places where we can incorporate the shedding function into a normal operator: the operator and its input queue as illustrated in Figure 5.1-b and 5.1-d respectively. In Figure 5.1-b, the shedding function determines whether the input tuple needs to be passed to its operator. In this case, the processing cost is the same as that of case (a) and case (c). However, it does not introduce any additional overhead cost for scheduling because it is considered as part of a normal operator. In Figure 5.1-d, the shedding function is incorporated into its input queue. When its input queue receives a tuple, it accepts it only when the tuple passes the shedding function and the load shedding function is enabled. This case decreases the processing cost by incorporating the shedding function as part of enqueue function of its input queue. It also decreases the peak memory requirement because it discards the tuples immediately if those tuples will not be processed further. It does not incur any additional scheduling overhead either. Therefore, we argue that, of the four choices presented, the last case (Figure 5.1-d) is the best one to perform the actual load shedding.

A load shedder can introduce less error in final query results by discarding less important tuples. Two kinds of load shedders are proposed in the literature [114]: random shedders and semantic shedders. A random shedder is implemented as a p `gate` function; for a tuple, it generates a random value $\acute{p}$. The tuple passes to the next operator if $\acute{p} \geq p$; otherwise, it is discarded. A semantic shedder discards tuples that are less important to final query results. For example, in a continuous query for detecting fire alarm over a temperature stream, the lower values in that temperature stream are less important than higher values. Therefor, it is natural to drop lower values first in order to get less errors in final query results. A semantic shedder acts functionally as a select operator, which drops less important tuples based on a condition that has selectivity of $1 - p$ in order to drop p percent of its tuples. A discarded tuple by a semantic shedder introduces less error in final query results. However, it requires specific information from a query and its application domain in order to determine the relative importance of a tuple in a stream.

We propose four kinds of semantic shedders over a numerical attribute in our system: smallest-first shedders, largest-first shedders, center-first shedders, and outlier-first shedders. A smallest-first shedder discards the tuples with smallest values first; while a largest-first shedder discards the tuples with largest values first. The center-first shedder discards the tuples nearest to a user-specified center; while an outlier-first shedder discards the tuples that are farthest away from a user-specified center. When users submit queries, they can specify different load shedders based on their applications.

We support both types of shedders in our load shedding system. If a semantic shedder is applicable, we assume that there exists a function between the selectivity of the shedder and the relative error in the final query results. For all the other cases, we assume that all tuples in the same data stream have equal importance to the accuracy of the final results of a query.

5.3 Load Shedding Techniques

The query processing congestion avoidance techniques that we propose consist of two components: `system load estimation component` and `load shedding component`. The system load estimation component is used to estimate the actual computation load based on current input rates of input data streams and characteristics of active continuous queries registered in the system. This estimated load is used to determine when to activate the load shedding mechanism and how much load to shed once it detects a query processing congestion. The load shedding component is used to execute the actual load shedding, which includes compute the optimal location of load shedders in order to minimize error introduced and how to allocate the total load shedding requirement among non-active load shedders. The system load estimation component proposed is independent of the load shedding component, and can be used in conjunction with any load shedding approaches (e.g., the load shedding techniques proposed for aggregation queries [30]).

5.3.1 Prediction of Query Processing Congestion

Consider a general query processing system with m active queries in the system over n data streams (denoted by $I_1, I_2, \cdots, I_n$). Each query Q_i has its predefined QoS requirements specified by its maximal tolerant tuple latency L_i and its maximal tolerable relative error E_i in final query results. The actual computation load of such a system at a time instant is completely determined by the input characteristics of its data streams and the characteristics of the queries at that time instant in the system. Let us assume that we know[1] the current input rates v_i of input stream I_i. Then we can estimate its actual computation load for such a given query processing system as follows.

Without loss of generality, m active queries in the system can be further decomposed into k operator paths [2] $\mathcal{P} = \{p_1, p_1, \cdots, p_k\}$. To prevent a query from violating its predefined QoS requirements (i.e., L_i and E_i), we have to guarantee that the output results from each of its operator paths do not violate its QoS requirements. Therefore, we push the QoS requirements of a query down to each operator path that is encompassed in this query. As a result, each operator path has QoS requirements for the final results from this path. These QoS requirements are the same as those of its query. For operator path p_i, the query processing system has to process all the tuples that arrived during the

[1]Actually, we can measure them directly, which will be discussed in Section 5.3.1.2.

[2]In this chapter, an operator path means a simple operator path and a complex operator path is partitioned into multiple simple operator paths. Each simple path is defined as a unique path from the leaf node to root node.

last L_i time units in order not to violate the MTTL L_i no matter what the scheduling strategy is. It may schedule the operators along the path multiple times within that L_i time units, or schedule some operators of that path more often than others, but the age of the oldest unprocessed or partially processed tuple left in the queues along that operator path must be less than L_i. Therefore, without considering the cost of scheduling, its minimal computation time $\mathcal{T}_i$ required for the operator path to process all the tuples arrived within L_i time units is

$$\mathcal{T}_i = \frac{\int_{t-L_i}^{t} v_k(t) d_t}{\mathcal{C}_i}; \qquad 1 \leq i \leq k \tag{5.1}$$

where $v_k(t)$ is the input rate of its input stream at time instant t, and C_i is the processing capacity of the operator path, as defined in (4.2). The equation (5.1) gives the minimal absolute computation time units the operator path p_i requires within its MTTL. Furthermore, the percentage of computation time units ϕ_i it requires is,

$$\phi_i = \frac{\mathcal{T}_i}{L_i} \tag{5.2}$$

Equation (5.2) shows that the query processing system has to spend at least ϕ_i portion of its CPU cycles to process the tuples along the operator path p_i within every MTTL time units in order to guarantee that the query results do not violate its MTTL L_i.

Without considering shared segments among operator paths in the system, the total percentage of computation time units Φ for a query processing system with k operator paths is:

$$\Phi = \sum_{i=1}^{k} \phi_i \tag{5.3}$$

by plugging (5.1) and (5.2) into (5.3), it is easy to see that

$$\Phi = \sum_{i=1}^{k} \frac{\int_{t-L_i}^{t} v_k(t) d_t}{\mathcal{C}_i L_i} \tag{5.4}$$

Due to the fact that the MTTL of a query ranges over no more than a few seconds, we can expect that the input rate during a MTTL period of time can be captured by its mean input rate. Then equation (5.3) can be approximated as:

$$\Phi \approx \sum_{i=1}^{k} \frac{\bar{v}_k L_i}{C_i L_i} = \sum_{i=1}^{k} \frac{\bar{v}_k}{C_i} \tag{5.5}$$

where $\bar{v}_k$ is the mean input rate of the input stream of the operator path p_k within a period of time of its MTTL. It is also worth noting that the length of MTTL of an operator path does not have a direct relationship with the minimal percentage of computation time units it requires in order not to violate its MTTL requirements. Therefore, when two or more operator paths share a shared segment, we do not need to deal with the problem of which MTTLs we have to use to compute the minimal percentage of computation time units it requires.

If the estimated total percentage of computation time units is greater than 1.0, the system will definitely experience a query processing congestion and the tuple latencies of some or all query results will be longer than those specified in their predefined QoS requirements. In order to avoid such a query processing congestion, we have to drop some tuples so that the total percentage of computation load is always less than 1.0. The total load Δ we have to shed is given as follows:

$$\Delta = \begin{cases} \Phi - 1.0 & ; \quad if \;\; \Phi > 1.0 \\ 0 & ; \quad if \;\; \Phi \leq 1.0 \end{cases} \tag{5.6}$$

For example, if the estimated total percentage of computation time units requires $\Phi = 1.25$, this means the system is short of 25% of computation time units. We have to shed enough tuples so that we can release at least 25% of computation time units in order to avoid a query processing congestion.

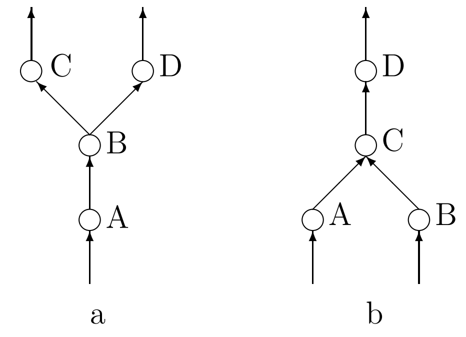

Figure 5.2 A shared Segment.

In (5.5), we have not considered sharing of segments by two or more operator paths. In a full-fledged DSMS, a shared segment introduced by a common subexpression between two queries is a common way of performance optimization. For example, when two operator paths ABC, ABD share a shared segment AB illustrated in Figure 5.2–a, the total percentage of processing time units that we computed into (5.5) for the shared segment $\{AB\}$ is $P(AB)$ given by

$$P(AB) = \frac{l_{ABC}\bar{v}_{AB}}{\mathcal{C}^S_{AB} l_{ABC}} + \frac{l_{ABD}\bar{v}_{AB}}{\mathcal{C}^S_{AB} l_{ABD}} = 2\frac{\bar{v}_{AB}}{\mathcal{C}^S_{AB}}$$

where $\mathcal{C}^S_{AB}$ is the processing capacity of the shared segment AB. But the system actually spends only $\frac{\bar{v}_{AB}}{\mathcal{C}^S_{AB}}$ portion of its computation time on the segment. Another $\frac{\bar{v}_{AB}}{\mathcal{C}^S_{AB}}$ portion of computation time is over estimated.

Similarly, if k operator paths share a shared segment, we have counted the processing time spent on that shared segment $k,\ k \geq 2$ times in (5.5), but the system only spends that processing time once, so we have over estimated it $k-1$ times.

Assume that there are g shared segments in the system, and that each of them is shared f_i times, where $f_i \geq 2$ and $1 \leq i \leq g$, the total percentage of computation time units over estimated due to those g shared segments is $\mathcal{D}$, where

$$\mathcal{D} = \sum_{i=1}^{g} (f_i - 1) \frac{\bar{v}_i}{\mathcal{C}_i^S} \tag{5.7}$$

and C_i^S is the processing capacity of the shared segment which can be computed from (4.2), and $\bar{v}_i$ is the mean input rate of the shared segment.

Therefore, by taking the shared segments into consideration, from (5.5) and (5.7), the total percentage of computation time units Φ:

$$\Phi \approx \sum_{i=1}^{k} \frac{\bar{v}_k}{\mathcal{C}_i} - \sum_{i=1}^{g} (f_i - 1) \frac{\bar{v}_i}{\mathcal{C}_i^S} \tag{5.8}$$

Equation (5.8) gives the approximate total percentage of the computation time units a query processing system requires given the input rates of its input data streams. From (5.6), we know that the system will experience a query processing congestion if the estimated total percentage of computation time units is larger than 100%. The portion of the percentage of the computation time units exceeding 1.0 is what we have to shed in order to avoid a query processing congestion. Note that the total percentage of computation time units estimated here is the minimal total percentage needed by the system. The overhead incurred by other tasks in the system such as scheduling, monitoring or estimating the input rates of data streams, and so on, are not explicitly considered here. However, those costs actually are taken into consideration when we compute the processing capacity of an operator. The more time a system spends on processing other tasks, the less is the process rate of an operator.

5.3.1.1 Algorithm For Finding A Shared Segment

Each operator is identified by a unique label in a query processing system. As a result, an operator path is labeled by a sequence of operator labels $\mathcal{X} = \{x_1, x_2, \cdots, x_n\}$, where $x_i, 1 \leq i \leq n$, is the label of i^{th} operator along the path from the leaf node.

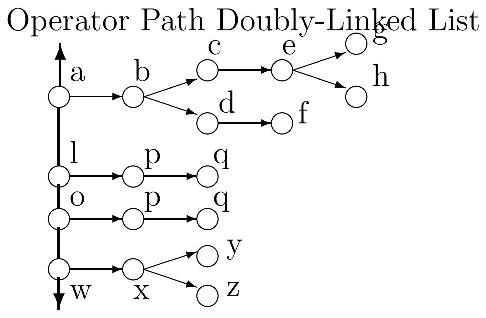

Figure 5.3 The Global Operator Path.

Hereafter, we call $\mathcal{X}$ the `path string` of the operator path. In a general continuous query processing system over streaming data, the shared segment between two or more operator paths exits due to query optimization through shared a common query expression among multiple queries. However, the shared relationship between two or more operator paths may change dynamically or periodically in a DSMS because of (a) addition of newly registered query plans or deletion of currently active query plans; (b) periodic revision of an active query plan in order to adapt to the changes of input characteristics of input data streams, which further causes deletion and addition of a shared segment. In order to handle the changes of the shared segments, we have developed an algorithm illustrated in Algorithm 4 which incrementally finds the shared segments and the number of times a shared segment is shared in the system. We first present the data structure used in the algorithm and then present our algorithm.

Data Structure: Each operator has two attributes $(label, frequency)$. The label attribute has its unique label in the system, and the frequency attribute maintains the number of times this operator is shared, which is initialized to zero. An operator path is represented as a linked list of its operator pointers starting from its leaf node. All operator paths are linked to a global doubly-linked path list illustrated in Figure 5.3 according to its path string. The operator paths in the global path link are sorted by the first label of its path string in order to find or delete an operator path efficiently. Once a new query plan is registered into the system, all its operator paths are inserted into the linked-list based on the first label of their path strings. For each operator path, if its first element does not exist in the list, all elements of that path are linked to the global list, and the frequency value of each operator along that path is set to one. If its first element is already in the list, it just increases the frequency value of the element by one, and then searches its second element along that branch, if found, increases its frequency value by one. Otherwise, its second element is linked to the global link list by creating another branch in the list. Its frequency value is set to one. Similarly, when an active query plan is deactivated, we have to delete the operator paths that is encompassed in it. To delete an operator path, we decrease the frequency value by one for each operator along that path. Whenever the value of frequency number of an operator is decreased to zero, the operator is deleted. Therefore, this global linked-list maintains all operator paths and all operators of all active query plans in the system.

Algorithm 4: Shared Segment Search Algorithm

```
   INPUT: $\mathcal{PCLL}$ – the global doubly linked list of operator path cluster;
   OUTPUT: $\mathcal{SSLL}$ – a set of shared segments with their frequency value;
1  $\mathcal{SSLL}$ = NULL;
   /*the following algorithm to maintain all shared segments and their
     frequencies.                                                        */
2  foreach (operator path cluster $\mathcal{PC}$ in $\mathcal{PCLL}$) do
3      if (the frequency value of the first operator $> 1$) then
4          $\mathcal{SSLL} = \mathcal{SSLL}$ UNION SEARCH($\mathcal{PC}$)
5      end
6  end
7  return SSLL;
```

The algorithm traverses the global doubly linked list and for each operator path cluster. It calls procedure SEARCH($\mathcal{PC}$) to find all shared segments of the operator path cluster $\mathcal{PC}$. If there exists shared segments in an operator path cluster, the frequency

Algorithm 5: SEARCH($\mathcal{PC}$)

```
INPUT: $\mathcal{PC}$ – the global doubly linked list of operator path cluster;
OUTPUT: $\mathcal{C}$ – a set of shared segments;
/*create a new set container for shared segments;                                   */
1  $\mathcal{C}$ = NULL;
2  if ($\mathcal{PC}$− >FirstOperator− >Frequency > 1) then
       /* Create a new shared segment $\mathcal{S}$ which consists of all elements before a
          fan-out element, including the fan-out element. The frequency value of
          the shared segment is the frequency value of the elements.                 */
3      Operator = $\mathcal{PC}$− >FirstOperator;
4      while ((Operator != NULL)&&(Operator− >numberOfChildren > 1)) do
5          $\mathcal{S} = \mathcal{S}$ + Operator;
6          Operator = Operator− >next;
7      end
       /* Add the shared segment $\mathcal{S}$ with its frequency value to the shared
          segment set $\mathcal{C}$.                                                 */
8      $\mathcal{C} = \mathcal{C} + \mathcal{S}$;
9  end
10 foreach (branch $\mathcal{B}$ of the fan-out element) do
       /*recursively CALL the search procedure SEARCH($\mathcal{B}$) and add the results
          to the set $\mathcal{C}$.                                                  */
11     $\mathcal{C} = \mathcal{C}$ UNION SEARCH($\mathcal{B}$);
12 end
13 return the shared segment set $\mathcal{C}$.
```

value of the first operator of the operator cluster must greater than one since the shared common subexpressions of two queries must start from the leaf operator of an operator path. The procedure SEARCH($\mathcal{PC}$) finds shared segments from a particular operator path cluster.

The cost of our algorithm is very compact. First, the algorithm has a linear time complexity of $O(n)$, where n is the number of active operators in the system. Second, the system does not need to execute the algorithm again until the global linked list is changed. In a continuous query system, it is unlikely that query plans are changed (addition, deletion, re-optimization) frequently due to the continuous computation requirement.

Finally, we only need to execute the algorithm for changed path clusters, and not for all path clusters in the system, when a query addition or deletion happens.

5.3.1.2 Predicting the Input Rate of Each Input Stream

In order to estimate system load, we have to know the input rates of all input streams. Therefore, we design a stream property monitor to monitor not only the arrival characteristics, but also the data properties of a data stream. By monitoring those properties in our system, we are able to: *i*) choose the most suitable scheduling algorithm, *ii*) optimize a continuous query plan adaptively, and *iii*) predict the system load and to shed partial load when a system is overloaded in order to satisfy the predefined QoS requirements. Here we only discuss related techniques that we use to monitor input rate of an input stream.

The characteristics such as the input rate of a data stream are highly dynamic. Some of them change regularly while others change irregularly. Typically, sensors that monitor the temperature send data periodically; while the robot which is in charge of house cleaning emits data whenever it detects that the floor is dirty and when it has done the house cleaning. To monitor[3] the input rate of an input stream efficiently, we sample a series of time points over the time axis. If the input rates at the last n points have not changed too much (the change is measured by the variance of those input rates), we decrease the sample rate; otherwise, we increase the sample rate. This adaptive algorithm makes it possible to monitor hundreds of data streams efficiently and effectively.

[3]Due to the high cost of continuous and precise monitoring of the properties of a data stream, we monitor the properties periodically and approximately which is sufficient to make a decision in most cases.

5.3.2 Placement of Load Shedders

As discussed in 5.2, a load shedder is considered as part of the input queue of a normal operator in order to minimize the overhead introduced by itself. Since each active load shedder incurs more or less extra load on a system, it is natural to decrease the number of load shedders in the system. For each operator path, at most one potential load shedder can be placed on it, which motivates us to find its optimal place among each operator path in the system. Once a query processing congestion occurs, it is desirable to activate as few of load shedders as possible, which motivates us to allocate the required shedding load among as few load shedders as possible. In the following section, the placement of a load shedder corresponds to the input queue of an operator. The loss/gain concept used in the following section shares some common characteristics as those introduced in [114] and in [30]. However, the methods used to calculate the gain and loss differ. Also our shedders have a maximal capacity, which is used to prevent too many tuples from violate the MMRE requirement of each operator path and this maximal capacity of a shedder is not present in any of them. Finally, we minimize the number of load shedders in the system through merging multiple shedders in query plans, which is not discussed in any of them either.

A load shedder can be placed at any location along an operator path. However, its location has a different impact on the accuracy of the final results, and on the amount of computation time units it releases. Specifically, placing a load shedder earlier in the query plan (i.e., before the leaf operator) is most effective in decreasing the amount of computation time units when a tuple is dropped, but its effect on the accuracy may not be most effective. On the other hand, placing a load shedder after the operator which has biggest output rate in the query plan has the lowest impact on the accuracy when a tuple is dropped, but the amount of computation time units released may not be the largest. Therefore, the best candidate location for a load shedder along an operator path is the place where the shedder is capable of releasing maximal computational time units while introducing minimal relative errors in final query results by dropping one tuple.

Without considering the shared segment among multiple operator paths, there are k candidate places to place a load shedder on an operator path $\mathcal{X}$ with k operators. Let $\{x_1, x_2, \ldots, x_k\}$ be its path label string, and v be the input rate of the data stream for this operator path. Let $b_1, b_2, \cdots, b_k$ be its k candidate places, where $b_i, 1 \leq i \leq k$ is the place right before the operator x_i.

We define the place weight W of a candidate place as the ratio of the amount of saved percentage of computation time units α to the relative error ϵ in its final results introduced by a load shedder at that place by discarding one tuple. The place weight W of a shedder at a particular location of an operator path with its QoS requirements as $(MTTL = L_i)$ and $(MTRE = E_i)$ is defined as:

$$\mathcal{W} = \frac{\alpha}{\epsilon} \tag{5.9}$$

$$\begin{aligned}
\alpha = & \quad \tfrac{v(d)}{\mathcal{C}^S} - \tfrac{v_{shedder}}{C^O_{shedder}} \\
\epsilon = & \quad \begin{cases} \frac{v(d)}{v_{shedder}} & \text{for a random shedder;} \\ f\left(\frac{v(d)}{v_{shedder}}\right) & \text{for a semantic shedder;} \end{cases} \\
v(d) = & \quad \begin{cases} E_i * v_{shedder} & \text{for a random shedder;} \\ E_i * f(\frac{1}{v_{shedder}}) & \text{for a semantic shedder;} \end{cases} \\
v_{shedder} = & \quad v \textstyle\prod_{i=x_1}^{x_n}(\sigma_i), x_1 \text{ to } x_n \text{are operators before the shedder}
\end{aligned}$$

where $\mathcal{C}^S$ is the processing capacity of the segment staring from the operator right after the load shedder until the root node (excluding the root node) along the operator path. If there is no operator after the shedder, $\mathcal{C}^S$ is defined as infinity and $(\frac{v(d)}{\mathcal{C}^S} = 0)$. The computation time units can be saved is zero and the shedder itself also introduces extra overhead by $\frac{v_{shedder}}{C^O_{shedder}}$. $v(d)$ is the maximal drop rate by a shedder at this place without violating the MTRE E_i defined for the operator path. $\frac{v(d)}{C^S}$ is the total computation time units it saves by dropping tuples at a rate of $v(d)$. However, a shedder also introduces additional overhead, which is $\frac{v_{shedder}}{C^O_{shedder}}$, to the system because it needs to determine whether a tuple should be dropped or not. $v_{shedder}$ is the input rate of the load shedder, and x_1 to x_n are the operators before the load shedder starting from leaf operator, and σ_i is the selectivity of the operator x_i. If there is no operator before the shedder, then $v_{shedder} = v$ and v is the input rate of the stream for the operator path $\mathcal{X}$. $\mathcal{C}^O_{shedder}$ is the processing capacity of the load shedder. If the load shedder is a semantic one, $f(.)$ is a function from selectivity of the shedder to the relative error in final query results.

In (5.9), the input rate of an input stream is the only item that changes over time. All the other items[4] do not change over time until we revise the query plan. Therefore, for the operator path $\mathcal{X}$, it has k candidate places for a load shedder. We compute the place weight for each of those k candidate places. And their partial orders do not change as input rate changes because all of them have the same input rate of the input stream at any time instant. The place where the load shedder has the biggest place weight is the most effective one. We only need to compute the weights of places once to find the best place of a load shedder for an operator path. Let $\mathcal{W}(p_i)$ be the weight of the place that a load shedder locates along the operator path p_i,

$$\mathcal{W}(p_i) = max\{\mathcal{W}(b_1), \mathcal{W}(b_2), \cdots \mathcal{W}(b_k), 0\} \tag{5.10}$$

where $b_1, b_2, \cdots, b_k$ are k candidate places along the path p_i. To prevent the case that a shedder introduces more overhead than it saves by dropping tuples at its maximal capacity, we add zero item in (5.10). If zero is the maximal place weight along the

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

operator path, it indicates that we cannot place a load shedder along this path because the additional overhead it introduces is bigger than what it saves.

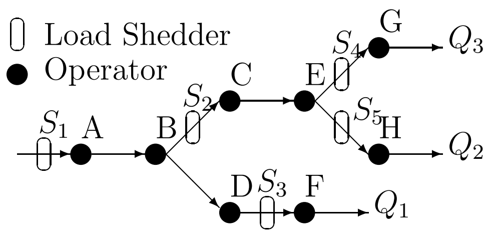

Figure 5.4 Locations Of Load Shedders.

For an operator path which shares a shared segment with other operator paths, placing a load shedder before or along the shared segment will not only impact the accuracy of the results of more than one query plan, but also has a different impact on them. For operator paths illustrated in Figure 5.4, the operator paths $\{ABDF\}$,which belongs to the query Q_1, $\{ABCEH\}$, which belongs to the query Q_2, and $\{ABCEG\}$, which belongs to the query Q_3, share the segment $\{AB\}$, and the last two operator paths further share the segment $\{ABCE\}$. Placing a load shedder anywhere along the segment $\{AB\}$ will impact all three query plans, and placing a load shedder anywhere along the segment $\{CE\}$ will impact query plans Q_2, Q_3, but not Q_1.

We consider all operator paths that are connected with each other through `fan-out` operators such as operator B, E as an `operator path cluster`. To determine the location of a load shedder in such an operator path cluster, we partition each operator path in this cluster into a set of operator segments. For each path, the first segment consists of all operators from its leaf operator to its first fan-out operator. Any other segments consist of all the operators between two fan-out operators plus the later fan-out operator. For example, the path cluster illustrated in Figure 5.4 is partitioned into a set of 5 segments $\{AB, CE, DF, G, H\}$. We compute the most effective position of a load shedder along each individual segment in the operator path cluster through (5.9) respectively. When we compute the most effective position of a load shedder along a shared segment such as $\{AB\}, \{CE\}$, the amount of percentage of computation time units saved is the sum of actual percentage of computation time units released by each segment starting from the operator right after the load shedder until the root operator. Similarly, the relative error introduced by the shedder is the sum of actual relative errors introduced in final query results of each path. For example, by placing a random shedder right before the operator A, the amount of percentage of computation time units released $\alpha = \frac{v(d)}{C^S_{ABDF}} + \frac{v(d)}{C^S_{ABCEG}} + \frac{v(d)}{C^S_{ABCDH}} - 2\frac{v(d)}{C^S_{AB}} - \frac{v_{shedder}}{C^O_{shedder}}$ and $\epsilon = 3\frac{v(d)}{v_{shedder}}$. Similarly, $\alpha = \frac{v(d)}{C^S_{CEG}} + \frac{v(d)}{C^S_{CEH}} - \frac{v(d)}{C^S_{CE}} - \frac{v_{shedder}}{C^O_{shedder}}$ and $\epsilon = f_{CEG}(\frac{v_d}{v_{shedder}}) + f_{CEH}(\frac{v_d}{v_{shedder}})$ if a semantic shedder is placed right before the operator C.

We have to merge two or more load shedders along an operator path cluster into one. The reasons for merging are: *a*) the most effective position of a load shedder along a segment does not mean the most effective position in its path cluster; *b*) different queries have different most-tolerant relative errors, which implies that different paths in an operator path cluster have different most-tolerant relative errors, therefore a shedder in a shared segment has different maximal capacity in terms of different paths; *c*) the overhead introduced by shedders can be decreased by decreasing the number of shedders. We consider three load shedders S_2, S_4, S_5 along the segments connected directly to the first fan-out operator E in Figure 5.4. If the place weight of the load shedder S_2 is the biggest one among them, the load we have to shed can be moved forward to the load shedder S_2 from load shedders S_4 and S_5. However, the maximal load that the load shedder S_4, and S_5 can shed respectively are different if the queries to which the segment $\{G\}$, and $\{H\}$ belong have a different most-tolerant relative error.

We define the `drop capacity` of load shedder s_i as $v(d) = D_i(e_i)$ such that $v(d)$ is maximized without violating its maximal tolerant relative error e_i. Before we merge load shedders of a path cluster, we have to initialize their maximal tolerant relative errors as follows. The load shedder among the segment, which outputs final query results, has its maximal tolerant relative error and this error is equal to the maximal tolerant relative error E_i of the query plan to which it belongs. All the other load shedders of this path cluster has a drop capacity of zero.

We start the merge procedure backward. We first process the fan-out operators closest to the root node, then process the fan-out operators closer to any of those processed fan-out operators. For a fan-out operator with m ($m \geq 2$) fans, let $W(S_{Main})$ be the place weight of the load shedder along the segment on which the fan-out operator lies, and this load shedder is termed `main-shedder`. Let $S_1, S_2, \cdots, S_k$ be the place weights of its k load shedders, which are the first load shedders along all the segments originating from the fan-out operator to root operator. Note that there are k, instead of m, load shedders along its m branches because some branches may not have a shedder, i.e, its place weight is zero, or some branches have more than one shedders at its branches or the shedders on some branches are eliminated as a result of merging process. Those shedders are termed `branch-shedders`. Let S_i be the place weight of the branch-shedder with the smallest most-tolerant relative error e_i among k branch-shedders. If $k * W(S_{Main}) <= \sum_{i=1}^{k} W(S_i)$, we eliminate the main-shedder since the total place weight that we gain at the Main shedders is no more than what we can get from branch shedders. Otherwise, considering each e_j of those k relative errors $e_1, e_2, \cdots, e_k$ associated with each branch-shedder, we maximize the drop rate $v(d)_j$ at the main-shedder such that the error introduced in final query results of the operator path at which S_i is located is no more than e_j. This maximized drop rate is limited such that the relative error introduced in final query results from each branch path does not more than the MTRE e_j of the branch. Notice, a drop rate at a main-shedder introduces a different relative error to each of its branch if some or all its branch-shedders are semantic shedders. Let $v(d)_s$ be the smallest one of the maximal drop rate $v(d)_1, v(d)_2, \cdots, v(d)_k$ and $\acute{e}_1, \acute{e}_2, \cdots, \acute{e}_k$ be the relative errors introduced by $v(d)_s$ on the paths on which its k branch-shedders lie. For each branch-shedder S_i, we shift $\acute{e}_i$ of its load shedding capacity to the main-shedder, and the corresponding capacity of a branch-shedder S_i decreases to $e_i - \acute{e}_i$. If $e_i - \acute{e}_i = 0$, its corresponding branch-shedder is deleted. After merging, one operator path cluster may have more than one shedder, each of them has a different load shedding capacity. This capacity implicitly determines its maximal drop rate there, which consequently determines the maximal load it can save.

Once the best location of a load shedder is determined, the load shedder function is incorporated into the input queue of an operator before we start to do load shedding. The shedder stays in non-active mode initially. All tuples bypass the shedder when it is in a non-active model, and there is no overhead during its non-active periods. Once it is activated, a tuple has to pass through the shedder before it can be processed by the next operator.

5.3.3 Allocation of Shedding Load Among Non-active Shedders

From the equation (5.6) in Section 5.3.1, we know when to shed load, and the total load we have to shed is Δ in order not to violate the predefined QoS requirements of active queries in the system. The algorithms presented in Section 5.3.2 gives a list of non-active load shedders and their load shedding capacities. Now, the problem is how to allocate the total shedding load among all or some of these load shedders by activating them with a goal of minimizing the maximum total relative error introduced by load shedding.

Let $\mathcal{S} = \{S_1, S_2, \cdots, S_m\}$ be the set of non-active shedders in the system. The allocation of shedding load problem is formalized to find a subset of shedders $\acute{\mathcal{S}}$, where $\acute{\mathcal{S}} \subseteq \mathcal{S}$, and to activate them such that

$$\forall i, S_i \in \acute{\mathcal{S}} \quad \text{and} \quad \begin{cases} \sum_{i=1}^{k} \alpha_i & = \Delta \\ \sum_{i=1}^{k} \epsilon_i & \text{is minimized, and } \epsilon_i \leq e_i \end{cases}$$

where α_i is the percentage of computation time units released by the load shedder S_i with a dropping rate of $v(d)_i$, and ϵ_i is the relative error it introduces by that dropping rate. Both of them are defined in (5.9).

The allocation problem is the well known `knapsack problem` by considering the total shedding percentage of computation time units Δ as the total capacity of a ship, and ϵ_i as the weight of the item, and α_i as the total value of the item. Although the 0-1 knapsack problem is a NP-hard problem, the fractional knapsack problem is solvable by a greedy strategy in $O(n \lg n)$ time. For the allocation problem, a load shedder does not have to shed load at its maximal dropping rate, but it can just shed part of its total capacity. Therefore, the allocation of shedding load problem can be solved in $O(n \lg n)$, where n is the total number of non-active shedders in the system.

To solve the problem of allocation of shedding load among the shedders when we detect a query processing congestion, we first sort all non-active shedders in the system by their place weights $W(S_i)$. Without loss of generality, we assume that $W(S_1) \geq W(S_2) \geq \cdots \geq W(S_x)$ after sorting. Obeying a greedy strategy, we activate the shedder with the greatest place weight, i.e., S_1, and let it shed load at a drop rate such that its saved load $\alpha = \Delta$ if $\Delta \leq \alpha_1$, where α_1 is its saved load when it operates at that drop rate, then we stop the procedure. Otherwise, let it work at its maximal drop rate without violating its capacity e_1, and update the total shedding load to $\Delta = \Delta - \alpha_1$. If the updated total shedding load $\Delta > 0$, we then activate the non-active shedder with the next greatest place weight. We repeat this procedure until the updated total shedding load becomes zero or there are no non-active shedders available in the system. If the procedure ends with a non-zero Δ, it indicates that the load shedding cannot guarantee minimal required computation resources to satisfy all predefined QoS requirements in the system. In this case, we have to either violate some QoS requirements or choose some victim query plans to deactivate them in order to meet QoS requirements of other query plans.

5.3.4 Decreasing the Overhead of Load Shedding

The overhead introduced by load shedding mainly consists of two parts. One is the overhead introduced by periodically estimating system load; another is the overhead of allocation of total shedding among all non-active shedders in the system. In order to decrease overhead introduced by estimating system load, we can increase the length of period of estimating system load when the previous estimated system load is far away from system capacity or the monitored input rates have not changed dramatically. Therefore, the overhead due to estimation of system load is very small if the overall system load is light and/or the input rates of all data streams changes gradually.

The second considerable overhead in load shedding problem is introduced by allocation of total shedding load among all non-active shedders. From (5.9), we can see that the place weight of a load shedder changes as the change of input rate of a data stream. Although this change does not change the partial order of shedders along the same operator path, it can change the partial order of shedders of two different operator paths when the ratio of the current input rate to the input rate used to compute the place weight changes dramatically. Therefore, we have to reorder the non-active shedders in the system when we allocate the total shedding load among all non-active shedders. In a heavy loaded system with dynamic input rates, we may have to frequently activate non-active shedders or deactivate active shedders, and this overhead is not negligible. In order to decrease the overhead of the $O(nlgn)$ solution of allocation problem, we have to decrease the number of shedders in the system. Instead of maintaining all partial orders of the whole list of shedders in the system, we partition n shedders into m groups, where $m \ll n$, according to their place weights. One shedder may upgrade to its upper level or degrade to its lower level group only when its input rate changes by a factor of its standard input rate that was used to calculate the place weight. This factor can be calculated in advance as a property of a shedder. Through maintaining those m groups of shedders in the system, we decrease the computation complexity of allocation problem to a constant. Namely, we activate the shedders in the first group and then shedders in the second group until the total saved load no less than what is required. The shedders within a group is selected based on their maximal drop capacity. The shedder with a bigger drop capacity is activated first. To promote or degrade a shedder to/from a different group, we only need to compute the ratio of the input rate of an operator path to its standard input rate if the load shedding mechanism is active.

5.4 Experimental Evaluation

5.4.1 Prototype Implementation

Currently, the load shedding techniques proposed in this chapter have been implemented within our QoS framework of proposed QoS driven DSMS, MavStream [76, 59, 107].

For load shedding, we have implemented both random and semantic shedders. Once a continuous query plan is registered with the system, the place weight of non-active shedders along each operator path of this query plan are computed, and then those shedders are classified into predefined groups according to their place weights in order to decrease the overhead introduced by allocation of total load shedding. The performance estimation subsystem estimates system load according to the input rate of each input stream from stream property monitor and then passes the estimated system load to load shedding module. The load shedding module works in either shedding-active mode, in which some shedders have been activated, or no-shedding-active mode, in which there is no active shedder in the system. If it works in no-shedding-active mode, it enters shedding-active mode once currently estimated load is bigger than system capacity. The load shedding module in-turn enables non-active shedders in highest-level group and/or next highest-level group until the total amount of saved percentages of computation time units is no less than the required shedding part. All active shedders are linked into a list according to their enabling orders. If it works in shedding-active mode and current estimated system load is less than system capacity by a minimal threshold value, the load shedding system deactivates an active shedder in the list in the reverse order of enabling it until there are no more active shedders in the system. All deactivated shedders are returned to their corresponding groups. Although our load shedding approach guarantees the minimal computation of resources required to deliver predefined QoS requirements, it is possible to violate a particular QoS requirement because of the shortage of a QoS guaranteed scheduling strategy. The following scheduling strategies have been implemented in the prototype to allocate all available resource among queries: the PC capacity scheduling strategy [77], which minimizes total tuple latency, the Chain strategy [29], which minimizes total queue size, and their hybrid – the threshold strategy [77], and other general strategies such as various-level round-robin strategies. We also implemented a QoS guaranteed scheduling strategy – Earliest Deadline First(EDF), which is guaranteed to deliver QoS requirement given guaranteed total computation resources.

5.4.2 Experiment Setup

In this section, we briefly discuss the results of various experiments that we have carried out to validate our load shedding techniques, and to show how our load shedding techniques along with our scheduling strategies satisfy the tuple latency requirement of a query plan. The system consists of a source manager, a query processing engine, and a run-time scheduling module, and a load shedding module. The source manager is in charge of the meta information of the tuples from each data stream. It also monitors the input characteristics and stream properties of an input stream. The query processing engine is in charge of query plan generation and execution. A set of relational operators such as `Project, Select` and `Window-based Symmetric Hash Join` have been implemented in our query engine. The run-time scheduling model is used to determine which operator or operator path to execute in any time slot. The load shedding module in our proposed system consists of a load estimation subsystem and our load shedding allocation subsystem. The load shedding module with our run-time scheduling model is capable of delivering tuple latency requirements.

Input data streams: The input we have generated are highly bursty streams that have the so-called self-similar property, which we believe resembles the situation in real-life applications. Each input stream is a super imposition of 64 or 128 flows. Each flow alternates ON/OFF periods, and it only sends tuples during its ON periods. The tuple inter-arrival time follows an exponential distribution during its ON periods. The lengths of both the ON and the OFF periods are generated from a Pareto distribution which has a probability massive function $P(x) = ab^a x^{-(a+1)}, x \geq b$. We use $a = 1.4$ for the ON period and $a = 1.2$ for the OFF period. For more detailed information about self-similar traffic, please refer to [85, 25]. In our experiment, we use 5 such self-similar input data streams with different mean input rates.

Experimental query plans: All of our queries are continuous queries that consist of `select, project`, and `symmetric hash join` operators. To be more close to a real application, we run 16 actual continuous queries with 116 operators over 5 different data streams in our system. The selectivity of each operator is widely distributed ranging from 0 to 1. Both the selectivity and the processing capacity of each operator can be determined by collecting statistics periodically during run time. The list of queries used for the experiments is presented in Appendix A. We group 16 queries into 3 groups with different QoS requirements in terms of maximal tolerant tuple latency and maximal tolerant relative error. The first group has QoS requirement of (tuple latency ≤ 0.6, relative error ≤ 0.15); the second and third group have a QoS requirement of (1.0, 0.25) and (1.5,0.5) respectively. All the experiments were run on a dedicated dual processor Alpha machine with 2GB of RAM. One of the processors was used to collect experiment results while another processor was used for query processing.

5.4.3 Experimental Results

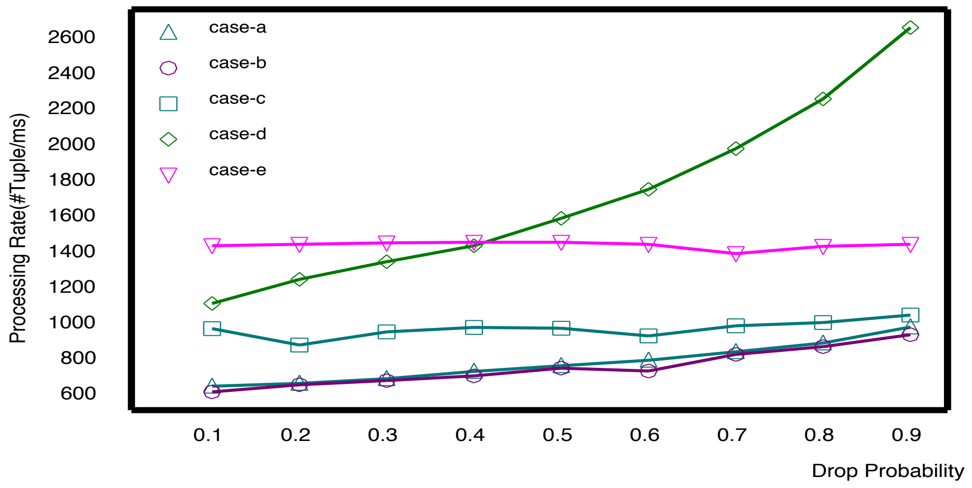

Figure 5.5 Cost of Load Shedders.

Cost of load shedders: Our first experimental results compare the cost of four different load shedders discussed in Section 5.2. Figure 5.5 shows the processing rate of a select operator with or without a load shedder as the drop probability of the shedder changes. Case-{a,b,c,d} in Figure 5.5 corresponds to the select operator with four (a,b,c,d) different implementation of a shedder discussed earlier in Figure 5.1. Case-e corresponds to the cost of the select operator without a load shedder. The processing rate is the number of tuples consumed by both shedder and select operator per millisecond. This graph shows that case-d performs much better than all the other cases. As drop probability of the shedder increases, the processing rate in case-d increases much faster than other cases. This is mainly because of the lower cost of discarding a tuple in this case as compared to the others. The processing rate of case-e remains flat because it does not have a load shedding function. All other cases (a,b,c) shows that the cost of the shedder is much bigger than that of a select operator. and Those implementations do not release any load for a select operator, instead of introducing extra overhead and errors.

Load Shedding with Path capacity strategy: This set of experiments

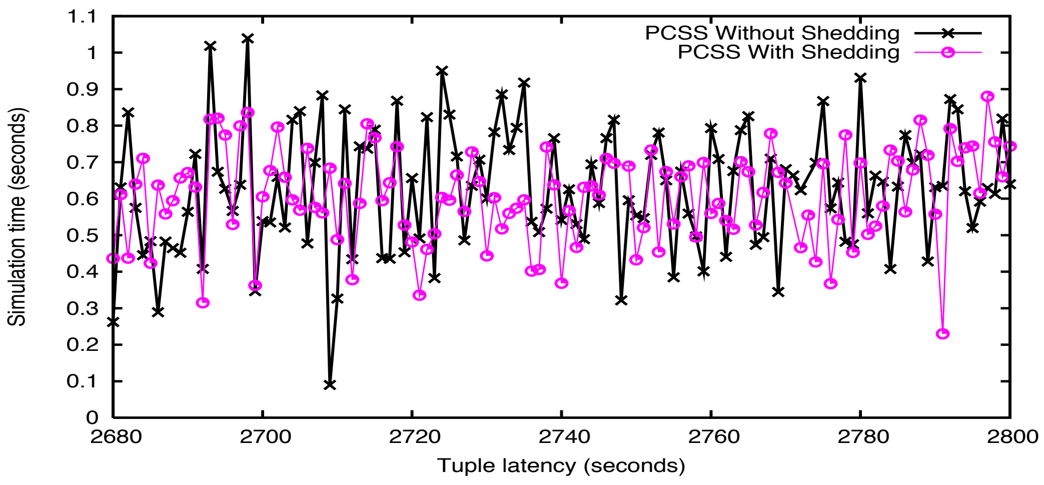

Figure 5.6 Tuple Latency Under Path (group1).

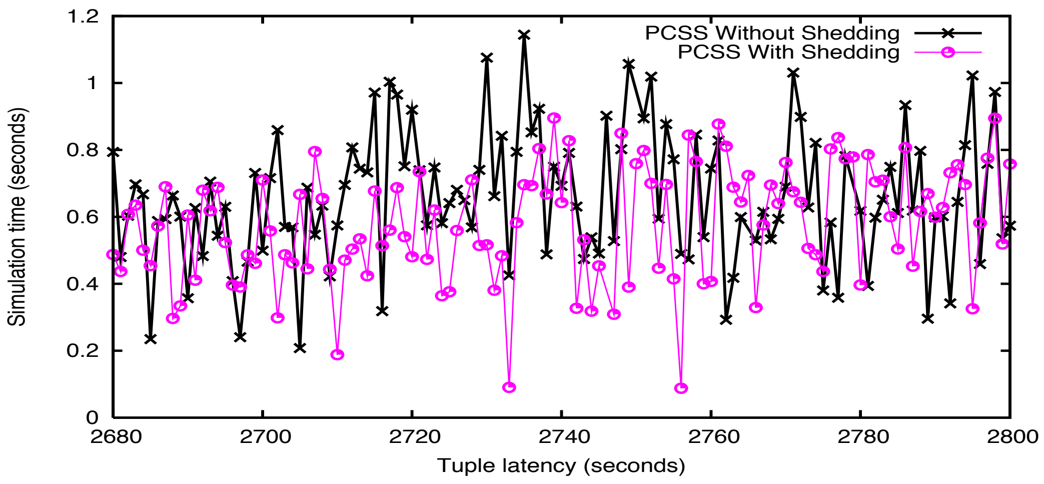

Figure 5.7 Tuple Latency Under Path (group3).

shows the performance of system under our load shedding techniques with the PC scheduling strategy. Figures 5.6 and 5.7 show the tuple latency of all output tuples of a query in group 1 and group 3 respectively. The tuple latencies are larger than the maximal tolerant tuple latency requirement in group 1; while those in Figure 5.7 are less than their maximal tolerant tuple latency requirement in group 3. This is because the path scheduling strategy is not a QoS guaranteed scheduling strategy though there are enough resources for all queries. However, the maximal tuple latencies under the PC strategy with load shedding are less than those without load shedding.

Figures 5.8 and 5.9 show the moving average of all tuples from a query in group and the total internal queue size respectively in order to see the impacts on accuracy of final query results by load shedding. From both figures, we can see that the peak values of both the average value and the total memory requirement under load shedding are decreased when system experiences a query processing congestion. This shows our load shedding techniques detect congestion correctly and shed corresponding load effectively.

Load Shedding with EDF scheduling strategy:

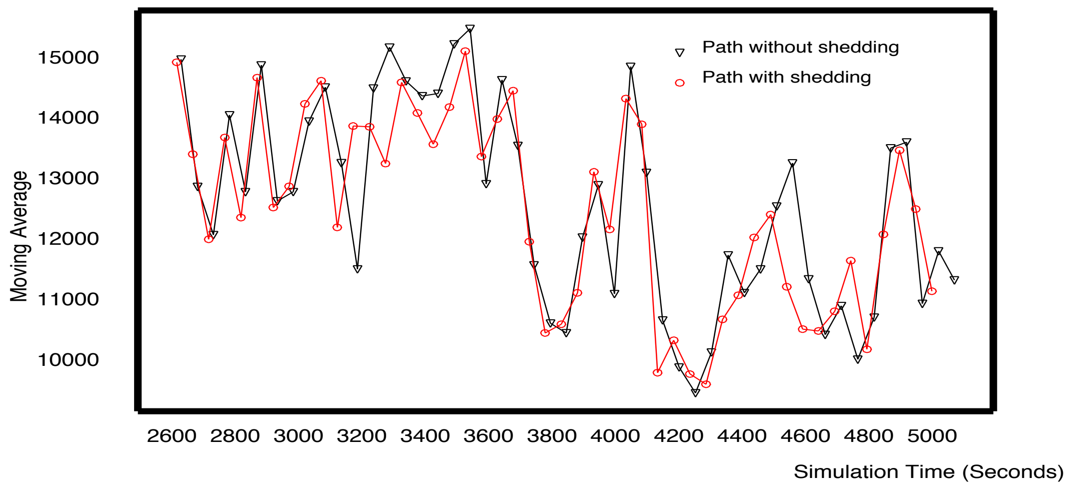

Figure 5.8 Moving Average Under Path.

This group of experiments show the performance of our load shedding technique with Earliest Deadline First strategy. The EDF schedules the operator path with earliest deadline, the deadline of an operator path is defined as the age of the load tuple along the path plus its maximal tolerant tuple latency. The EDF is a QoS guaranteed scheduling strategy in terms of tuple latency. Figure 5.10 clearly shows that the tuple latencies under EDF with load shedding do not violate the predefined maximal tolerant tuple latency, and those latencies are less than those under EDF without load shedding. Figure 5.11 shows that the total internal queue size under EDF scheduling with/without load shedding. Both figures show that the query processing congestions under EDF come earlier than under PC strategy. This is because the overhead introduced by EDF itself is much bigger than that introduced by Path. This means the system capacity decreases under EDF, and the number of tuples it can backlog without violating predefined QoS decreases as well.

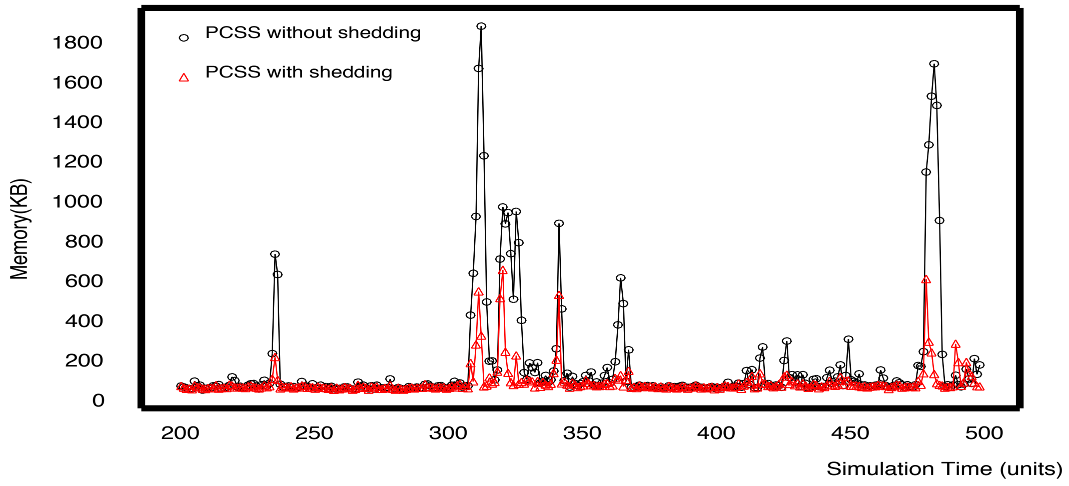

Figure 5.9 Total Internal Queue Size Under Path.

5.5 Summary

In this chapter, we have proposed a general purpose query processing congestion avoidance framework and load shedding techniques for a DSMS in order to meet all QoS requirements. We first presented our system load estimation technique, which can be coupled with any other load shedding techniques, based on input rate of data streams. The estimated system load provides sufficient information to determine when to shed load and how much to shed. Second, we discussed how to allocate total load shedding to shedders. Specifically, we provide solution to the following problems: where a load shedder should be placed in a query plan, and how to allocate total shedding load among those load shedders with a goal of minimizing total relative errors introduced into the final query results. Finally, we implemented a prototype of our proposed load shedding approach within our data stream management system, and conducted various experiments to evaluate our load shedding techniques.

We are continuing work on QoS guaranteed scheduling strategies to provide a complete QoS delivery framework for a general-purpose data stream management system. We are also exploiting the techniques to guarantee QoS delivery of various ECA rules.

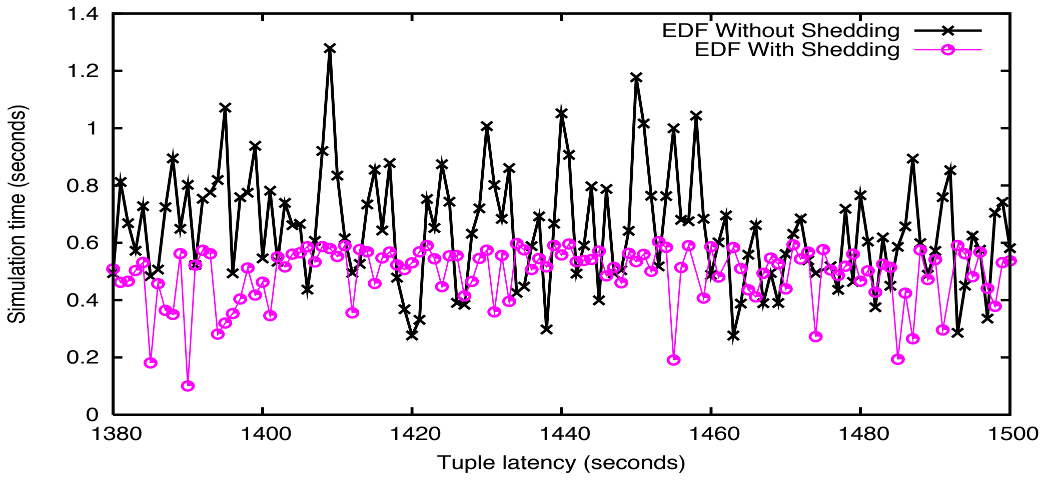

Figure 5.10 Tuple Latency Under EDF (group1).

One of main functions of a DSMS is to provide enhanced active capability, which not only requires the processing of continuous queries with QoS requirements, but also needs to detect various complex events and trigger corresponding actions with predefined QoS requirements.

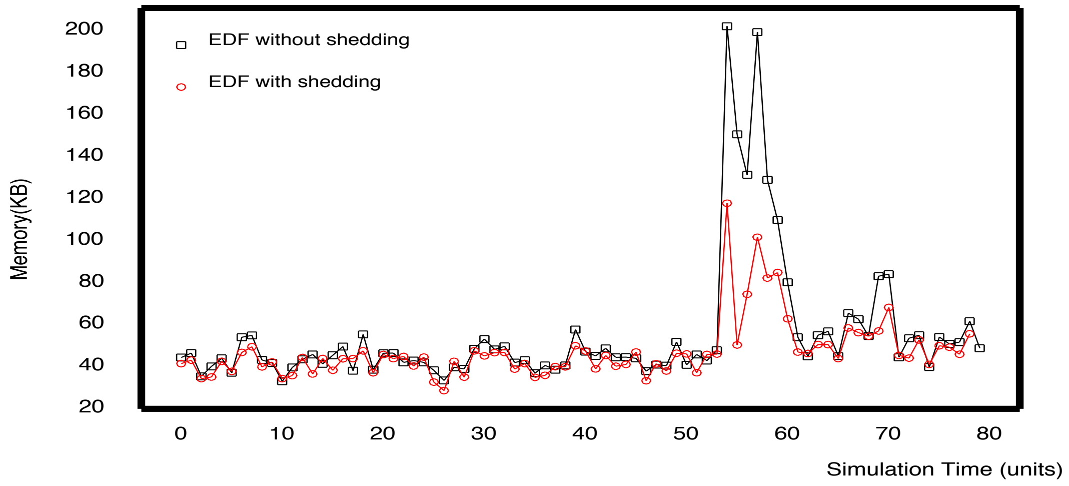

Figure 5.11 Total Internal Queue Size Under EDF.

CHAPTER 6

EVENT AND RULE PROCESSING

Event processing in the form of ECA rules has been researched extensively from the situation monitoring viewpoint to detect changes in a timely manner and to take appropriate actions. Several event specification languages and processing models have been developed, analyzed, and implemented. More recently, data stream processing has been receiving a lot of attention to deal with applications that generate large amount of data in real-time at varying input rates and to compute functions over multiple streams that satisfy QoS requirements. A few systems based on the data stream processing model have been proposed to deal with change detection and situation monitoring. However, current data stream processing models lack the notion of composite event specification and computation, and they cannot be readily combined with event detection and rule specification, which are necessary and important for many applications.

Although research seems to address these two as separate topics, there are a number of similarities and differences between the two models. We argue that for many of the applications considered for stream processing, event and rule processing are needed and are not currently supported. On the contrary, event processing systems concentrate on complex event and rule processing in a DBMS environment and do not consider complex stream processing. By synthesizing these two and combining their strengths, we argue that the integrated model will be better than the sum of its parts. In this chapter, we first summarize the characteristics of both threads of work to set the stage for understanding the differences and similarities. We then propose an integrated model that combines the two using a uniform computation model. We also introduce the notion of a semantic window, which significantly extends the current window concept for CQs, and stream modifiers in order to extend current stream computation model for complex change detection. We further discuss the extension of event specification to include CQs. Finally, we discuss our integrated model and its implementation. The extensions proposed in this chapter are critical for integrating the two computation models in a seamless manner. The ability to use CQs as events and the extended specification of CQs and events are also needed for the integration of the two.

The rest of the chapter is organized as follows. Section 6.1 discusses the necessary and importance of an integrated model for event and rules processing over data streams. Event processing model is explained in Section 6.2. Detailed comparisons of event processing and data stream processing models are presented in Section 6.3. Integrated model is presented in Section 6.4. Finally, we summarize out work in event and rule processing under the context of data stream processing and future directions in Section 6.5.

6.1 Introduction

Event processing [45, 105, 47, 56, 53, 82, 31, 37, 63, 87, 106, 50, 48] and lately data stream processing [24, 32, 41, 89, 76, 95] have evolved independently based on situation monitoring application needs. Several event specification languages [57, 58, 53, 54, 38, 103, 9, 10] for specifying composite events have been proposed and triggers have been successfully incorporated into relational databases. Different computation models [55, 87, 52, 53, 50, 31, 37, 48] for processing events, such as Petri nets [52, 53], extended automata [55, 87, 58], and event graphs [31, 37, 50] – have been proposed and implemented. Various event consumption modes [31, 52, 53, 37, 38] (or parameter contexts) have been explored. Similarly, data stream processing has received a lot of attention lately, and a number of issues – from architecture [32, 89, 76, 41, 98] to Quality-Of-Service (QoS) [114, 20, 44, 77, 29, 33] – have been explored. Although both of these topics seem different on the face of it, we argue that there are more similarities than differences between them. Not surprisingly, the computation model used for data stream processing is not very dissimilar from some of the event processing models (e.g., event graph), but used with a different emphasis.

As many of the stream applications are based on sensor data, they invariably give rise to events on which some actions need to be taken. In other words, many stream applications seem to not only need computations on streams, but also these computations generate interesting events (e.g., car accident detection and notification, network congestion control, network fault management [78], and intrusion detection) and several such events may have to be composed, detected and monitored for taking appropriate actions. Currently, to the best of our knowledge, none of the work addresses the specification and computation of the above two threads of work. Our premise for this chapter is that although each one is useful in its own right, their combined expressiveness and computation power are critical for many applications of data stream processing. Hence there is a need for synthesizing the two into a more expressive and more powerful model that combines the strengths of each one.

We use the following running example to explain the current limitations of each model, and the need for the integrated model. Consider the following car accident detection scenario that is slightly different, but more effective, from the linear road benchmark [15].

Example 1 (Car ADN). *In a car accident detection and notification system, each express way in an urban area is modeled as a linear road, and is further divided into equal-length segments (e.g., 5 miles). Each registered vehicle on an express way is equipped with a sensor and reports its location periodically (say, every 30 seconds). Based on this location stream data, we can detect a car accident in a near-real time manner. If a car reports the same location (or with speed zero mph) for four consecutive times,* *FOLLOWED BY at least one car in the same segment with a decrease in its speed by 30% during its four consecutive reports, then it is considered as a potential accident. Once an accident is detected, the following life saving actions may have to be taken immediately: i) notify the nearest police/ambulance control room about the car accident, ii) notify all the cars in* 5 *upstream segments about the accident, and iii) notify the toll station so that all cars that are blocked in the upstream for up to* 20 *minutes by the accident will not be tolled.*

Every car in the express way is assumed to report its location every 30 seconds forming the primary input data for the above example. The format of car location data stream (i.e., *CarLocStr*) is given below:

```
CarLocStr(
    timestamp,    /* time stamp of this record   */
    car_id,       /* unique car identifier       */
    speed,        /* speed of the car            */
    exp_way,      /* expressway: 0..10           */
    lane,         /* lane: 0, 1, 2, 3            */
    dir,          /* direction: 0(east), 1(west) */
    x-pos);       /* coordinates in express way  */
```

CarSegStr is the car segment stream (or the input *CarLocStr* stream), but with the location of the car replaced by the segment corresponding to the location. The query shown below produces the *CarSegStr* from the *CarLocStr* stream.

```
SELECT timestamp, car_id, speed, exp_way, lane, dir,
  (x-pos/5 miles) as seg FROM CarLocStr;
```

Detecting an accident in the above Car ADN example has three requirements, and they are (1) IMMOBILITY: checking whether a car is at the same location for four consecutive time units i.e., over a 2 minutes window, in our example, as the car reports its location every 30 seconds. (2) SPEED REDUCTION: finding whether there is at least one car that has reduced its speed by 30% or more during four consecutive time units.

and (3) SAME SEGMENT: determining whether the car that has reduced its speed (i.e., car identified in (2)) is in the same segment and it follows the car that is immobile (i.e., car identified in (1)). Immobility of a car can be computed using CQs that are supported by the current data stream processing systems as shown below:

```
SELECT    car_id, AVG(speed) as avg_speed
FROM      CarLocStr [2 minutes sliding window]
GROUP BY  car_id
HAVING    avg_speed = 0;
```

With the current event and stream processing models, using a declarative language[1], it is difficult or impossible to efficiently compute the speed reduction. Whether the cars that are found in requirements (1) and (2) are from the same segment can be readily determined in an event processing model using a `sequence` operator [53, 31, 9]. Notifications or life saving actions have to be taken once the cars are identified, which is not supported by current stream processing model as well. As the cars that are identified in requirement (3) can be separated by more than 4 time units, it requires an efficient, meaningful and less redundant manner for notifications. In other words, number of times the accident is reported should be kept to a minimum. The above can be done efficiently using the current event processing models, but not the current stream processing model. Although a JOIN operator can be used to compute it, the number of reports is not minimized.

As shown, all the above requirements strongly call for an integrated model. Furthermore, the second and third requirements pose challenges for synthesizing an integrated model. We will later illustrate how the above can be specified elegantly and be computed efficiently using the integrated model proposed in this chapter. Some of the earlier work

[1]Models that are based on procedures may compute this, but they are more difficult to use than those models that are based on non-procedural (e.g., SQL) or declarative languages. In this chapter, we consider the latter one.

on sequence processing [106], temporal aggregation [97], and trigger processing in Ariel [63] address some computational and performance aspects that may be relevant to the extensions proposed in this chapter. However, they were not in the context of streams and they addressed processing from event logs in the relational context without the notion of a window. In addition, performance work did not address either the real-time aspects or the QoS constraints.

6.2 Event Processing Model

Event-Condition-Action (ECA) rules are used to process event sequences and to make the underlying system active for applications such as situation monitoring, access control, and change detection. They consist of three components and they are 1) Event: occurrence of interest such as data-manipulation-events, clock-events, and external-notification-events, 2) Condition: can be a simple or a complex query, and 3) Action: specifies the operations that are to be performed when an event occurs and the corresponding condition evaluates to be true. ECA rules can be defined either at application level or at system level. A number of event processing systems using ECA rules have been proposed and implemented in the literature ACOOD [50], ADAM [47], Alert [105], Ariel [63, 64], COMPOSE [56], Hipac [45], ODE [55, 87], REACH [31], Rock & Roll [48], SAMOS [52, 53], Sentinel [37, 38], SEQ [106], UBILAB [82], and [96, 97]. A comprehensive introduction and description about most of these systems can be found in [121, 101].

Primitive or simple events are specific to a domain and are predefined. On the other hand, composite or complex events are composed of more than one primitive or composite event forming an event expression using event operators. Event operators [57, 58, 53, 54, 38, 103, 9, 10] are used to compose events and can be unary, binary, or ternary based on the number of operands.

Primitive events are detected by the underlying system at the time of occurrence. For example, the time of occurrence of an event can be the time at which a method is invoked by an object. Composite events are detected based on the event operator semantics when all of its constituent events occur. The time of occurrence of a composite event depends on the event operator semantics and detection semantics [51].

Consider two primitive events `DS1` (Data from sensor 1) and `DS2` (Data from sensor 2), a composite event `DS1` **AND** `DS2`, along with the event occurrences of DS1 at 10 a.m. and DS2 at 11 a.m. This composite event is detected when event DS2 occurs (i.e., at 11 a.m.], since event DS1 has started the AND event occurrence, but the time of occurrence of the composite will be a time point 11 a.m. in detection-based semantics, and will be an interval [10 a.m. - 11 a.m.] in interval-based semantics.

Several approaches have been proposed for the detection of composite events in the literature and they are: event detection graphs (EDGs) [50, 31, 38], extended finite state automaton [55, 87], colored Petri nets [54, 52, 53], and event algebra [82]. EDGs have been shown to be based on operators rather than instances and hence are efficient as compared to other approaches based on the computation and storage requirements for detecting events.

In this chapter, we draw upon EDGs for our integrated model as it corresponds to operator trees and has similarities with respect to query processing over data streams whereas the other representations do not share these characteristics with query processing. We also use the masking capability introduced in ODE [58] to filter events on arbitrary conditions.

Event Detection Graphs: In an EDG, leaf nodes represent primitive events and internal nodes represent composite events (or event operators) and event occurrences flow in a bottom-up fashion. Figure 6.1-b shows a composite event AND with two events E1 and E2. Whenever there is an occurrence of event E1 or E2, it is propagated to the

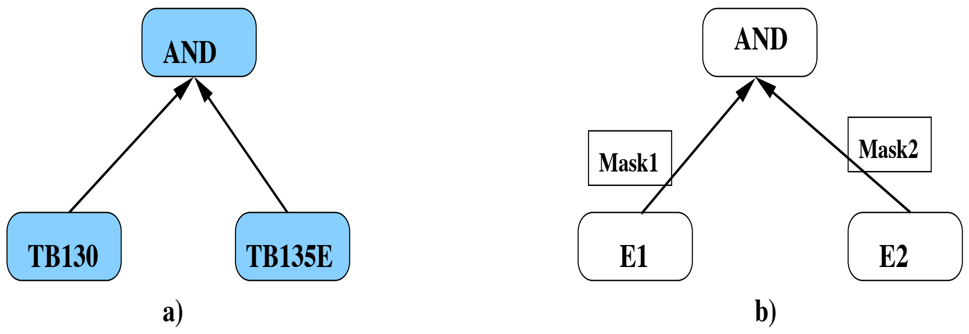

Figure 6.1 a) EDG without Masks; b) EDG with Masks.

AND node. An AND event is detected whenever both its constituent events occur, and it intuitively represents the Cartesian product (or JOIN) operation in a DSMS with a window size of one tuple. In addition, as shown in Figure 6.1-b EDGs can also support masks (to filter event occurrences) while events are propagated. For example, there can be a mask on event E1 such that it only propagates events from 11 a.m. to 1 p.m. Masks/predicates from conditions can also be pushed into events thereby saving propagation of unnecessary events up to the event graph.

Event Consumption Modes: The AND event shown in Figure 6.1-b is detected when both the events E1 and E2 occur. All event occurrences of event E1 and E2 are combined, and none of them are removed after they have been taken part in event detection.

Unconstrained event operator semantics correspond to the unrestricted (or general) context. This means events, once they occur, cannot be discarded at all. For a sequence event operator [53, 31, 36], all event occurrences that occur after a particular event will get paired with that event as per the unrestricted context semantics. In the absence of any mechanism for restricting event usage (or consumption), events need to be detected and parameters for those composite events need to be computed using the unrestricted context definitions of the event operators. However, the number of events produced (with unrestricted context) can be large and not all event occurrences may be meaningful for an application. In addition, detection of these events has substantial computation and space overhead that may become a problem for situation monitoring applications. Event consumption modes (or contexts) are supported by several active database systems, such as *ACOOD*, *SAMOS*, *Sentinel*, and *REACH* for restricting the unnecessary events from being detected.

Consider the application in which the Texas Department of Transportation wants to monitor "Whether there is a *traffic block involving Interstate 30* and *traffic block involving Interstate 35E* in Dallas". This requirement can be mapped to a complex event expression using an AND operator (i.e., "I30 block in Dallas (TB_I30)" **AND** "I35E block in Dallas (TB_I35E)"). EDG for the above example is shown in Figure 6.1-a. Whenever there is a traffic block in I30 or I35E it is propagated to the AND node. Complex event AND is detected at the internal node whenever both the child nodes propagate the corresponding events. Since an AND event is detected whenever both its constituent events occur, it intuitively represents the Cartesian product (normal JOIN) operation in a DSMS with a window size of 1 tuple.

As shown in Figure 6.1-b, EDGs can also support masks (to filter event occurrences) while events are propagated. For example, Texas Department of Transportation wants to monitor "Whether there is a *traffic block in Interstate 30* and *traffic block in Interstate 35E* in Dallas from 6.30 a.m. to 9 a.m., from 11 a.m. to 1 p.m., and from 4 p.m. to 6 p.m.". This can be done using the event mask which can filter out the events that does not occur over those time intervals. Masks can also be optimized by pushing them into the events thereby saving propagation of unnecessary events up the event graph.

In real-world applications, all these combinations (i.e., unrestricted) of events may not be interesting and in order to restrict the detection to meaningful composite events, event detection (or consumption) modes were introduced. Event detection modes place restrictions on the pairing of events to detect a composite event. For example, Sentinel supports the following event consumption modes (also known as parameter contexts): Recent, Proximal-Unique [49], Chronicle, Continuous, and Cumulative.

Event Detection Stages: Typically composite event detection is done in two stages: *i*) Merging of events in composite event nodes, and *ii*) Applying conditions on detected events. In the first stage of event detection, events are merged based on the event operator semantics that is in turn based on the time of occurrence of its constituent events. In the second stage detected events are checked for conditions that are specified in an ECA rule and the corresponding actions are taken. Conditions can be function (or method) calls with all the event properties as parameters.

6.3 Analysis of Event Vs. Stream Processing

Processing of events using event detection graphs (analogous to a query tree) and a data flow architecture, is similar to the processing of data streams. In this section, we analyze the relationship between event processing and data stream processing models. This forms the basis of our integrated model that combines the strengths of both.

6.3.1 Inputs and Outputs

Inputs (or data sources) to an event processing model are a stream of events (or event histories). Event streams are considered as a sequence of events that are ordered by their time of occurrence. Most of event streams considered by event processing models are generated internally by the underlying system such as the data manipulation operations (i.e., INSERT, UPDATE, and DELETE) in RDBMSs, system clock, etc. The input rate of an event stream is not assumed to be very high and highly bursty as they are generated by the underlying system. Also the event items of an event stream are well-defined tuples and have a simply data structure, which usually consists of primitive data types. The outputs of event operators also form event streams, which are ordered by their occurrence timestamps (may be an interval for composite events). Once the events are detected, they are used to trigger a set of ECA rules. Once the ECA rules are triggered, necessary conditions are checked and predefined actions are taken.

Main inputs to a data stream processing model are data streams. Input tuples in a data stream can be ordered by any attribute and not always by the timestamp (e.g., sequence_id of a TCP packet in a TCP packet stream) as in the case of event sequences. In addition to streams, a data stream processing model can also include processing of static relations, which are not typically supported in an event processing model (although conditions and actions can access stored relations). Furthermore, the input characteristics of data streams are highly unpredictable and dynamic (e.g., bursty). The data items from a data stream can be as complex as an unstructured message or document. The main output of a data stream processing model, if one of its inputs is a data stream, is a data stream too. Thus, conceptually, both the models have similar inputs and outputs. However, the data sources in data stream processing model are mostly external sources with high input rates and highly bursty input model, where as the data sources in event processing model are mainly internal ones with relatively low input rates.

6.3.2 Consumption Modes Vs. Windows

Event consumption modes or contexts were introduced primarily to determine how many events from the same event sequence should be kept for the purpose of detecting composite events. The number of events to be kept depended solely on the context of the operator and the semantics of the operator. For example, for the sequence operator, one could not drop any event from the past if no context was associated (i.e., use of general context). On the other hand, for the OR operator, that is not the case. Event contexts indirectly keep a small portion of (the head of) an event sequence based solely on the value of timestamp and the context. Also, the set of event instances retained at a binary operator depend upon the dynamics of both event streams. For instance, if the completing/terminating event did not occur, event instances for the other operator would accumulate for some contexts such as chronicle and continuous.

In contrast, the notion of a window in stream processing is defined on each stream and does not depend upon the operator semantics. Also, the window need not be defined only in terms of either time or physical number of tuples, although that is typical in most of the applications. The objectives of defining a window in stream processing are *1*) to convert blocking operators into a non-blocking computation, and *2*) to produce output in a continuous manner. Hence, the window generated by a context is not the same as the window concept in streams.

6.3.3 Event Operators Vs. CQ Operators

Event operators are quite different from the operators supported by current stream processing model. Event operators are mainly used to express and define the computation on event (tuple) level and to reduce the number of output events through consumption modes or profiles, and they solely use the timestamp of an event for detecting composite events. For example, *AND* operator in the event processing model is used to express and compute the occurrence (appearance) of two events (tuples) from its two input event streams. Thus, event operators do not perform any computation over the attributes of these events (tuples). On the other hand, current stream processing operators are mostly modified relational operators [2], which focus on how to express and define the computation at the attribute level, rather than the tuple level. Additionally, stream

[2]The blocking operators have been converted as non-blocking in data stream model for properly computing CQs.

processing operators have input queues and internal windows in order to deal with highly bursty inputs and to convert blocking operators to non-blocking operators.

Both relational or event operators do not have input queues and internal windows, as relational operators perform computations over static relations and event operators perform computations over low input-rate event streams. This is also because of the underlying assumption that query or event processing systems have sufficient processing capacity and resources to handle their data sources.

6.3.4 Computation Model

Computations in event processing models are decomposed into three main components, which correspond to each component of an Event-Condition-Action rule: *1*) computations performed at the tuple level, which are carried out by the event operators, *2*) computations performed at the attribute level, which are carried out by the condition checks, and *3*) computations for processing rules (triggering actions). In some event processing systems, a smaller part of the computations that are carried out at the attribute level are moved to the event detection component (i.e., the mask proposed in [57, 58]), improving the performance. Computations in stream processing models are not clearly partitioned into different components as in the case of event processing. However, considering the functionalities, computations in stream processing models can be viewed as two components: operator computation and window computation. The former involves complicated condition checking and attribute level manipulations, and the latter is required to maintain a snapshot of tuples or status information for blocking operator computations.

Thus, computations that are performed at the tuple level and the computations for rule processing in the event processing models are absent in the stream processing model. On the other hand, window computations in the stream processing model do not appear in the event processing model. Even though both of them operate at the attribute level, operator computation in the stream processing model is more expressive and powerful than the computation performed for condition checking in the event processing model. From the above it is clear that the computations in both models have different emphasis and different purposes, and they are for different applications. Thus, an integration of these two computation models is needed in order to support the applications that need both stream and event processing and such an integrated model is more powerful and useful and can support larger class of applications.

6.3.5 Best-Effort Vs. QoS

The notion of QoS is not present in the event processing literature. Although, there is some work on real-time events and event showers [26], event processing models do not support any specific QoS requirements. Typically, in the event processing model, whenever an event occurs, it is detected or propagated to form a composite event as soon as possible. Thus, events are detected based on the best-effort method. On the other hand, QoS support in a stream processing model is necessary and critical to the success of data stream management systems (DSMSs) for the following reasons: 1) the input of a stream processing system is highly dynamic and unpredictable in contrast to its fixed computation resources. During overload periods, some queries cannot get sufficient resources to compute their results, which can cause unexpectedly long delays for the final output results. 2) many stream-based applications require real-time responses from underlying stream processing system. A delayed response may not be useful, and may even cause serious problems. Different applications can tolerate different response times or inaccuracy in the final query results, and 3) queries with different QoS requirements must be treated differently with a goal to minimize the overall violation of predefined QoS specifications. A number of QoS delivery mechanisms have been explored and proposed [114, 20, 44, 74] in the literature for stream processing systems.

6.3.6 Optimization and Scheduling

Event expressions are represented as event graphs for detection. There has been some work carried out in rewriting event expressions, so that event detection can be made efficient by constructing optimal event detection graphs. Common event sub-expressions are grouped in order to reduce the overall response time and computation effort. In general, event processing does not deal with runtime optimizations. On the other hand, efficient approaches for processing CQs are important to a DSMS. The concept of queues and windows in a DSMS introduce even more challenges and opportunities for query optimization. Optimizations in stream processing model include 1) sharing of queues (inputs) among multiple operators, 2) sharing of windows (synopsis), 3) sharing of operator computations, and 4) sharing of common sub-expressions. The notion of scheduling is also absent from event processing systems. Typically a data-flow architecture (implicitly, a First-In-First-Out scheduling strategy is employed in event processing models) is assumed as indicated earlier and memory usage or event-latency has not been addressed in the literature. On the other hand, optimizing memory capacity, tuple latency, and the combination of these two have prompted many scheduling algorithms [29, 33, 77] in stream processing.

6.3.7 Buffer Manager and Load Shedding

None of the event processing systems assume the presence of queues between event operators. Events were assumed to be processed as soon as they are detected (not necessarily occurred) and partial results are maintained in event nodes. Most of the event processing models assume that the incoming events are not bursty and hence do not provide any kind of buffer management or explicit load shedding strategies. Event consumption modes can be loosely interpreted as load shedding, used from a semantics viewpoint rather than QoS viewpoint. On the other hand, load shedding is extremely important in a stream processing environment. Even with the choice of the best scheduling strategy, it is imperative to have load shedding strategies as the input rates can vary dynamically. Several load shedding strategies, placement of load shedders, and the amount of tuples to be shed (possibly limiting the error in query results) have been proposed [114, 20, 44, 74].

6.3.8 High-Level Rule Processing

ECA Rules describe how the underlying system should respond when an event occurs, making the system reactive. Rules are either used to extend the range of applications that can be supported by the underlying system or to change the way in which new applications are developed. Rules are considered important as they allow users to specify predefined actions that need to be taken when an event occurs and the corresponding conditions are satisfied. Existing event processing systems support dynamic enabling and disabling of rules. On the other hand, rule execution semantics specify how the set of rules should behave in the systems once they have been defined. A rich set of rule execution semantics [121, 35] has been proposed to accurately define and efficiently execute various rules in the literature for event processing models. Those semantics include rule processing granularity, instance/set oriented execution, iterative/recursive execution, conflict resolution, sequential/concurrent execution, coupling modes, and termination. Stream processing systems do not support high-level rule specification and processing, which are critical to many real-world applications.

6.3.9 Summary

Similarities and differences between event and stream processing models have been discussed above. Both models employ a similar data-flow architecture as their computation model over streaming data. However, both models have their limitations for handling applications that require stream processing followed by event processing, and most of the functionalities provided by these two models are complementary to each other. The stream processing model focuses on providing a set of functionalities similar to those provided by DBMSs to process and manage data streams. As a result, a general framework and a set of comprehensive techniques such as the notion of a window, optimization techniques, scheduling strategies, load shedding, and others, have been proposed and are being developed. On the other hand, the event processing models focus on detecting composite events and rule processing under the assumption that event sequences are generated (mostly) within the underlying systems with relatively low input rate. Consequently, scheduling strategies, load shedding techniques, QoS support, and so on were not explored for that model. However, the three component computation model (i.e., event processing, condition checking, and rule processing) developed in the event processing are specialized for event detection and rule processing. In contrast, event computation at tuple level and rule processing are absent in the data stream processing model. Although CQs consisting of modified relational operators and aggregation operators can be used in situation monitoring, its expressiveness and computation capability of complicated events are limited and it is also not well-suited for detecting composite events and applying contexts to reduce the number of meaningful events compared with the techniques provided by the event processing models.

Clearly, it is desirable and natural to combine the strengths of both models into an integrated model with a general framework and a set of comprehensive techniques of stream processing model plus the event computation model (i.e., computation at tuple level, consumption modes, and so on) and sophisticated rule processing capabilities. This integrated model will be much stronger and can serve a larger class of applications than what are currently supported by both the models individually.

6.4 EStreams: An Integrated Model

The proposed integrated model, termed EStreams (for *E*vent and *Stream* processing *s*ystem) is shown in Figure 6.2 and it consists of three stages: 1) CQ processing stage used for computing CQs over data streams, 2) event processing stage that is used for detecting events with/without masks, and 3) rule processing stage that is used to check conditions, and to trigger predefined actions once events are detected.

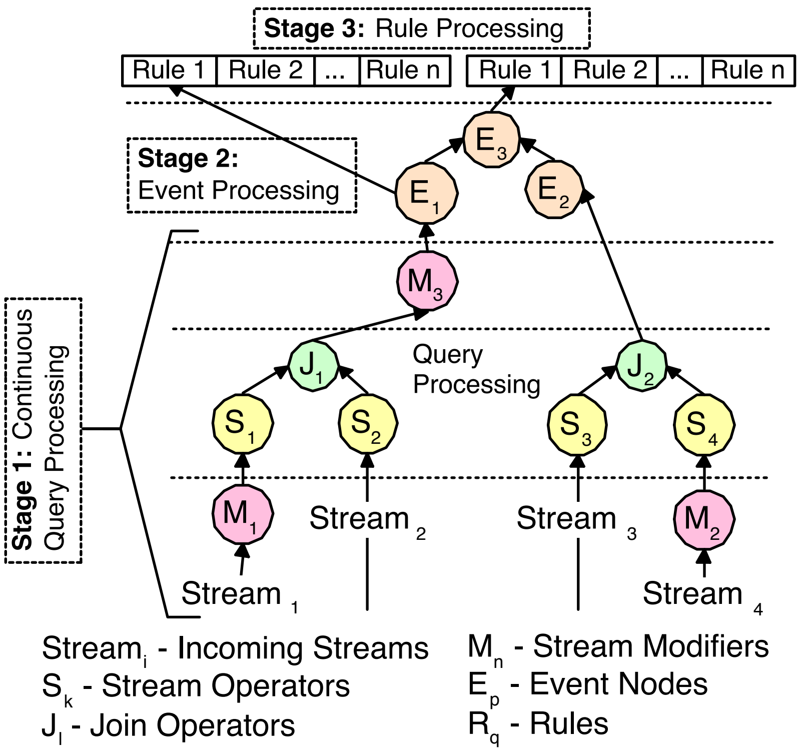

Figure 6.2 EStreams: Three Stage Integration Model.

The seamless nature of our integrated model is due the compatibility of the chosen event processing model (i.e., an event detection graph) with the structure used for stream processing. Based on our analysis, synthesizing both the processing models requires the following issues to be addressed: 1) handling highly bursty event streams (generated by the CQ processing stage) in event processing, 2) processing of events streams based on attributes and not solely on timestamp, 3) specification of events/event expressions, rules and CQs. We have enhanced both the models, as described below, to address the above mentioned issues: 1) Outputs of CQs have to be fed as primitive inputs to event operators in the event processing stage. We have named the continuous queries, so that in the event processing stage the outputs of CQs can be used for detecting events directly. 2) We have introduced stream modifiers that can detect complex changes between tuples in a stream. 3) We have also introduced the semantic window to enhance the expressiveness and computation efficiency of CQs, and to allow creation of more meaningful windows. For the event processing model, 4) We have enhanced the event operators by introducing input queue(s) for each event operator, which makes it possible to handle highly bursty outputs from CQ processing stage and to integrate event operators with stream processing operators seamlessly through the inter-queue and to take advantages of the techniques (i.e., scheduling strategies, load shedding) developed for stream processing model. 5) We have enhanced the event expressions in such a way that primitive events can be the outputs of continuous queries over event streams based on event attributes, and not only on timestamp. 6) We have also enhanced the event consumption modes to support more meaningful windows. Finally, 7) We have extended SQL allowing user to specify events/event expressions, rules and CQs together.

6.4.1 Continuous Query (CQ) Processing Stage

This stage processes normal CQs where it takes streams as inputs, and outputs computed continuous streams. The scheduling algorithms and QoS delivery mechanisms (i.e., load shedding techniques) along with other techniques developed for stream processing model can be applied directly. In many cases, final results of stream computations need to be viewed as events for defining situations that use multiple streams and composite events: 1) A large group of stream applications are interested in not only the normal computations introduced by CQs with Select-Join-Project and aggregation operators, but also in the changes to one or more attributes of a monitored object over time, and 2) current windows based on size (i.e., in terms of time, number of tuples, values of attribute) are somewhat restrictive (refer to Section 6.4.1.2).

To overcome the above shortcomings, we enhance CQs in the following aspects to support more complicated computations required by many stream applications: 1) Ability to name the CQs is needed so that they can be used to define primitive events, and can be used in defining other CQs. 2) We introduce a family of operators, which can be used to compute the changes of one or more attributes over a data stream. These operators are termed as *stream modifiers* in this thesis. 3) Finally, we extend the current *window* concepts to a *semantic window* in order to express and compute more meaningful and complicated windows in an accurate and efficient way. These enhancements not only greatly improve the ability of stream processing model to express more complicated computation requirements through semantic windows and named CQs, but also improve the ability to compute more accurate final results in a more efficient way through stream modifiers and semantic windows. However, none of the enhancements affect the operator semantics, scheduling algorithms, QoS delivery mechanisms, and other components proposed for stream data processing.

6.4.1.1 Named Continuous Queries

Many computations over streaming data are difficult to express and to be computed as a single CQ. In order to express computations more clearly, CQs are named. The name of a CQ is analogous to the name of a table in a DBMS and it has the same scope and usage as that of a table. The queue (buffer) associated with each operator in a CQ supports the output of a named CQ to be fed directly into the input queue of another named CQ. A named CQ is defined by using the following CREATE CQ statement.

```
CREATE CQ CQ_Name AS (Normal CQ statements)
```

However, the FROM clause in a named CQ can use any previously defined CQs through their unique names. The meta information of a named CQ is maintained in a CQ_dictionary in the system. The meta information includes the *query name*, its *input sources*, all *output attributes* ordered by their order in final output tuples, and its *output destination(s)*. If the output destination is to an application, it can be in the form of a named pipe, socket, or an output queue/buffer. A default output destination is defined in the system simply as a sink if there is no destination associated with this CQ. A CQ with a sink as its destination can be disabled in the system until a meaningful destination is associated to it (usually for test purpose). For example, you can define a named CQ and register it with the system, and then register another CQ that refers to this CQ. Once a named CQ is registered to the system, if it refers to any other named CQ, the system will automatically associate it to the destination of its referred CQs. A named CQ can output its final results to multiple destinations.

6.4.1.2 Semantic Window

In current stream processing models, all blocking operators are supported through the concept of a window, which defines a historical snapshot of a finite portion of a stream at any time point. This window defines the *meaningful* set of data used by an operator. Accurate window definition not only impacts the accuracy of final results, but also has a significant impact on system performance. However, the current window types either Size-based (Time-based, Tuple-based) or Attribute-based windows are not very meaningful and they are the only window types that are defined and supported by current stream processing models. A Time-based widow can be simply expressed by *[Range N time units]* and a Tuple-based window can be expressed by *[Row N tuples]*, where N specifies the length of the window. Attribute-based is introduced in [32] and can be expressed by `Size s, Advance i` along with a user-defined or predefined procedure-based function/operator, where `s` is the size of the window in terms of values of an attribute and `i` is an integer or predicate that specifies how to advance the window when it slides. In addition, a partitioned sliding window [14] is defined as $[PartitionByA_1, \cdots, A_kRowsN]$. It logically partitions input stream into different sub-streams based on the equality of attributes $A_1, \cdots, A_k$ and the computations (i.e., aggregation) are performed over each sub-stream. The outputs from all sub-streams are merged into one single output stream.

Although the basic window types[3] are useful for many applications, they are still limited by their inability to express more meaningful windows (i.e., based on semantic information) required by applications. The functionality of a stream operator does not change when it is applied to different applications. On the contrary, its window specification changes as the applications change in order to compute results correctly and efficiently. We call the functionality of an operator as its `global property` and the window property of an operator as its `local property`. Expressing a meaningful and accurate window under different application domains is complicated, and it requires a general-purpose format to express the window concept and a more efficient way to com-

[3]From now, all the above mentioned 4 types of windows are referred as basic windows.

pute the defined window than what is currently available [4]. The window itself can be based on a *computation*, which is used to determine a meaningful snapshot of a portion of an input stream for its respective operator. The **semantic window**, introduced in this chapter, can express more complicated and meaningful windows required by different applications than the basic windows and the computations can be carried out through well-developed and highly optimized SQL query processing engines, which is more efficient.

The main function of a semantic window is to determine which tuples in the current window should be deleted [5] after it adds a new tuple [6] into the current window. Before defining a semantic window, we have to identify the scope of data that it can access to perform computations. Clearly, all tuples in the current window and the new tuple that needs to be added to the current window are fully accessible by that window. Therefore, it is natural to express a semantic window based on the semantic information provided by current window and the new tuple. Each semantic window specifications uses a $\mathcal{CW}$ reference and a $\mathcal{NT}$ reference. Through the $\mathcal{CW}$, a semantic window can access any tuple in the current window. Similarly, all the attributes of the new tuple that needs to be added to the current window can be accessed through $\mathcal{NT}$.

Instead of introducing new operations for a semantic window in current SQL, we claim that a semantic window can be expressed using current SQL statements (i.e.,

[4]The computation required to maintain a window is not discussed in detail in the literature, not mentioning the optimization of window computation.

[5]Since the oldest tuple in a window is the least useful for most applications, in this chapter, we concentrate on windows that delete the oldest tuple (like basic windows). Deletion of other tuples (i.e., every n^{th} tuple) can also be achieved when procedures are used, rather than SQL, for the semantic window.

[6]Similarly, instead of a single tuple, a batch of new tuples can be merged into the window for performance reasons.

SELECT-FROM-WHERE statement) over $\mathcal{CW}$ and $\mathcal{NT}$ with little effort needed for its implementation. We also can take advantage of the well-developed SQL query processing engine and its optimization techniques provided in stream processing model to efficiently compute semantic windows with little effort to modify current data stream systems.

A semantic window defines a finite portion of historical tuples seen so far from a data stream through the condition $\mathcal{SWC}$ (*semantic window condition*), over $\mathcal{CW}$ and $\mathcal{NT}$. Each $\mathcal{NT}$ is appended to $\mathcal{CW}$ by continuously evaluating the $\mathcal{SWC}$ through Algorithm 6.

Algorithm 6: Add New Tuple Algorithm

```
INPUT: ($\mathcal{CW}$, $\mathcal{NT}$);
OUTPUT: Modified $\mathcal{CW}$;
if $\mathcal{CW}$ is empty or not initialized then
    // append the new tuple to current window Appended Window
    $\mathcal{AW} \leftarrow \mathcal{NT} + \mathcal{CW}$;
    return $\mathcal{AW}$ as the new $\mathcal{CW}$;
else
    Appended Window $\mathcal{AW} \leftarrow \mathcal{NT} + \mathcal{CW}$;
    while ((the condition $\mathcal{SWC}$ over the $\mathcal{AW}$ is not TRUE) AND ($\mathcal{AW}$ is
    NOT empty)) do
        Delete the oldest tuple in the $\mathcal{AW}$;
    end
    return $\mathcal{AW}$ as the new $\mathcal{CW}$;
end
```

The $\mathcal{SWC}$ can be any arbitrary condition over $\mathcal{CW}$ and $\mathcal{NT}$. However, to simplify the way to express a $\mathcal{SWC}$, we use the CHECK statement shown below where $\mathcal{NT}$ is considered as a one tuple relation.

```
Stream [      CHECK     logical_Expression
              SELECT    a_1, a_2, ..., a_n
              FROM      𝒞𝒲, 𝒩𝒯
              WHERE     Conditions
              GROUP BY  Attributes
              HAVING    Conditions          ]
```

All the clauses used in the above statement (i.e., SELECT, FROM, WHERE, GROUP BY, and HAVING) have the same semantics and usage as in the standard SQL. However, only $\mathcal{CW}$ and $\mathcal{NT}$ can appear in FROM clause. $a_1, a_2, \ldots, a_n$ are the attribute names (or alias after applying aggregate functions) from either $\mathcal{CW}$ or $\mathcal{NT}$. The CHECK clause is a logical expression which consists of the attribute names used in the SELECT statement, relational and logical operators, and parentheses. The CHECK clause is evaluated and a Boolean value is returned once the SELECT clause is evaluated. The CHECK clause requires only one row from the FROM clause after applying other clauses. If more than one row is returned, only the oldest one in current window is used to evaluate the CHECK clause.

Based on the definition of $\mathcal{SWC}$ and named CQs we provide few examples to demonstrate the use of semantic window to support some of the basic windows as well as other more meaningful ones.

Example 2 (Similar with $\sqsubset$ *Row* $50,000$ $\sqsupset$). *The average speed of each car over a 50,000-tuple sliding window on stream CarLocStr can be computed using the following CQ with a regular row window and a semantic window respectively. (Note: the $\mathcal{SWC}$ is evaluated after we append the $\mathcal{NT}$ to $\mathcal{CW}$).*

```
   CREATE   CQ AvgSpeedTuple AS
   SELECT   car_id, AVG (speed)
     FROM   CarLocStr [Row 50,000]
 GROUP BY   car_id
```

```
   CREATE   CQ AvgSpeedTuple AS
   SELECT   car_id, AVG(speed)
     FROM   CarLocStr [ CHECK CWCount <= 50,000
                        SELECT COUNT(*) AS CWCount FROM 𝒞𝒲 ]
 GROUP BY   car_id
```

It is worth noting that windows based on the number of tuples, irrespective of the kind of definition (i.e., Range or semantic) used, are not commutative [14]. From the above example we had clearly shown that semantic windows can be used to express and compute the common Tuple-based windows. Similarly we can represent other types of windows. However, a semantic window can be used to express and compute more meaningful windows, as discussed below.

Example 3 (Disjoint Window). *Compute the moving average speed of the car with car_id 100 within each segment.*

```
 CREATE   CQ AvgSpeedOfCar100_Wrong AS
 SELECT   AVG(speed)
   FROM   CarSegStr [ CHECK CWSeg = NTSeg
                      SELECT 𝒞𝒲.seg AS CWSeg, 𝒩𝒯.seg AS NTSeg
                        FROM 𝒞𝒲, 𝒩𝒯
                      WHERE (𝒞𝒲.timestamp =
                            (SELECT MIN(𝒞𝒲.timestamp) FROM 𝒞𝒲)) ]
  WHERE   𝒞𝒲.car_id = 100
```

The above CQ AvgSpeedOfCar100_Wrong does not compute the average speed of the car with car_id 100 since the window is computed before the outer WHERE clause is evaluated. The window will change whenever a report from a car from different segment is received. Instead of specifying all the computations in one CQ, we can take advantage of the named CQs proposed in this chapter and use the following two CQs to accurately and efficiently compute the Example 3. Similarly, we can also compute Example 3 in one CQ by using the semantic window with GROUP BY, which will be discussed later. However, it is not efficient since a sub-window should be maintained for each car.

```
CREATE  CQ FilterForCar100 AS
SELECT  *
  FROM  CarSegStr
 WHERE  𝒞𝒲.car_id = 100
CREATE  CQ AvgSpeedOfCar100 AS
SELECT  AVG(speed)
  FROM  FilterForCar100 [ CHECK CWSeg = NTSeg
                          SELECT 𝒞𝒲.seg AS CWSeg, 𝒩𝒯.seg AS NTSeg
                            FROM 𝒞𝒲,𝒩𝒯
                          WHERE (𝒞𝒲.timestamp =
                            (SELECT MIN(𝒞𝒲.timestamp) FROM 𝒞𝒲)) ]
```

Example 4. *Consider a network traffic stream NTStr consisting of timestamp, pkg_id, pkg_size, source_IP, dest_IP, source_port, and dest_port. We want to compute the number of flows determined by the unique combination of (source_IP, dest_IP) over a window in which the total number of bytes transferred does not exceed 100 MB and the number of time units spanning the window does not exceed 10 seconds.(Note: Below "+" indicates concatenation)*

```
CREATE  CQ NumberOfFlows AS
SELECT  COUNT(distinct source_IP + dest_IP)
  FROM  NTStr [     CHECK ((CW_S <= 100M) AND
                          (NT_TS - CW_TS < 10Sec))
                  SELECT SUM (CW.pkg_size) AS CW_S,
                         MIN(CW.timestamp) AS CW_TS,
                         MIN(NT.timestamp) AS NT_TS
                    FROM CW, NT
                GROUP BY NT.timestamp ]
```

As $\mathcal{NT}$ is a one tuple relation, the GROUP BY clause in $\mathcal{SWC}$ in the above example outputs only one tuple. Otherwise, GROUP BY will partition the window into multiple sub-windows, which will be discussed below.

Before we discuss the GROUP BY clause used in $\mathcal{SWC}$, we will give an example to show that GROUP BY is necessary and cannot be replaced by moving it to the CQ.

Example 5. *Compute the moving average speed of each car on its current segment. Two CQs are provided, the first one gives what is required, and the second one does different computation.*

```
CREATE  CQ CarMovingAvgSpeed AS
SELECT  AVG (speed), car_id
  FROM  CarSegStr [    CHECK CWSeg = NTSeg
                    SELECT CW.seg AS CWSeg, NT.seg AS NTSeg
                      FROM CW, NT
                     WHERE (CW.timestamp =
                             (SELECT MIN(CW.timestamp) FROM CW))
                  GROUP BY CW.car_id ]
```

In the above CQ, we partition the input stream into multiple sub-streams based on car_id, apply the window computation (i.e., a window formed by the tuples from the same segment for a particular car) to form a correct window, and then compute the moving average speed of the car based on the tuples in its window. Finally, the outputs from all sub-streams are merged into an output stream. As shown below, if the GROUP BY is moved down to the CQ, the query just forms one semantic window from the input stream, applies the GROUP BY and finally output the results.

```
   CREATE  CQ CarMovingAvgSpeed AS
   SELECT  AVG (speed), car_id
     FROM  CarSegStr [  CHECK CWSeg = NTSeg
                        SELECT 𝒞𝒲.seg AS CWSeg, 𝒩𝒯.seg AS NTSeg
                          FROM 𝒞𝒲, 𝒩𝒯
                         WHERE (𝒞𝒲.timestamp =
                               (SELECT MIN(𝒞𝒲.timestamp) FROM 𝒞𝒲)) ]
 GROUP BY  car_id
```

Clearly, the window formed by the latter case (i.e., GROUP BY statement is part of the CQ) is based on the input tuple without considering the car_id. If two consecutive tuples are from two different cars that are on the different segments, the query always outputs the current speed of each car as the window is a one tuple window. But, if the two consecutive tuples are from two different cars that are on the same segment, then the query computes the average speed of these two cars, rather than the average speed of a particular car. Therefore, the GROUP BY in $\mathcal{SWC}$ is necessary.

The semantic window expressed using a GROUP BY is similar to the partitioned window type introduced in [14]. However, the condition that maintains each logical sub-window is more meaningful and powerful than a simple condition on the number of rows. When a GROUP BY clause is used in the $\mathcal{SWC}$ definition, a semantic window is logically split into a number of sub-windows based on the attributes in the GROUP BY clause, and the $\mathcal{SWC}$ is applied to each sub-window. Each logical sub-window is labeled by the value(s) of GROUP BY attributes. The new tuple is added only to the logical sub-window whose label matches the corresponding values of GROUP BY attributes in the new tuple. Since a new tuple is only added to one logical sub-window, only that logical sub-window is evaluated (actually, evaluation results for all the other sub-windows are always TRUE because they are not changed). When a stream operator computes over its semantic window, the computation is only on the sub-windows that have been changed (after adding the new tuple). If the computation is applied to other sub-windows, duplicated results will be generated.

Example 6. *Considering the network traffic stream NTStr in Example 4, we want to monitor the number of sessions of each host connected to the SIGMOD web server, which has its IP addresses as 199.222.69.250 and 199.222.69.251, over a sliding window of last 10,000 packets of each host or 5 minutes (300 seconds).*

```
CREATE  CQ SIGMODTraffic AS
SELECT  *
  FROM  NTStr
 WHERE  dest_IP = 199.222.69.250 OR dest_IP = 199.222.69.251

CREATE  CQ NumberOfSessions AS
SELECT  source_IP, count(distinct source_port) as sessions
  FROM  SIGMODTraffic [    CHECK ((NPack <= 10,000) OR
                                  (( NT_TS - CW_TS) <= 300Sec))
                          SELECT count(*) AS NPack,
                                  min(CW.timestamp) AS CW_TS,
                                  min(NT.timestamp) AS NT_TS
                            FROM CW, NT
                        GROUP BY CW.source_IP ]
```

The semantic window concept introduced above greatly enhances the expressiveness power of CQs as it allows accurate and meaningful ways of expressing a window for stream-based applications, and it also provides an efficient way of computing semantic windows through highly optimized SQL engines. Further, the adaptation of SQL for expressing semantic windows makes it even more useful as it avoids introducing new language constructs and defining its semantics and can be easily integrated into current stream processing model.

The computation introduced by maintaining a semantic window is necessary for window-based operators and its computation overhead is less than those introduced by basic window types. This is due to the fact that 1) accurate definition of a window reduces the computation overhead required to compute the operator as the number of tuples decreases in its window, 2) well-developed optimization techniques and the well-optimized SQL engine can be used to process SQL-based semantic window efficiently. More important, 3) more meaningful and accurate window definition and computation can be supported only by our semantic window and not by current window definitions and computations. Many critical stream-based applications need accurate results that can only be obtained through accurate definition of windows, which cannot be achieved through basic window types.

Therefore, the semantic window introduced in this chapter is necessary, critical, and efficient for data stream processing model and our integration model. The detailed computation overhead and optimizations of semantic windows are an interesting problem and will be part of our future work.

6.4.1.3 Stream Modifiers

The `stream modifiers` are introduced for CQs in order to extend the computation of current stream processing to capture the changes of interest in an input data stream. Before we introduce the detailed semantics of a stream modifier, we define the state in an input stream as follows:

Definition 1 (State). *A state in a data stream is defined as a tuple in that stream. An n-state in a stream is denoted by $s =< v_1, v_2, \cdots, v_n >$, where v_i is the value corresponding to attribute A_i defined in the stream schema.*

Definition 2 (Stream Modifier). *A stream modifier is defined as a function to compute the changes (i.e., relative change of an attribute) between two consecutive states of its input stream. A stream modifier is denoted by $M(< s_1, s_2, \cdots, s_i > [, P < pseudo >][, O|N < v_1, v_2, \cdots, v_j >])$, where M is called the modifier function that computes a particular kind of change. The i-tuple $< s_1, s_2, \cdots, s_i >$ is the parameter required by the modifier function M. The following $P < pseudo >$ defines a pseudo value for the*

M function in order to prevent underflow. The following j-tuple element is called the untouched attribute that needs to be output without any change. The O|N part is called modifier profile, which determines whether the oldest values or the latest values of the j-tuple that needs to be output. If O is specified, the oldest values are output or the latest values are output if N is specified. Both untouched attributes and modifier profile are optional.

A family of stream modifiers could be defined using the above definitions. Currently, we have implemented the following three commonly used stream modifiers in our system. In the following definitions, x^i and x^{i+1} are the values of attribute x from state i and $i+1$ respectively, and anything in between "[ ]" is optional.

ADiff() is used to detect absolute changes over two consecutive states. It returns absolute change of the values of attribute s_1, and the values of a subset of attributes given in $O|N <>$ profile. It is formally defined for case O as follows:

$$ADiff(< s_1 > [, N < v_1, v_2, \cdots, v_j >]) =< \frac{s_1^{i+1} - s_1^i}{s_1^i}[, v_1^{i+1}, v_2^{i+1}, \cdots, v_j^{i+1}] >$$
$$ADiff(< s_1 > [, O < v_1, v_2, \cdots, v_j >]) =< \frac{s_1^{i+1} - s_1^i}{s_1^i}[, v_1^{i}, v_2^{i}, \cdots, v_j^{i}] >$$

RDiff() is used to detect the relative changes over two consecutive states. It returns relative change of the values of attribute s_1, s_2, and the values of a subset of attributes given in $O|N <>$ profile. It is formally defined for case N as follows:

$$RDiff(< s_1 >, P < pseudo > [, N < v_1, v_2, \cdots, v_j >])$$
$$=< \frac{s_1^{i+1} - s_1^i + pseudo}{s_1^i + pseudo}[, v_1^{i+1}, v_2^{i+1}, \cdots, v_j^{i+1}] >$$
$$RDiff(< s_1 > [, P < pseudo >][, O < v_1, v_2, \cdots, v_j >])$$
$$=< \frac{s_1^{i+1} - s_1^i + pseudo}{s_1^i + pseudo}[, v_1^{i}, v_2^{i}, \cdots, v_j^{i}] >$$

ASlope() is used to compute the slope ratio of two attributes over two consecutive states. It returns the slope ratio of the values of attributes s_1, s_2, and the values of a subset of attributes given in $O|N <>$ profile. It is formally defined for case O as follows:

$$ASlope(< s_1, s_2 >, P < pseudo > [, N < v_1, \cdots, v_j >])$$
$$=< \frac{s_1^{i+1} - s_1^i + pseudo}{s_2^{i+1} - s_2^i + pseudo}[, v_1^{i+1}, v_2^{i+1}, \cdots, v_j^{i+1}] >$$
$$ASlope(< s_1, s_2 > [, P < pseudo >][, O < v_1, \cdots, v_j >])$$
$$=< \frac{s_1^{i+1} - s_1^i + pseudo}{s_2^{i+1} - s_2^i + pseudo}[, v_1^i, v_2^i, \cdots, v_j^i] >$$

A stream modifier can only be used in a SELECT clause and is shown in the example in Section 6.4.1.4.

Similarly, any aggregation operator can be used inside a stream modifier. The output of an aggregation operator is considered a normal attribute in a stream modifier. For example we can use $ADiff(< AVG(speed) >, N < car\_id, location, timestamp >)$ in an SQL statement.

Since we already have a `window` concept in current CQs, we can further extend the computation of a stream modifier within a window. When a window is specified, a stream modifier can be used to compute the changes between the oldest tuple and the latest tuple, instead of the two consecutive tuples.

6.4.1.4 CQ for CAR ADN Example

The following `IMMOBILE` and `DECREASE` queries can be used to find all cars that stay at the same location and the cars whose speed has decreased by 30% within the last 2 minutes using the extensions described so far.

```
CREATE  CQ DECREASE AS
SELECT  RDiff(<speed> as C_speed, p<0.01>,
             N<car_id, location, timestamp>)
  FROM  CarLocStr [   CHECK CWtime - NTtime <= 2
                     SELECT MIN(𝒞𝒲.timestamp) AS CWtime,
                            MIN(𝒩𝒯.timestamp) AS NTtime
                      FROM 𝒞𝒲, 𝒩𝒯
                   GROUP BY 𝒞𝒲.car_id ]
 WHERE  C_speed <= -30%

CREATE  CQ IMMOBILE AS
SELECT  RDiff(<speed> as C_speed, p <0.01>,
             N<car_id, location, timestamp>)
  FROM  CarLocStr [   CHECK CWtime - NTtime <= 2
                     SELECT MIN(𝒞𝒲.timestamp) AS CWtime,
                            MIN(𝒩𝒯.timestamp) AS NTtime
                      FROM 𝒞𝒲, 𝒩𝒯
                   GROUP BY 𝒞𝒲.car_id ]
 WHERE  C_speed = 0.0
```

6.4.2 Event Processing

In this section we will discuss the enhanced event expression computations, event specification using the extended SQL, and event node inputs and outputs.

Enhanced Event Operators and Event Expressions: EDGs in the current event processing systems do not have input queues/buffers for event operators as the input rate of an event stream is not assumed to be very high and highly bursty and events are processed from bottom to up based on their arrival timestamp (similar to a tuple-based FIFO scheduling strategy). Thus, in our integrated model, input queues/buffers are added to event operator nodes to handle the highly bursty input generated by the CQs from the CQ processing stage. In a traditional event processing system, primitive events can be of either class level or instance level, but both are based on timestamps. Instance level events play an important role for events generated by stream processing, but with the dynamic nature of incoming streams it is difficult or impossible to determine the instance level events ahead of time. Example discussed below reveals the limitations of the current event operators that operate based on timestamp.

Let us take the Car ADN example that has three requirements. Let us assume that event *Eimmobile* represents IMMOBILITY (requirement 1) and event *Edecrease* represents SPEED REDUCTION (requirement 2). An accident is represented as event *Eaccident* and is detected when an event *Eimmobile* happens before (i.e., followed by) event *Edecrease*. In addition, both the cars that are detected should be in the same segment for the event *Eaccident* to happen. Let us assume that stream *CarSegStr* (refer to Section 6.1) sends inputs to the named continuous queries CQ1 and CQ2. CQ1 checks the car for immobility and CQ2 checks for speed reduction. CQ1 inputs to the event node *Eimmobile* and CQ2 inputs to event *Edecrease* with the following formats

CQ1: (timestamp, car_id, speed, exp_way, lane, dir, segment_id)

CQ2: (timestamp, car_id, speed, exp_way, lane, dir, segment_id, decrease_in_speed)

Let us assume that *Eimmobile* occurs at 10.00 a.m. with tuple `(10.00 a.m, 1, 0 mph, EW123, 3, NW, 104)`, and event *Edecrease* occurs at 10.03 a.m. and 10.04 a.m. with tuples
`(10.03 a.m, 2, 40mph, EW123, 1, NW, 109, 45%)`, `(10.04 a.m, 5, 20mph, EW123, 4, NW, 104, 40%)`.

Two events are said to be in sequence if the first event proceeds the second event either in time or a monotonically increasing number. Thus, tuples with `car_id 1` (i.e., *Eimmobile*) and `car_id 2` (i.e., *Edecrease*) trigger the event *Eaccident*. Similarly tuples with `car_id 1` and `car_id 3` triggers the event *Eaccident*. From above it is evident that in current event processing systems (with or without event masks), the important condition that both the cars should be from the same segment is checked only after the event *Eaccident* is detected. This introduces a high overhead on the event computation as there can be many unnecessary detection of event *Eaccident*. The above example can be modeled using instance level events, but all the instances of a class should be predefined (or known previously). This may be impossible in a system where the data streams' attribute values are dynamic. In addition, they require lot of event nodes, and introduce a high overhead for computation. Hence, event detection computation has to be enhanced to support efficient detection of events. In our integrated model this is achieved by enhancing event expression computation by pushing the event masks into the event operator node, so that attribute conditions are checked before the events are detected.

Inputs to Event Processing Stage: CQs output data streams in the form of tuples. These tuples are fed as input event streams to the event processing stage. We assume that each tuple in a data stream that enters the system is time stamped or has an ordering attribute (i.e., has a monotonically increasing sequence attribute) and can be used in the event processing stage. With enhanced event expression computations any attribute of an event (tuple) can be used in the event processing stage for 1) checking conditions, 2) masking the inputs to the event nodes, and 3) merging event streams.

Event Specification using Extended SQL: CQ processing is compatible with the event graph processing approach used by the event processing model. Thus, by suitably extending queries over event streams and processing them using a push model, users can monitor diverse event stream combinations in a timely and meaningful manner. Users can specify events based on CQs using the CREATE EVENT statement shown below:

```
CREATE EVENT  $\mathcal{E}_{name}$
      SELECT  $\mathcal{A}_1$, $\mathcal{A}_2$, ..., $\mathcal{A}_n$
        MASK  $\mathcal{AC}_1$, $\mathcal{AC}_2$, ..., $\mathcal{AC}_n$
        FROM  $\mathcal{E}_S$ | $\mathcal{E}_X$
```

CREATE EVENT creates a named event $\mathcal{E}_{name}$, SELECT selects the attributes $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_n$ from either the event stream $\mathcal{E}_S$ or the event expression $\mathcal{E}_X$, MASK applies conditions on the attributes $\mathcal{AC}_1, \mathcal{AC}_2, \ldots, \mathcal{AC}_n$ of those events that enter the event operator node in the event detection graph. $\mathcal{E}_S$ is a named CQ or a CREATE CQ statement and $\mathcal{E}_X$is an event expression that combines more than one event using event operators, and event attributes. Below shown is the CREATE EVENT statement for a primitive event *Eprim* that selects all the cars that have segment id "<15" from the event stream $E1$ produced by *CarLocStr*. As shown, MASK applies attribute condition $\mathcal{AC}_1$ (i.e., seg_id<15), and SELECT selects the car_id and seg_id FROM $E1$.

```
CREATE EVENT  prim
      SELECT  stream.car_id, E1.seg_id
        MASK  stream.seg_id < 15
        FROM  (CREATE CQ E1 as ...);
```

Creation of Events for CAR ADN Example: Section 6.4.1.4 provides the CQs for the CAR ADN example. *Eimmobile* is the event created from the CQ *IMMOBILE* and *Edecrease* is the event created from the CQ *DECREASE*. Event *Eaccident* represents an accident and is detected when an event *Eimmobile* happens before *Edecrease*, and both the seg_ids are same. Event expression $\mathcal{E}_X$ for the accident is *Eaccident* =

Eimmobile SEQUENCE *Edecrease* where SEQUENCE is an event operator. CREATE EVENT for the same is shown below:

```
CREATE EVENT  Eaccident
      SELECT  Eimmobile.car_id as EIcar_id, Edecrease.car_id as EDcar_id,
              Eimmobile.seg_id, Eimmobile.timestamp
        MASK  Eimmobile.seg_id = Edecrease.seg_id
        FROM  (CREATE EVENT Eimmobile
                        SELECT IMMOBILE.car_id, IMMOBILE.seg_id
                          FROM IMMOBILE)
              SEQUENCE
              (CREATE EVENT Edecrease
                        SELECT DECREASE car_id, DECREASE.seg_id
                          FROM DECREASE)
```

Event nodes are created in the EDGs based on the CREATE EVENT specifications. Output from the CQ is fed as inputs to the event nodes in the EDGs as event streams. In the integrated model, input to the event nodes can be created in many ways (i.e., for the FROM clause) and they are: (*1*) from the CQ, (*2*) from the underlying system, and (*3*) from an external source. Once *Eaccident* is detected, it is propagated to the rule processing stage.

6.4.3 Rule Processing

The rule system is responsible for triggering predefined actions. A rule is used to trigger predefined actions once its associated event is detected. In our integrated model rules can be specified and created using the CREATE RULE statement as shown below.

```
CREATE RULE  $\mathcal{R}_{name}$ [, $\mathcal{CM}$, $\mathcal{CT}$, $\mathcal{P}$]
         ON  $\mathcal{E}_{name}$
R_CONDITION  Begin; (Simple or Complex Condition); End;
   R_ACTION  Begin; (Simple or Complex Action); End;
```

As shown above, CREATE RULE creates the rule $\mathcal{R}_{name}$ along with its properties such as coupling mode $\mathcal{CM}$ (e.g., immediate, deferred), consumption mode or context $\mathcal{CT}$ (e.g., recent, continuous) and priority $\mathcal{P}$ (e.g., 1, 2 where 1 is the highest) a positive integer used to set rule priority. ON specifies the event $\mathcal{E}_{name}$ associated with the rule and it can be replaced by the CREATE EVENT statement. In addition, a rule also contains conditions associated with the rule and actions to be performed when conditions results are true. Conditions on attributes act as event mask. Other conditions that are pertinent to the rule, and those that are complex (i.e., any arbitrary condition such as average, standard deviations, PL/SQL code etc.,) are specified in the rule condition.

Enhanced Event Consumption Modes: Analogous to the event operator semantics, current event consumption modes are also based on time and have similar drawbacks. In order to make the consumption modes more meaningful and consistent with stream processing, we introduce semantic windows and attribute based event consumption. For example, the recent mode from [36] can be viewed as a single tuple window and is used by applications where the events are happening at a fast rate and multiple occurrences of the same type of event only refine the previous data value.

We will explain the limitation of the current consumption modes based on the recent consumption mode. Let us assume that event *Eimmobile* occurs at 10.00 a.m. with tuple `(car_id 1, Time 10.00 a.m, seg_id 1, Speed 0)`. It is propagated to the SEQUENCE node and it waits in the composite event node for the event *Edecrease* to occur, in order to detect *Eaccident*. When the next instance of event *Eimmobile* occurs at 10.01 a.m. with the tuple `(car_id 4, Time 10.01 a.m, seg_id 3, Speed 0)`, it replaces the previous instance with "car_id 1" even though both have different car_ids. This is because recent mode is based on a single tuple window, and the computation is based only on the timestamp where the instance with a latest timestamp replaces the previous instance. In order to detect the event occurrences without losing potential events, events can be replaced based on some attributes forming a set or partition. Thus, in the above example only event instance of "car_id 1" that occurs at a later point in time can replace "car_id 1" that occurred at 10.00 a.m., and not "car_id 4" that had occurred at 10.01 a.m.

Currently, there is no notion of windows and all the events are kept in the event node until a new instance occurs or until it is consumed. As the input rate is likely to be bursty in the integrated EStreams model (as it is fed by streams), there is a need for associating a window with each event node. Thus, by introducing windows along with the event consumption modes, the event processing system will be able to handle bursty event streams.

Event consumption modes are specified as an option along with the CREATE RULE statement. In our integrated model event consumption modes are enhanced to support attribute and window based computations. As mentioned previously, events can be unary, binary or ternary. Thus, specifying consumption modes in event expressions requires the attributes/windows for all the operators. Enhanced event consumption modes are specified as shown below, where $\mathcal{CT}$ represents the context, L, M and R represents the left, middle, and right events. Attributes (A), number of events (E), and time units (T) are optional. When an operator is binary only L and R are specified. Similarly A is specified when attribute based partition is required and E/T is specified when the events (tuples) have to be maintained in an event node at a point in time.

$\mathcal{CT}$ `[ L[A, E/T], M[A, E/T], R[A, E/T] ]`

Let us create a rule *Rsample* for an event *Esample*. This event selects the cars with id ">1000" from the named CQ *Estream*. In addition we want to partition the window based on the attribute car_id. This rule should have immediate as $\mathcal{CM}$, recent as $\mathcal{CT}$ (including the attribute as a left event as *Esample* is a primitive event) and with highest priority $\mathcal{P}$ (i.e., 1). There are no conditions/actions associated. CREATE RULE statement for this rule is shown below:

```
CREATE RULE   Rsample, IMMEDIATE, RECENT L[Esample.car_id], 1
         ON   (CREATE EVENT Esample
                          SELECT Estream.car_id, Estream.timestamp
                            MASK Estream.car_id > 1000
                            FROM (CREATE CQ as Estream ...))
```

Rule Creation for CAR ADN Example: When an event corresponding to an accident *Eaccident* is detected, various types of life saving actions are required to be performed. Creation of event *Eaccident* is shown in Section 6.4.2. Rule corresponding to the CAR ADN example is shown below.

```
   CREATE RULE   AccidentNotify, IMMEDIATE,
                 RECENT L[Eimmobile.car_id], R[Edecrease.car_id], 1
            ON   EVENT Eaccident
   R_CONDITION   Begin; (true); End;
      R_ACTION   Begin;
                   //INDICATE POLICE CONTROL ROOM
                 PCR(Eaccident.seg_id, Eaccident.EIcar_id,
                         Eaccident.EDcar_id, Eaccident.timestamp);
                   //INDICATE AMBULANCE CONTROL ROOM
                 ACR(Eaccident.seg_id, Eaccident.timestamp
                         Eaccident.EIcar_id, Eaccident.EDcar_id);
                   //INDICATE UPSTREAM CARS
                 UpSSeg(Eaccident.seg_id, Eaccident.timestamp);
                   //INDICATE TOLL STATION
                 TollSt(Eaccident.seg_id, Eaccident.timestamp);
                 End;
```

6.5 Summary

In this chapter, we first discussed the necessity and importance of event and rule processing under the context of data stream processing and then analyzed the similarities and differences between the stream processing model and the event processing model developed separately. Based on the sophisticated requirements of advanced stream applications and our analysis, we have proposed EStream, an integrated model that combine their strengths through the techniques and enhancements proposed in this chapter. Those techniques and enhancements include:

1. **named CQs** to support association of primitive events with continuous queries;
2. **stream modifiers** to detect more complicated changes in a stream;
3. **semantic windows** to enhance the expressiveness and computation efficiency of CQs and to allow creation of more meaningful windows apart from the basic windows;
4. **extension of event operators** to handle highly bursty inputs from stream processing by introducing input queues/buffers;
5. **extended event expressions** in such a way that primitive events can be the outputs of continuous queries over event streams based on event attributes;
6. **extended event consumption modes** through semantic windows and attribute based event consumption so that they can be more meaningful and consistent with stream processing;
7. we have extended SQL to support combined specification of events and CQs.

By using the above techniques and enhancements, our integrated model not only supports a larger class of applications, but also provides more accurate and efficient ways for processing CQs and event expressions. All the enhancements proposed in this chapter do not affect any current stream processing technique and can be easily integrated into any current data stream management systems. Finally, a prototype of the proposed integrated model is underway.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

In this thesis, we discussed problems and issues related to QoS support in a general DSMS. Those problems include system capacity planning, QoS parameter verification, run-time resource allocation, load shedding, and event and rule processing. We developed comprehensive solutions and algorithms for these problems. Although each component developed can be used individually to assist QoS support in DSMSs, the more important contribution of this thesis is that it presented algorithms and solutions for various phases of QoS support in DSMSs, starting from system capacity planning to a set of comprehensive QoS deliver mechanisms after systems are deployed and to verify and conforming QoS requirements. These algorithms and solutions form a framework for a DSMS to manage, control, deliver, and verify QoS requirements in a general DSMS. There is no doubt that lot more work can be done for the framework and the problems discussed in this thesis. In the rest of the chapter, we summarize each component of the framework and provide some further work.

7.1 Modeling Continuous Queries

The continuous query modeling work forms the basis of our QoS framework for DSMSs. It provides a closed-form solution to estimate various QoS parameters through our queueing model. The queueing model provides a formal way for system capacity planning before deploying a DSMS for a specific application. On the other hand, the estimated QoS parameters provide sufficient quantitative information to integrate various run-time QoS management mechanisms, such as run-time scheduling strategies, load shedding, admission control, and others, into our framework. Those quantitative information makes it possible for DSMSs to dynamically choose suitable QoS deliver mechanisms, such as switching from a memory-favorable scheduling strategy to a tuple-latency favorable scheduling strategy, dynamically activating and deactivating load shedding, and admission control. Additionally, the estimated QoS parameters provide an effective way to verify whether defined QoS requirements of a continuous query have been indeed satisfied.

There are at least two ways to further extend our modeling work to a more general system.

1. In this thesis, all our analysis work are based on Poisson input data streams. Although the results of our analysis based on this class of input streams are sufficient for most applications, it would be more accurate and general if we can extend our analysis to more general data streams, such as self-similar data streams with Long-Range Dependence (LRD) process. It would also be interesting to study how bursty behavior in input data streams impacts the performance of different query plans of a query and the performance of a DSMS.
2. Our queueing model is a one-server queueing model. However, in a multiple-processor architecture, there are multiple servers to serve the input tuples. Extending our analysis to a multiple-server queueing model would be more useful and can support larger group of applications.

7.2 Scheduling Strategies

The family of scheduling strategies proposed in this thesis satisfies requirements of a DSMS for many stream-based applications that have different QoS requirements. The PC strategy is developed for those applications that consider tuple latency as the most critical requirement. The MOS strategy is useful for those applications that are in favor of minimization of the total memory requirement. The greedy segment strategy and the simplified segment strategy provide trades-off between the tuple latency and the total memory requirement for those applications that are sensitive to both tuple latency and the total memory requirement. More practically, the threshold strategy – the dynamical hybrid of the PC and MOS strategy, makes it more practical for a large group of stream-based applications since it inheritances the advantages of both strategies.

A list of open problems and further work in scheduling strategies include:

1. All scheduling strategies proposed in this thesis and in the literature so far have not fully taken QoS requirements into consideration. One possible reason for that is the high-overhead introduced by scheduling itself since a QoS-aware scheduling strategy has to do scheduling at the granularity of tuple if we need each output tuple to satisfy defined QoS requirements. However, the cost of processing one tuple is low as compared with the cost of scheduling it. To schedule a batch of tuples during each run, the tuple latency requirement and the bursty input mode make it difficult to optimally determine the size of a batch.
2. In this thesis, we developed an event-driven preemptive scheduling model. It would be useful to develop different scheduling models such as a non-preemptive one, CPU-limited preemptive scheduling model, and others, and to compare and contrast those models for developing various run-time scheduling strategies.
3. Further study of the size of a batch (either in terms of CPU time or the number of tuples) for scheduling is important and necessary to decrease the overhead introduced by scheduling itself and to be adaptive to different scenarios (i.e., different input modes, different applications).
4. Operator internal scheduling has not been studied in the literature so far for binary or multiple-way operators. As we have shown in Section 4.1.1, different operator-internal-scheduling strategies have different impact on the total memory require-

ment of a DSMS. Choosing the right operator-internal-scheduling strategy can further decrease the total memory requirement of a DSMS.

7.3 Load Shedding

The set of comprehensive load shedding techniques makes sure that a query processing system has sufficient resources to compute registered queries in the system without violating their QoS requirements. The technique we have developed for estimating system load is a closed-form solution and can be efficiently computed based on current input rates and characteristics of continuous queries. The system load estimation component does not depend on the load shedding component or other components in our QoS framework and can be integrated into other load shedding mechanisms. The algorithms for load shedding and load distribution are optimal ones and are generally applicable for DSMSs. Some open problems for further work in load shedding include:

1. High-overhead problem in load shedding. A shedder, either a random or semantic one, introduces at least as much cost as a predict operator. In a DSMS with hundreds of continuous queries, it has to introduce hundreds or even thousands of shedders. The total cost of shedders can be high. The load shedding mechanisms based on dropping tuples through shedder have their limitations in practical DSMSs. It is necessary and important to exploit other load shedding mechanisms (see below).
2. Approximation algorithms. As we discussed above, it is more appropriate to develop approximation algorithms for expensive operators such as two or multiple-way join operators. There has been some initial work in the literature; however, further development in this direction is necessary to make it practically useful.
3. Anytime (contract) algorithms [127, 83, 68]. Anytime algorithms are extensively studied in real-time systems and artificial intelligence systems. An anytime algo-

rithm is an algorithm that can be interrupted at any time and still produces results of a certain quality and the quality of results of an anytime algorithm is a function of allocated computation time. Therefore, anytime algorithms are ideal choices for load shedding in DSMSs if each operator (at least each of expensive operators) can be implemented as an anytime algorithm. Once the system is short of resources, it can limit the computation time allocated for each operator that is implemented as anytime algorithm and the algorithms internally determine the optimal results based on allocated computation time. However, how to implement an operator as an efficient anytime algorithm at this point is unclear.

4. Admission control. Rather than dropping tuples, it is more efficient and effective to temporarily suspend some continuous queries in the system to shed load.

7.4 Event and Rule Processing

The EStream, an integrated model of data stream processing and event and rule processing, has strengths of both models and is better than the sum of its parts. The integrated model can satisfy a larger group of data stream applications that need to not only perform efficient continuous query processing but also complicated event and rule processing. The enhancements proposed for the integrated model improve upon both computation and expressive power of DSMSs. Especially, the concept of semantic window greatly enhances the computation and expressive power of continuous queries from different application domains.

A list of open problems for further work in event and rule processing are:

1. Rule processing in the context of data stream processing needs further investigation. In this thesis, we mainly focus on the complex event processing rather than event processing.

2. Batch processing for semantic window. In this thesis, the semantic window is maintained through inserting one tuple each time. To further decrease overhead, it would be more efficient to admit a batch of tuples, rather than one individual tuple.
3. Load shedding mechanism through shrinking the window. It is more useful for many applications to discard the older tuples than newly arrived tuples. Current load shedding mechanisms discard tuples from incoming data streams or partially processing tuples in the input queues of an operator. Since most of operators do their computation over a window of tuples that are a snapshot of most recent inputs, it is natural and useful to drop the oldest tuple or least used tuple in the windows of an operator, rather than newer tuples in input queues of an operator.

Data stream processing along with the solutions and algorithms developed in this thesis satisfies the needs of a large group of applications and will be widely accepted and used by applications from different domains and emerging applications as the cost of collecting data becomes cheaper. We believe that data stream processing has a bright future.

APPENDIX A

List of Experimental Query Plans

In this appendix, we list the continuous queries that we used in our experiments and their properties such as selectivity, processing capacity, and so on. For each query, we list the following details of the query:

1. C-SQL statement;
2. Query Plan;
3. Operator Properties Table;

A.1 Query T5_a

The SQL statement of the CQ T5_a is given in C-SQL Statement. Its physical query plan and the properties of each operator in the query plan are given in Figure A.1 and in Table A.1 respectively.

C-SQL Statement:

```
SELECT S0.networkAdd, S0.hostAdd, S0.port, S0.serviceType
FROM S0 JOIN (S1 JOIN S2 ON
(S1.protocolId == S2.protocolId)  [ Tuple 20000])
ON (S0.protocolId == S1.protocolId)  [ Tuple 20000]
WHERE stream0 AS S0 [(S0.hostAdd > 100 )&&(S0.serviceType >= 5)],
      stream1 AS S1 [(S1.hostAdd > 100 )&&(S1.serviceType <= 3)],
      stream2 AS S2 [(S2.hostAdd > 100 )&&(S2.serviceType >= 6)]
```

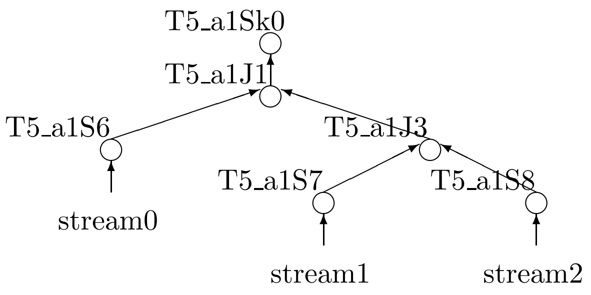

Figure A.1 Query Plans of T5_a.

Table A.1 Properties of Operators

operatorName	selectivity selectivity	right selectivity (if pliable)	pro_rate #tuples/second
T5_a1J1	0.076489	0.076426	12932.3
T5_a1J3	0.076436	0.075448	12904.5
T5_a1S6	0.365376	N/A	288706
T5_a1S7	0.487451	N/A	286432
T5_a1S8	0.244045	N/A	304208

A.2 Query T5_b

The SQL statement of the CQ T5_b is given in C-SQL Statement. Its physical query plan and the properties of each operator in the query plan are given in Figure A.2 and in Table A.2 respectively.

C-SQL Statement

```
SELECT S0.networkAdd, S0.hostAdd, S0.port, S0.serviceType
FROM (S0 JOIN S1 ON (S0.protocolId == S1.protocolId)  [ Tuple 20000])
        JOIN S2 ON (S1.protocolId == S2.protocolId)  [ Tuple 20000]
WHERE stream0 AS S0 [(S0.hostAdd > 100 )&&(S0.serviceType >= 5)],
      stream1 AS S1 [(S1.hostAdd > 100 )&&(S1.serviceType <= 3)],
      stream2 AS S2 [(S2.hostAdd > 100 )&&(S2.serviceType >= 6)]
```

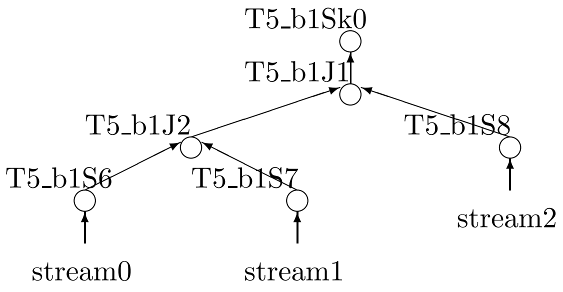

Figure A.2 Query Plans of T5_b.

Table A.2 Properties of Operators

operatorName	selectivity selectivity	right selectivity (if applicable)	pro_rate #tuples/second
T5_b1J1	0.074753	0.075361	13099.5
T5_b1J2	0.07697	0.075538	13069.9
T5_b1S6	0.365373	N/A	287880
T5_b1S7	0.487542	N/A	284132
T5_b1S8	0.24403	N/A	289945

A.3 Query T5_c

The SQL statement of the CQ T5_c is given in C-SQL Statement. Its physical query plan and the properties of each operator in the query plan are given in Figure A.3 and in Table A.3 respectively.

C-SQL Statement

```
SELECT S0.networkAdd, S0.hostAdd, S0.port, S0.serviceType
FROM (S0 JOIN S2 ON (S0.protocolId == S2.protocolId)  [ Tuple 20000])
JOIN S1 ON (S0.protocolId == S1.protocolId)  [ Tuple 20000]
WHERE stream0 AS S0 [(S0.hostAdd > 100 )&&(S0.serviceType <= 3)],
      stream1 AS S1 [(S1.hostAdd > 100 )&&(S1.serviceType >= 5)],
      stream2 AS S2 [(S2.hostAdd > 100 )&&(S2.serviceType >= 6)]
```

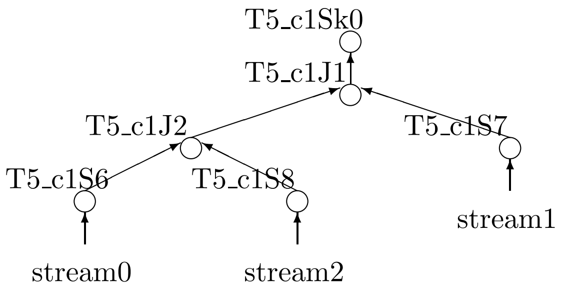

Figure A.3 Query Plans of T5_c.

Table A.3 Properties of Operators

operatorName	selectivity selectivity	right selectivity (if applicable)	pro_rate #tuples/second
T5_c1J1	0.074699	0.07544	13258.6
T5_c1J2	0.076502	0.076579	12922.6
T5_c1S6	0.487642	N/A	280660
T5_c1S7	0.36548	N/A	295909
T5_c1S8	0.244029	N/A	307833

A.4 Query T7_a

The SQL statement of the CQ T7_a is given in C-SQL Statement. Its physical query plan and the properties of each operator in the query plan are given in Figure A.4 and in Table A.4 respectively.

C-SQL Statement

```
SELECT S0.networkAdd, S0.hostAdd, S0.port, S0.serviceType
FROM (S0 JOIN S1 ON (S0.protocolId == S1.protocolId)  [ Tuple 20000])
    JOIN (S2 JOIN S3 ON (S2.protocolId == S3.protocolId)  [ Tuple 20000])
              ON (S0.protocolId == S2.protocolId)  [ Tuple 20000]
WHERE stream0 AS S0 [(S0.hostAdd > 100 )&&(S0.serviceType <= 6)],
      stream1 AS S1 [(S1.hostAdd > 500 )&&(S1.serviceType >= 3)],
      stream2 AS S2 [(S2.hostAdd < 1000 )&&(S2.serviceType <= 5)],
      stream3 AS S3 [(S3.hostAdd > 100 )&&(S3.serviceType >= 5)]
```

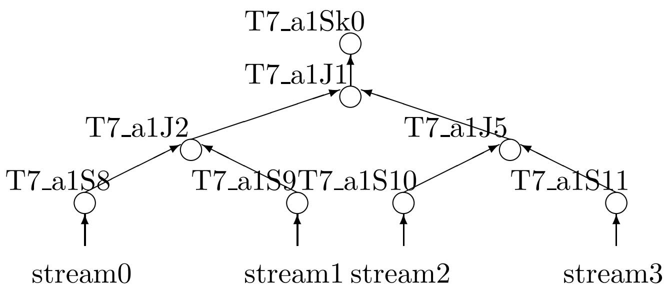

Figure A.4 Query Plans of T7_a.

Table A.4 Properties of Operators

operatorName	selectivity selectivity	right selectivity (if applicable)	pro_rate #tuples/second
T7_a1J1	0.0767896	0.0765271	14666.9
T7_a1J2	0.0765741	0.0760241	15356.5
T7_a1J5	0.0764634	0.0765628	14842.6
T7_a1S10	0.182757	N/A	332017
T7_a1S11	0.397383	N/A	319262
T7_a1S8	0.841642	N/A	292415
T7_a1S9	0.548975	N/A	311973

A.5 Query T7_b

The SQL statement of the CQ T7_b is given in C-SQL Statement. Its physical query plan and the properties of each operator in the query plan are given in Figure A.5 and in Table A.5 respectively.

C-SQL Statement

```
SELECT S0.networkAdd, S0.hostAdd, S0.port, S0.serviceType
FROM  (S0 JOIN S2 ON (S0.protocolId == S2.protocolId)  [ Tuple 20000])
    JOIN
      (S1 JOIN S3 ON (S1.protocolId == S3.protocolId)  [ Tuple 20000])
                 ON (S0.protocolId == S1.protocolId)  [ Tuple 20000]
WHERE  stream0 AS S0 [(S0.hostAdd > 100 )&&(S0.serviceType >= 5)],
       stream1 AS S1 [(S1.hostAdd > 100 )&&(S1.serviceType <= 6)],
       stream2 AS S2 [(S2.hostAdd > 100 )&&(S2.serviceType >= 2)],
       stream3 AS S3 [(S3.hostAdd > 500 )&&(S3.serviceType <= 5)]
```

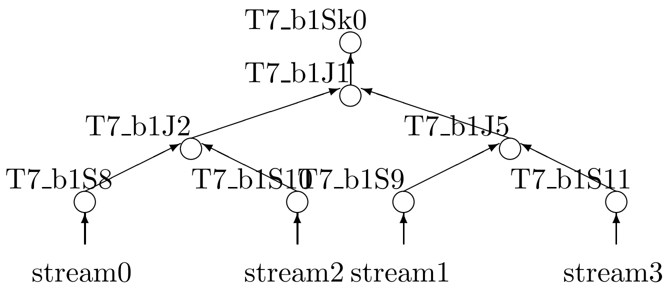

Figure A.5 Query Plans of T7_b.

Table A.5 Properties of Operators

operatorName	selectivity selectivity	right selectivity (if applicable)	pro_rate #tuples/second
T7_b1J1	0.076977	0.0759418	14733.3
T7_b1J2	0.0767028	0.0762223	15392.9
T7_b1J5	0.0770829	0.0758878	15480.5
T7_b1S10	0.665078	N/A	306234
T7_b1S11	0.629823	N/A	308307
T7_b1S8	0.451377	N/A	313630
T7_b1S9	0.81704	N/A	298194

A.6 Query T7_c

The SQL statement of the CQ T7_c is given in C-SQL Statement. Its physical query plan and the properties of each operator in the query plan are given in Figure A.6 and in Table A.6 respectively.

C-SQL Statement

```
SELECT S0.networkAdd, S0.hostAdd, S0.port, S0.serviceType
FROM  (S0 JOIN S3 ON (S0.protocolId == S3.protocolId)  [ Tuple 20000])
       JOIN
        (S2 JOIN S1 ON (S2.protocolId == S1.protocolId)  [ Tuple 20000])
                   ON (S0.protocolId == S2.protocolId)  [ Tuple 20000]
WHERE stream0 AS S0 [(S0.hostAdd > 100 )&&(S0.serviceType >= 5)],
      stream1 AS S1 [(S1.hostAdd > 100 )&&(S1.serviceType <= 5)],
      stream2 AS S2 [(S2.hostAdd > 100 )&&(S2.serviceType >= 4)],
      stream3 AS S3 [(S3.hostAdd > 100 )&&(S3.serviceType <= 4)]
```

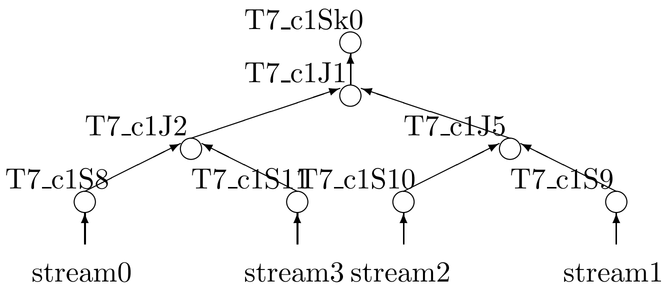

Figure A.6 Query Plans of T7_c.

Table A.6 Properties of Operators

operatorName	selectivity selectivity	right selectivity (if applicable)	pro_rate #tuples/second
T7_c1J1	0.0769256	0.0761092	14529.2
T7_c1J2	0.0768249	0.0766272	15399.4
T7_c1J5	0.075977	0.0767645	15287.4
T7_c1S10	0.458319	N/A	317689
T7_c1S11	0.579149	N/A	311219
T7_c1S8	0.438759	N/A	312566
T7_c1S9	0.703679	N/A	308879

A.7 Query T7_d

The SQL statement of the CQ T7_d is given in C-SQL Statement. Its physical query plan and the properties of each operator in the query plan are given in Figure A.7 and in Table A.7 respectively.

C-SQL Statement

```
SELECT S0.networkAdd, S0.hostAdd, S0.port, S0.serviceType
FROM  (((S0 JOIN S1 ON (S0.protocolId == S1.protocolId)  [ Tuple 20000])
      JOIN S2 ON (S0.protocolId == S2.protocolId)  [ Tuple 20000])
      JOIN S3 ON (S0.protocolId == S3.protocolId)  [ Tuple 20000])
WHERE stream0 AS S0 [(S0.hostAdd > 100 )&&(S0.serviceType >= 5)],
      stream1 AS S1 [(S1.hostAdd > 100 )&&(S1.serviceType <= 5)],
      stream2 AS S2 [(S2.hostAdd > 100 )&&(S2.serviceType >= 3)],
      stream3 AS S3 [(S3.hostAdd > 100 )&&(S3.serviceType <= 3)]
```

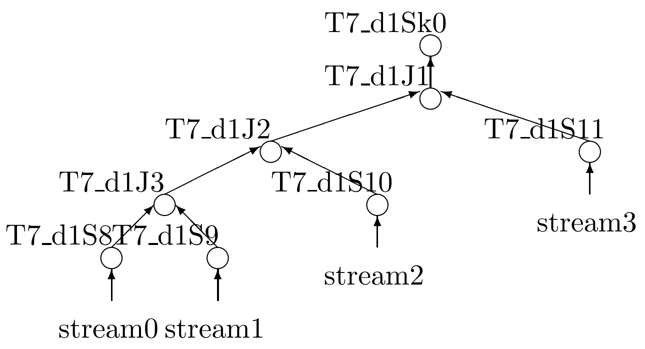

Figure A.7 Query Plans of T7_d.

Table A.7 Properties of Operators

operatorName	selectivity selectivity	right selectivity (if applicable)	pro_rate #tuples/second
T7_d1J1	0.076389	0.07623	13278.9
T7_d1J2	0.074184	0.076177	13315.3
T7_d1J3	0.0768	0.075839	13161.8
T7_d1S10	0.609688	N/A	278320
T7_d1S11	0.487402	N/A	278166
T7_d1S8	0.365409	N/A	285997
T7_d1S9	0.732454	N/A	271739

A.8 Query T7_e

The SQL statement of the CQ T7_e is given in C-SQL Statement. Its physical query plan and the properties of each operator in the query plan are given in Figure A.8 and in Table A.8 respectively.

C-SQL Statement

```
SELECT S0.networkAdd, S0.hostAdd, S0.port, S0.serviceType
FROM  (((S0 JOIN S2 ON (S0.protocolId == S2.protocolId)  [ Tuple 20000])
      JOIN S1 ON (S0.protocolId == S1.protocolId)  [ Tuple 20000])
      JOIN S3 ON (S0.protocolId == S3.protocolId)  [ Tuple 20000])
WHERE stream0 AS S0 [(S0.hostAdd > 100 )&&(S0.serviceType <= 3)],
      stream1 AS S1 [(S1.hostAdd > 100 )&&(S1.serviceType >= 5)],
      stream2 AS S2 [(S2.hostAdd > 100 )&&(S2.serviceType <= 2)],
      stream3 AS S3 [(S3.hostAdd > 100 )&&(S3.serviceType >= 6)]
```

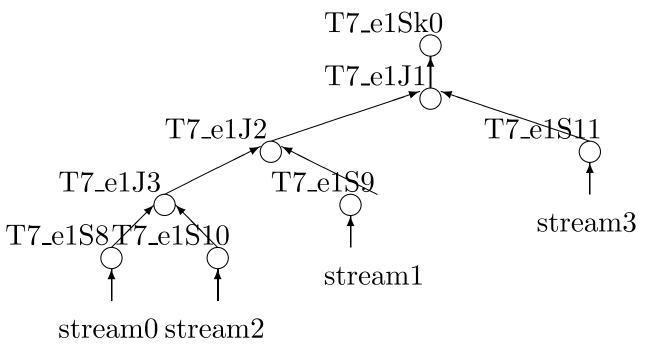

Figure A.8 Query Plans of T7_e.

Table A.8 Properties of Operators

operatorName	selectivity selectivity	right selectivity (if applicable)	pro_rate #tuples/second
T7_e1J1	0.076266	0.077657	13254.3
T7_e1J2	0.077361	0.076291	13101.6
T7_e1J3	0.075317	0.07654	12952.1
T7_e1S10	0.365653	N/A	292407
T7_e1S11	0.244382	N/A	296613
T7_e1S8	0.487598	N/A	276466
T7_e1S9	0.365661	N/A	297120

A.9 Query T7_f

The SQL statement of the CQ T7_f is given in C-SQL Statement. Its physical query plan and the properties of each operator in the query plan are given in Figure A.9 and in Table A.9 respectively.

C-SQL Statement

```
SELECT S0.networkAdd, S0.hostAdd, S0.port, S0.serviceType
FROM  (((S0 JOIN S3 ON (S0.protocolId == S3.protocolId)  [ Tuple 20000])
      JOIN S1 ON (S0.protocolId == S1.protocolId)  [ Tuple 20000])
      JOIN S2 ON (S0.protocolId == S2.protocolId)  [ Tuple 20000])
WHERE stream0 AS S0 [(S0.hostAdd > 100 )&&(S0.serviceType <= 2)],
      stream1 AS S1 [(S1.hostAdd > 100 )&&(S1.serviceType >= 5)],
      stream2 AS S2 [(S2.hostAdd > 100 )&&(S2.serviceType <= 3)],
      stream3 AS S3 [(S3.hostAdd > 100 )&&(S3.serviceType >= 6)]
```

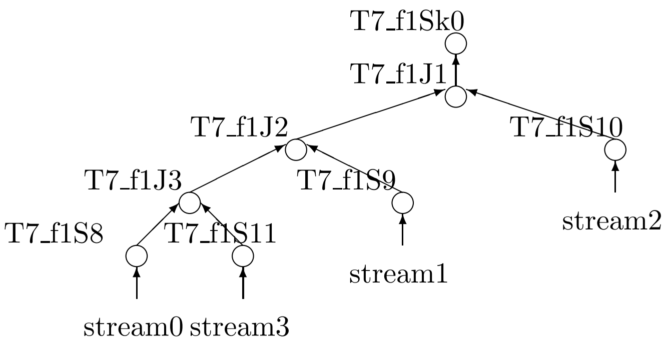

Figure A.9 Query Plans of T7_f.

Table A.9 Properties of Operators

operatorName	selectivity selectivity	right selectivity (if applicable)	pro_rate #tuples/second
T7_f1J1	0.072118	0.075375	13498.3
T7_f1J2	0.075059	0.075855	13125.6
T7_f1J3	0.076446	0.075717	12789.3
T7_f1S10	0.48775	N/A	282689
T7_f1S11	0.244381	N/A	296837
T7_f1S8	0.365629	N/A	282628
T7_f1S9	0.365629	N/A	291977

A.10 Query T7_g

The SQL statement of the CQ T7_g is given in C-SQL Statement. Its physical query plan and the properties of each operator in the query plan are given in Figure A.10 and in Table A.10 respectively.

C-SQL Statement

```
SELECT S0.networkAdd, S0.hostAdd, S0.port, S0.serviceType
FROM  S0 JOIN
        (S1 JOIN
         (S2 JOIN S3 ON (S2.protocolId == S3.protocolId) [ Tuple 20000])
             ON (S1.protocolId == S2.protocolId)  [ Tuple 20000])
          ON (S0.protocolId == S1.protocolId)  [ Tuple 20000]
WHERE stream0 AS S0 [(S0.hostAdd > 100 )&&(S0.serviceType >= 5)],
      stream1 AS S1 [(S1.hostAdd > 100 )&&(S1.serviceType <= 2)],
      stream2 AS S2 [(S2.hostAdd > 100 )&&(S2.serviceType >= 6)],
      stream3 AS S3 [(S3.hostAdd > 100 )&&(S3.serviceType <= 3)]
```

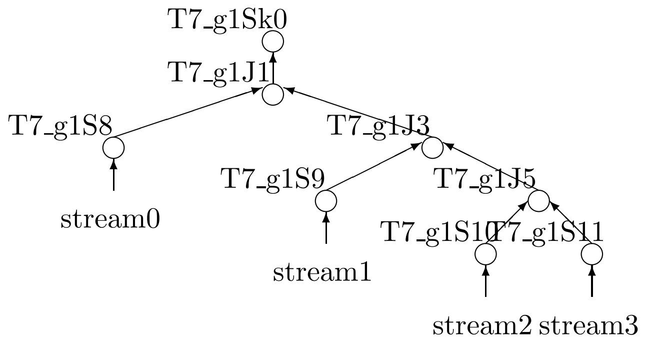

Figure A.10 Query Plans of T7_g.

Table A.10 Properties of Operators

operatorName	selectivity selectivity	right selectivity (if applicable)	pro_rate #tuples/second
T7_g1J1	0.0762252	0.0772773	15422.7
T7_g1J3	0.0763198	0.0762073	15072
T7_g1J5	0.0760592	0.0769382	15017.7
T7_g1S10	0.243878	N/A	327713
T7_g1S11	0.458441	N/A	316283
T7_g1S8	0.463258	N/A	317088
T7_g1S9	0.365881	N/A	322708

A.11 Query T7_h

The SQL statement of the CQ T7_h is given in C-SQL Statement. Its physical query plan and the properties of each operator in the query plan are given in Figure A.11 and in Table A.11 respectively.

C-SQL Statement

```
SELECT S0.networkAdd, S0.hostAdd, S0.port, S0.serviceType
FROM  S0 JOIN
          (S3 JOIN
     (S1 JOIN S2 ON (S1.protocolId == S2.protocolId)  [ Tuple 20000])
                         ON (S3.protocolId == S1.protocolId)  [ Tuple 20000])
       ON (S0.protocolId == S3.protocolId)  [ Tuple 20000]
WHERE stream0 AS S0 [(S0.hostAdd > 100 )&&(S0.serviceType <= 2)],
      stream1 AS S1 [(S1.hostAdd > 100 )&&(S1.serviceType >= 5)],
      stream2 AS S2 [(S2.hostAdd > 100 )&&(S2.serviceType <= 3)],
      stream3 AS S3 [(S3.hostAdd > 100 )&&(S3.serviceType >= 6)]
```

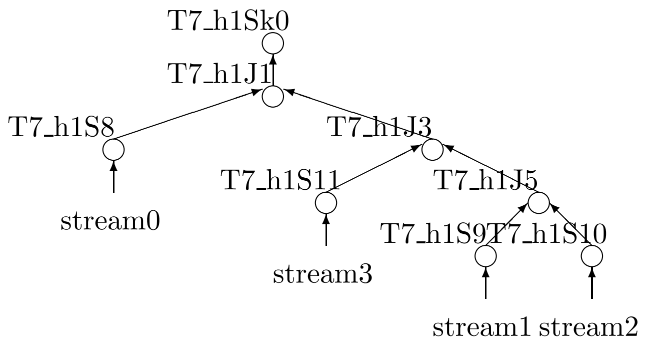

Figure A.11 Query Plans of T7_h.

Table A.11 Properties of Operators

operatorName	selectivity selectivity	right selectivity (if applicable)	pro_rate #tuples/second
T7_h1J1	0.0767856	0.0768352	15206.2
T7_h1J3	0.0805319	0.0790611	14356.1
T7_h1J5	0.0763591	0.0764978	15113.2
T7_h1S10	0.463546	N/A	315839
T7_h1S11	0.244232	N/A	325491
T7_h1S8	0.438139	N/A	314613
T7_h1S9	0.365712	N/A	321142

A.12 Query T9_a

The SQL statement of the CQ T9_a is given in C-SQL Statement. Its physical query plan and the properties of each operator in the query plan are given in Figure A.12 and in Table A.12 respectively.

C-SQL Statement

```
SELECT S0.networkAdd, S0.hostAdd, S0.port, S0.serviceType
FROM  ((((S4 JOIN S1 ON (S4.protocolId == S1.protocolId)  [ Tuple 20000])
      JOIN S2 ON (S4.protocolId == S2.protocolId)  [ Tuple 20000])
      JOIN S3 ON (S4.protocolId == S3.protocolId)  [ Tuple 20000])
      JOIN S0 ON (S4.protocolId == S0.protocolId)  [ Tuple 30000])
WHERE stream0 AS S0 [(S0.hostAdd > 100 )&&(S0.serviceType >= 5)],
      stream1 AS S1 [(S1.hostAdd > 100 )&&(S1.serviceType <= 5)],
      stream2 AS S2 [(S2.hostAdd > 100 )&&(S2.serviceType >= 3)],
      stream3 AS S3 [(S3.hostAdd > 100 )&&(S3.serviceType <= 3)],
      stream4 AS S4 [(S4.hostAdd > 100 )&&(S4.serviceType >= 2)]
```

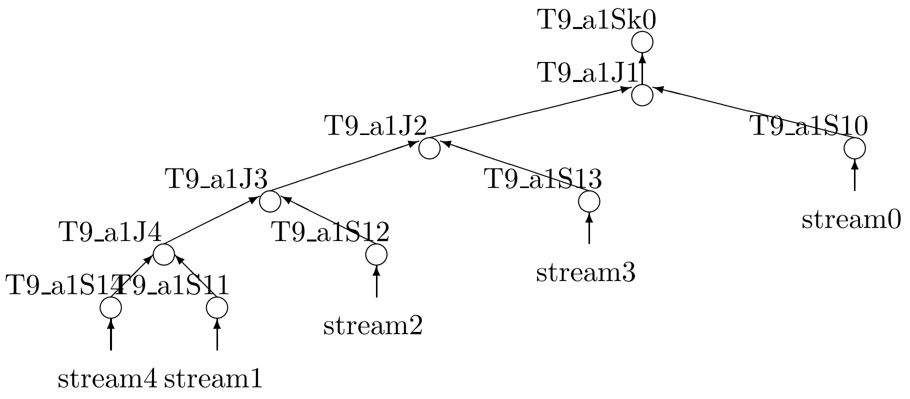

Figure A.12 Query Plans of T9_a.

Table A.12 Properties of Operators

operatorName	selectivity selectivity	right selectivity (if applicable)	pro_rate #tuples/second
T9_a1J1	0.105475	0.105671	11478.6
T9_a1J2	0.0760633	0.0766489	15399
T9_a1J3	0.0760277	0.0764726	15610.7
T9_a1J4	0.0767913	0.0765642	15336.3
T9_a1S10	0.45101	N/A	310669
T9_a1S11	0.716014	N/A	305312
T9_a1S12	0.565805	N/A	309834
T9_a1S13	0.475302	N/A	311873
T9_a1S14	0.719599	N/A	302650

A.13 Query T9_b

The SQL statement of the CQ T9_b is given in C-SQL Statement. Its physical query plan and the properties of each operator in the query plan are given in Figure A.13 and in Table A.13 respectively.

C-SQL Statement

```
SELECT S0.networkAdd, S0.hostAdd, S0.port, S0.serviceType
FROM  S0 JOIN (S1 JOIN (S2 JOIN (S3 JOIN S4
        ON (S3.protocolId == S4.protocolId)  [ Tuple 20000])
        ON (S2.protocolId == S3.protocolId)  [ Tuple 20000])
                ON (S1.protocolId == S2.protocolId)  [ Tuple 20000])
      ON (S0.protocolId == S1.protocolId)  [ Tuple 30000]
```

```
WHERE stream0 AS S0 [(S0.hostAdd > 100 )&&(S0.serviceType <= 5)],
      stream1 AS S1 [(S1.hostAdd > 100 )&&(S1.serviceType >= 5)],
      stream2 AS S2 [(S2.hostAdd > 100 )&&(S2.serviceType <= 3)],
      stream3 AS S3 [(S3.hostAdd > 100 )&&(S3.serviceType >= 4)],
      stream4 AS S4 [(S4.hostAdd > 100 )&&(S4.serviceType <= 5)]
```

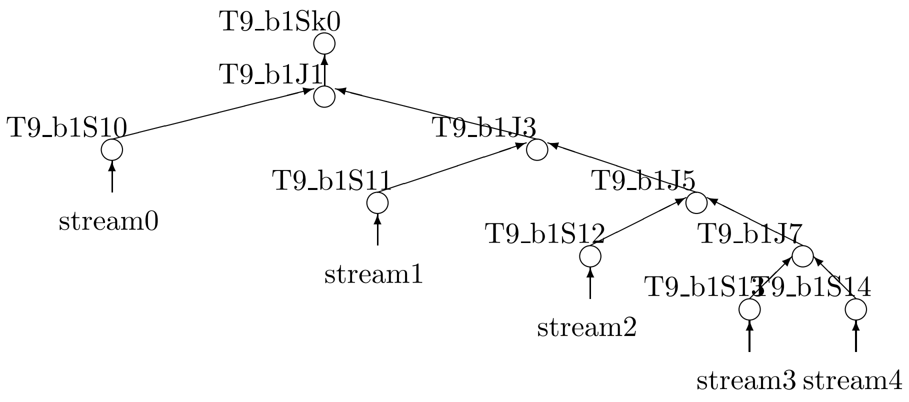

Figure A.13 Query Plans of T9_b.

Table A.13 Properties of Operators

operatorName	selectivity selectivity	right selectivity (if applicable)	pro_rate #tuples/second
T9_b1J1	0.107301	0.107323	11475.7
T9_b1J3	0.0766401	0.0757045	15379
T9_b1J5	0.0761899	0.0770867	15217.7
T9_b1J7	0.0766296	0.076385	15226.4
T9_b1S10	0.743672	N/A	292693
T9_b1S11	0.372883	N/A	320399
T9_b1S12	0.487549	N/A	313245
T9_b1S13	0.475848	N/A	315099
T9_b1S14	0.701362	N/A	304616

A.14 Query T9_c

The SQL statement of the CQ T9_c is given in C-SQL Statement. Its physical query plan and the properties of each operator in the query plan are given in Figure A.14 and in Table A.14 respectively.

C-SQL Statement

```
SELECT S0.networkAdd, S0.hostAdd, S0.port, S0.serviceType
FROM  (S0 JOIN S1 ON (S0.protocolId == S1.protocolId)  [ Tuple 20000])
      JOIN ((S2 JOIN S3 ON (S2.protocolId == S3.protocolId)  [ Tuple 20000])
           JOIN S4 ON (S2.protocolId == S4.protocolId)  [ Tuple 20000])
      ON (S0.protocolId == S2.protocolId)  [ Tuple 20000]
WHERE stream0 AS S0 [(S0.hostAdd > 100 )&&(S0.serviceType <= 5)],
      stream1 AS S1 [(S1.hostAdd > 100 )&&(S1.serviceType >= 4)],
      stream2 AS S2 [(S2.hostAdd > 100 )&&(S2.serviceType <= 3)],
      stream3 AS S3 [(S3.hostAdd > 100 )&&(S3.serviceType >= 3)],
      stream4 AS S4 [(S4.hostAdd > 100 )&&(S4.serviceType <= 6)]
```

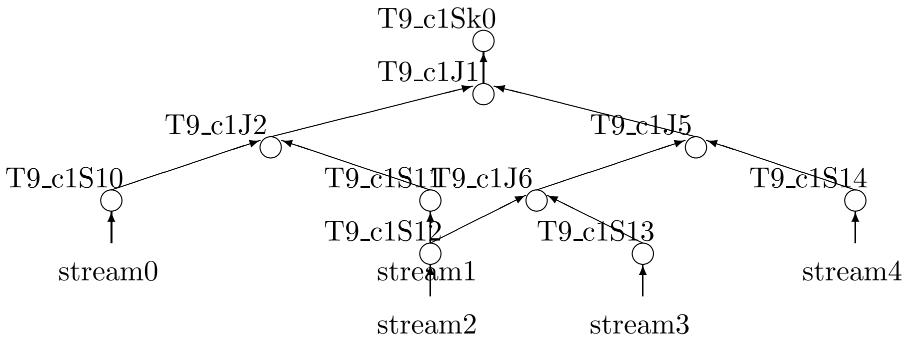

Figure A.14 Query Plans of T9_c.

Table A.14 Properties of Operators

operatorName	selectivity selectivity	right selectivity (if applicable)	pro_rate #tuples/second
T9_c1J1	0.0793667	0.0757112	14879.9
T9_c1J2	0.084171	0.0837157	14323
T9_c1J5	0.0759827	0.0764632	15474.5
T9_c1J6	0.0763753	0.0767848	15077
T9_c1S10	0.755938	N/A	297103
T9_c1S11	0.475429	N/A	314064
T9_c1S12	0.470528	N/A	317018
T9_c1S13	0.579459	N/A	310279
T9_c1S14	0.780705	N/A	300406

A.15 Query T9_d

The SQL statement of the CQ T9_d is given in C-SQL Statement. Its physical query plan and the properties of each operator in the query plan are given in Figure A.15 and in Table A.15 respectively.

C-SQL Statement

```
SELECT S0.networkAdd, S0.hostAdd, S0.port, S0.serviceType
FROM  ((S0 JOIN S1 ON (S0.protocolId == S1.protocolId) [ Tuple 20000])
           JOIN S2 ON (S0.protocolId == S2.protocolId) [ Tuple 20000])
  JOIN (S3 JOIN S4 ON (S3.protocolId == S4.protocolId) [ Tuple 20000])
      ON (S0.protocolId == S3.protocolId)  [ Tuple 20000]
WHERE stream0 AS S0 [(S0.hostAdd > 100 )&&(S0.serviceType >= 5)],
      stream1 AS S1 [(S1.hostAdd > 100 )&&(S1.serviceType <= 5)],
      stream2 AS S2 [(S2.hostAdd > 100 )&&(S2.serviceType >= 4)],
      stream3 AS S3 [(S3.hostAdd > 100 )&&(S3.serviceType >= 3)],
      stream4 AS S4 [(S4.hostAdd > 100 )&&(S4.serviceType <= 6)]
```

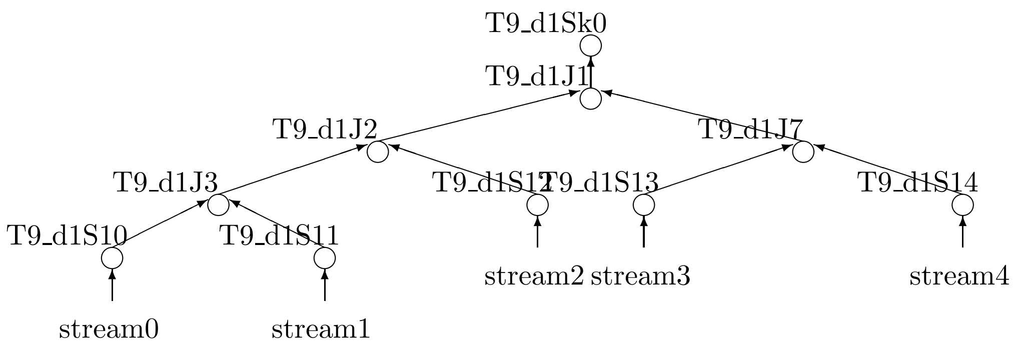

Figure A.15 Query Plans of T9_d.

Table A.15 Properties of Operators

operatorName	selectivity selectivity	right selectivity (if applicable)	pro_rate #tuples/second
T9_d1J1	0.0814745	0.0781117	14851.3
T9_d1J2	0.0763706	0.0762074	15216.2
T9_d1J3	0.0841094	0.0843216	14385.6
T9_d1J7	0.0760247	0.0762887	15515.4
T9_d1S10	0.437332	N/A	316206
T9_d1S11	0.753574	N/A	300337
T9_d1S12	0.423588	N/A	318579
T9_d1S13	0.600737	N/A	308715
T9_d1S14	0.829362	N/A	296772

A.16 Query T9_e

The SQL statement of the CQ T9_e is given in C-SQL Statement. Its physical query plan and the properties of each operator in the query plan are given in Figure A.16 and in Table A.16 respectively.

C-SQL Statement

```
SELECT S0.networkAdd, S0.hostAdd, S0.port, S0.serviceType
FROM S0 JOIN (
         (S1 JOIN S2 ON (S1.protocolId == S2.protocolId) [Tuple 20000])
         JOIN
         (S3 JOIN S4 ON (S3.protocolId == S4.protocolId) [Tuple 20000])
         ON (S1.protocolId == S3.protocolId)  [ Tuple 20000])
```

```
      ON (S0.protocolId == S1.protocolId)  [ Tuple 30000]
WHERE stream0 AS S0 [(S0.hostAdd > 128 )&&(S0.serviceType >= 5)],
      stream1 AS S1 [(S1.hostAdd > 200 )&&(S1.serviceType <= 5)],
      stream2 AS S2 [(S2.hostAdd > 100 )&&(S2.serviceType >= 3)],
      stream3 AS S3 [(S3.hostAdd > 500 )&&(S3.serviceType <= 3)],
      stream4 AS S4 [(S4.hostAdd > 100 )&&(S4.serviceType >= 2)]
```

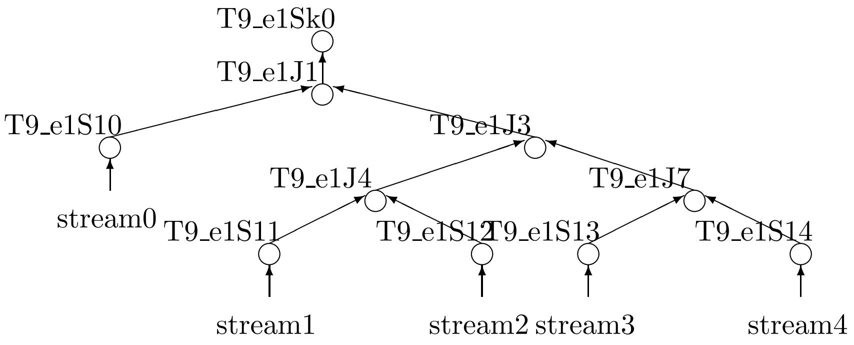

Figure A.16 Query Plans of T9_e.

Table A.16 Properties of Operators

operatorName	selectivity selectivity	right selectivity (if applicable)	pro_rate #tuples/second
T9_e1J1	0.107645	0.107062	11373.1
T9_e1J3	0.0768394	0.0771324	14703.9
T9_e1J4	0.0764475	0.0762762	15393.1
T9_e1J7	0.0761344	0.0762441	15269.7
T9_e1S10	0.438456	N/A	318381
T9_e1S11	0.68573	N/A	306286
T9_e1S12	0.597537	N/A	308851
T9_e1S13	0.465752	N/A	317484
T9_e1S14	0.730004	N/A	303850

REFERENCES

[1] Borealis home page, http://nms.lcs.mit.edu/projects/borealis/.

[2] Ebay home page, http://www.ebay.com.

[3] Google home page, http://www.google.com.

[4] ipolicy networks home page. http://www.ipolicynetworks.com.

[5] Mavhome home page, http://cygnus.uta.edu/mavhome/.

[6] Stanford stream data management (stream) project. http://www-db.stanford.edu/stream.

[7] Yahoo home page, http://www.yahoo.com.

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

[9] R. Adaikkalavan and S. Chakravarthy. SnoopIB: Interval-Based Event Specification and Detection for Active Databases. In *Proceedings, East-European Conference on Advances in Databases and Information Systems*, Sep. 2003.

[10] R. Adaikkalavan and S. Chakravarthy. Formalization and Detection of Events Over a Sliding Window in Active Databases Using Interval-Based Semantics. In *Proceedings, East-European Conference on Advances in Databases and Information Systems*, Sep. 2004.

[11] L. Amsaleg, M. Franklin, and A. Tomasic. Dynamic query operator scheduling for wide-area remote access. *Journal of Distributed and Parallel Databases*, 3(6), July 1998.

[12] A. Arasu, B. Babcock, S. Babu, J. McAlister, and J. Widom. Characterizing memory requirements for queries over continuous data streams. In *Proceedings, International Conference on Principles of Database Systems*, pages 221–232, Jun. 2002.

[13] A. Arasu, S. Babu, and J. Widom. Cql: A language for continuous queries over streams and relations. In *Database Programming Languages, 9th International Workshop, DBPL 2003, Potsdam, Germany, September 6-8, 2003*, pages 1–19, 2003.

[14] A. Arasu, S. Babu, and J. Widom. The CQL Continuous Query Language: Semantic Foundations and Query Execution. *Stanford Technical Report*, Oct. 2003.

[15] A. Arasu et al. Linear Road: A Stream Data Management Benchmark. In *Proceedings, International Conference on Very Large Data Bases*, Sep. 2004.

[16] A. Arasu and J. Widom. Resource sharing in continuous sliding-window aggregates. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases, Toronto, Canada*, pages 336–347, Aug. 2004.

[17] R. Avnur and J. Hellerstein. Eddies: Continuously adaptive query processing. *In Proc. of the 2000 ACM SIGMOD Intl. Conf. on Management of Data*, pages 261–272, May 2000.

[18] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 1–16, June 2002.

[19] B. Babcock, M. Datar, and R. Motwani. Sampling from a moving window over streaming data. In *Proceedings of the 2002 Annual ACM-SIAM Symp. on Discrete Algorithms*, 2002.

[20] B. Babcock, M. Datar, and R. Motwani. Load shedding for aggregation queries over data streams. In *Proceedings of International Conference on Data Engineering (ICDE)*, Match 2004.

[21] B. Babcok, S. Babu, M. Datar, R. Motwani, and D. Thomas. Operator scheduling in data stream systems. *The VLDB Journal*, 13:333–353, 2004.

[22] S. Babu, K. Munagala, J. Widom, and R. Motwani. Adaptive caching for continuous queries. In *Proceedings of the 21st International Conference on Data Engineering, ICDE 2005, 5-8 April 2005, Tokyo, Japan*, pages 118–129, 2005.

[23] S. Babu, U. Srivastava, and J. Widom. Exploiting -constraints to reduce memory overhead in continuous queries over data streams. *ACM Trans. Database Syst.*, 29(3):545–580, 2004.

[24] S. Babu and J. Widom. Continuous queries over data streams. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 109–120, September 2001.

[25] J. Beran. *Statistics for Long-Memory Processes.* New York: Chapman and Hall, 1994.

[26] R. Birgisson, J. Mellin, and S. F. Andler. Bounds on Test Effort for Event-Triggered Real-Time Systems. In *International Workshop on Real-Time Computing and Applications Symposium*, 1999.

[27] W. Bischof. Analysis of m/g/1-queues with setup times and vacations under six different service disciplines. *Queueing systems*, 39, December 2001.

[28] P. Bonnet, J. E. Gerhke, and P. Seshadri. Towards Sensor Database Systems. In *Proceedings, International Conference on Mobile Data Management*, Jan. 2001.

[29] B. Brian, B. Shivnath, D. Mayur, and M. Rajeev. Chain: Operator scheduling for memory minimization in stream systems. In *Proc. of the International Conference on Management of Data (SIGMOD)*, June 2003.

[30] R. M. Brian Babcock, Mayur Datar. Load shedding for aggregation queries over data streams. In *Proc. of the 20th International Conference on Data Engineering*, March 2004.

[31] A. P. Buchmann et al. *Rules in an Open System: The REACH Rule System.* Rules in Database Systems, 1993.

[32] D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. TatbuL, and S. Zdonik. Monitoring streams - a new class of data management applications. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*, 2002.

[33] D. Carney, U. etintemel, A. Rasin, S. Zdonik, M. Cherniack, and M. Stonebraker. Operator scheduling in a data stream manager. *In Proc. Of the 2003 Intl. Conf. On Very Large Data Bases*, 2003.

[34] C.Fricker and M. Jaibi. Monotonicity and stability of polling models. *Queueing systems*, 15, 1994.

[35] S. Chakravarthy et al. HiPAC: A Research Project in Active, Time-Constrained Database Management (Final Report). Technical Report XAIT-89-02, Xerox Advanced Information Technology, Cambridge, MA, Aug. 1989.

[36] S. Chakravarthy et al. Composite Events for Active Databases: Semantics, Contexts, and Detection. In *Proceedings, International Conference on Very Large Data Bases*, pages 606–617, 1994.

[37] S. Chakravarthy et al. Design of Sentinel: An Object-Oriented DBMS with Event-Based Rules. *Information and Software Technology*, 36(9):559–568, 1994.

[38] S. Chakravarthy and D. Mishra. Snoop: An Expressive Event Specification Language for Active Databases. *Data and Knowledge Engineering*, 14(10):1–26, Oct. 1994.

[39] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. A. Shah. Telegraphcq: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

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

[41] J. Chen, D. Dewitt, F. Tian, and Y. Wang. Niagaracq: A scalable continuous query system for internet databases. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 379–390, 2000.

[42] D. J. Cook, M. Youngblood, E. Heierman, K. Gopalratnam, S. Rao, A. Litvin, and F. Khawaja. Mavhome: An agent-based smart home. In *Proceedings of the IEEE International Conference on Pervasive Computing and Communications*, pages 521–524, 2003.

[43] M. E. Crovella and A. Bestavros. Self-similarity in World Wide Web traffic: evidence and possible causes. *IEEE /ACM Transactions on Networking*, 5(6):835–846, 1997.

[44] A. Das, J. Gehrke, and M. Riedewald. Approximate join processing over data streams. In *Proc. of the International Conference on Management of Data (SIGMOD)*, June 2003.

[45] U. Dayal et al. The HiPAC Project: Combining Active Databases and Timing Constraints. *SIGMOD Record*, 17(1):51–70, Mar. 1988.

[46] A. Deshpande. An initial study of overheads of eddies. *SIGMOD Record*, 33(1):44–49, 2004.

[47] O. Diaz, N. Paton, and P. Gray. Rule Management in Object-Oriented Databases: A Unified Approach. In *Proceedings, International Conference on Very Large Data Bases*, Sep. 1991.

[48] A. Dinn, M. H. Williams, and N. W. Paton. ROCK & ROLL: A Deductive Object-Oriented Database with Active and Spatial Extensions. In *Proceedings, International Conference on Data Engineering*, 1997.

[49] L. Elkhalifa. InfoFilter: Complex Pattern Specification and Detection Over Text Streams. Master's thesis, The University of Texas at Arlington, 2004.

[50] H. Engstrom, M. Berndtsson, and B. Lings. Acood essentials. Technical report, University of Skovde, 1997.

[51] A. Galton and J. Augusto. Two Approaches to Event Definition. In *Proceedings, International Conference on Database and Expert Systems Applications*, 2002.

[52] S. Gatziu and K. R. Dittrich. SAMOS: An Active, Object-Oriented Database System. *IEEE Quarterly Bulletin on Data Engineering*, 15(1-4):23–26, Dec. 1992.

[53] S. Gatziu and K. R. Dittrich. Events in an Object-Oriented Database System. In *Proceedings of Rules in Database Systems*, Sep. 1993.

[54] S. Gatziu and K. R. Dittrich. Detecting Composite Events in Active Databases using Petri Nets. In *Proceedings of Workshop on Research Issues in Data Engineering*, Feb. 1994.

[55] N. H. Gehani and H. V. Jagadish. Ode as an Active Database: Constraints and Triggers. In *Proceedings, International Conference on Very Large Data Bases*, pages 327–336, Sep. 1991.

[56] N. H. Gehani, H. V. Jagadish, and O. Shmueli. COMPOSE: A System For Composite Event Specification and Detection. Technical report, AT&T Bell Laboratories, Dec. 1992.

[57] N. H. Gehani, H. V. Jagadish, and O. Shmueli. Composite Event Specification in Active Databases: Model & Implementation. In *Proceedings, International Conference on Very Large Data Bases*, pages 327 – 338, 1992.

[58] N. H. Gehani, H. V. Jagadish, and O. Shmueli. Event Specification in an Object-Oriented Database. In *Proceedings, International Conference on Management of Data (SIGMOD)*, pages 81–90, San Diego, CA, June 1992.

[59] A. Gilani. Design and implementation of stream operators, query instantiator and stream buffer manager. Master's thesis, The University of Texas at Arlington, 2004.

[60] L. Golab and M. T. zsu. Processing sliding window multi-joins in continuous queries over data streams. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*, September 2003.

[61] D. Goldberg, D. Nichols, B. M. Oki, and D. B. Terry. Using collaborative filtering to weave an information tapestry. *Commun. ACM*, 35(12):61–70, 1992.

[62] S. Guha, N. Koudas, and K. Shim. Data-streams and histograms. In *Proceedings of the 2001 Annual ACM Symp. on Theory of Computing*, pages 471–475, July 2001.

[63] E. N. Hanson. The Design and Implementation of the Ariel Active Database Rule System. *IEEE Transactions on Knowledge and Data Engineering*, 8(1), 1996.

[64] E. N. Hanson. Active Rules in Database Systems. pages 221–232. Springer, New York, 1999.

[65] J. Hellerstein, M. Franklin, and et al. Adaptive query processing: Technology in evolution. *IEEE Data Engineering Bulletin*, 23(2):7–18, June 2000.

[66] J. Hellerstein, S. Madden, V. Raman, and M. Shah. Continuously adaptive continuous queries over streams. *In Proc. of the 2002 ACM SIGMOD Intl. Conf. on Management of Data*, June 2002.

[67] M. R. Henzinger, P. Raghavan, and S. Rajagopalan. Computing on data streams. *TR-1998-011, Compaq Systems Research Center, Palo Alto, CA*, May 1998.

[68] E. Horvitz and S. Zilberstein. Computational tradeoffs under bounded resources. *Artif. Intell.*, 126(1-2):1–4, 2001.

[69] J. H.V and C. Faloutsos. Data reduction - a tutorial. In *Proceedings of Forth international conference on Knowledge Discovery and Data Mining*, August 1998.

[70] V. Jacobson and M. J. Karels. Congestion avoidance and control. *Computer Communication Review*, 18(4), August 1988.

[71] J.Daigle and M.Roughan. Queue-length distributions for multi-priority queueing systems. In *INFOCOM'99*, pages 641–648, March 1999.

[72] Q. Jiang, R. Adaikkalavan, and S. Chakravarthy. Estream: Towards an integration model for event and stream processing. *Technical Report CSE-2004-10, UT Arlington*, November 2004.

[73] Q. Jiang and S. Chakravarthy. Analysis and validation of continuous queries over data streams. *TR CSE-2003-7, UT Arlington*, July 2003.

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

[75] Q. Jiang and S. Chakravarthy. Queueing analysis of relational operators for continuous data streams. In *12th International Conference on Information and Knowledge Management (CIKM'03)*, November 2003.

[76] Q. Jiang and S. Chakravarthy. Data stream management system for mavhome. In *Proceedings of the 2004 ACM Symposium on Applied Computing*, Mar. 2004.

[77] Q. Jiang and S. Chakravarthy. Scheduling strategies for processing continuous queries over streams. In *21st Annual British National Conference on Databases*, July 2004.

[78] Q. Jiang and S. Chakravarthy. NFM^i: An inter-domain network fault management systems. In *Proceedings of International Conference on Data Engineering (ICDE)*, April 2005.

[79] J. Kang, J. F. Naughton, and S. Viglas. Evaluating window joins over unbounded streams. In *Proceedings of International Conference on Data Engineering (ICDE)*, Match 2003.

[80] D. G. Kendall. Some problems in the theory of queues. *Journal of the Royal Statistical Society*, B 13:151–185, 1951.

[81] L. Kleinrock. *Queueing Systems, Volumes I and II.* John Wiley & Sons, 1975.

[82] A. Kotz-Dittrich. Adding Active Functionality to an Object-Oriented Database System - a Layered Approach. In *Proc. of the Conference on Database Systems in Office, Technique and Science*, Mar. 1993.

[83] K. Larson and T. Sandholm. Bargaining with limited computation: deliberation equilibrium. *Artif. Intell.*, 132(2):183–217, 2001.

[84] T. T. Lee. M/g/1/n queue with vacation time and limited service discipline. 9:180 – 190, 1989.

[85] W. Leladn, M. Taqqu, W. Willinger, and D. Wilson. On the self-similar nature of ethernet traffic. *IEEE/ACM transaction on Networking*, Feb 1994.

[86] W. E. Leland, M. S. Taqq, W. Willinger, and D. V. Wilson. On the self-similar nature of Ethernet traffic. In D. P. Sidhu, editor, *ACM SIGCOMM*, pages 183–193, San Francisco, California, 1993.

[87] D. L. Lieuwen, N. H. Gehani, and R. Arlein. The Ode Active Database: Trigger Semantics and Implementation. In *Proceedings, International Conference on Data Engineering*, pages 412–420, Mar. 1996.

[88] L. Liu, C. Pu, and W. Tang. Continual queries for internet scale event-driven information delivery. *Knowledge and Data Engineering*, 11(4):610–628, 1999.

[89] S. Madden and M. J. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *Proceedings of International Conference on Data Engineering (ICDE)*, 2002.

[90] S. R. Madden et al. TAG: a Tiny AGgregation Service for Ad-Hoc Sensor Networks. *In Proc. of OSDI*, Dec. 2002.

[91] S. R. Madden et al. The Design of an Acquisitional Query Processor for Sensor Networks. In *Proceedings, International Conference on Management of Data (SIGMOD)*, Jun. 2003.

[92] J. Medhi. *Stochastic Models in Queueing Theory.* Academic Press, 1999.

[93] D. Menasce, D. A. Menasce, and V. A. F. Almeida. *Capacity Planning for Web Performance: metrics, models, and methods.* Prentice Hall, 1998.

[94] D. A. Menasc, V. A. F. Almeida, and L. W. Dowdy. *Capacity Planning and Performance Modeling: from mainframes to client-server systems.* Prentice Hall, 1994.

[95] M. F. Mokbel et al. PLACE: A Query Processor for Handling Real-time Spatio-temporal Data Streams. In *Proceedings, International Conference on Very Large Data Bases.*

[96] I. Motakis and C. Zaniolo. Formal Semantics for Composite Temporal Events in Active Database Rules. *Journal of System Integration*, 7(3-4):291–325, 1997.

[97] I. Motakis and C. Zaniolo. Temporal Aggregation in Active Database Rules. In *Proceedings, International Conference on Management of Data (SIGMOD)*, pages 440–451, 1997.

[98] R. Motwani et al. Query Processing, Resource Management, and Approximation in a Data Stream Management System. In *Proceedings, Conference on Innovative Data Systems Research*, Jan. 2003.

[99] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. S. Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, approximation, and resource management in a data stream management system. In *CIDR*, 2003.

[100] C. Olston, J. Jiang, and J. Widom. Adaptive filters for continuous queries over distributed data streams. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003*, pages 563–574, 2003.

[101] N. W. Paton. *Active Rules in Database Systems.* Springer, New York, 1999.

[102] K. K. Ramakrishnan and R. Jain. A binary feedback scheme for congestion avoidance in computer networks. *ACM Transactions on Computer Systems*, 8(2):158–181, 1990.

[103] C. Roncancio. Toward Duration-Based, Constrained and Dynamic Event Types. In *Active, Real-Time, and Temporal Database Systems*, pages 176–193, 1997.

[104] I. Rubin and J. C.-H. Wu. Analysis of an m/g/1/n queue with vacations and its iterative application to fddi timed-token rings. 3:842–856, 1995.

[105] U. Schreier et al. Alert: An Architecture for Transforming a Passive DBMS into an Active DBMS. In *Proceedings, International Conference on Very Large Data Bases*, 1991.

[106] P. Seshadri, M. Livny, and R. Ramakrishnan. The Design and Implementation of a Sequence Database System. In *Proceedings, International Conference on Very Large Data Bases*, pages 99–110, 1996.

[107] S. Sonune. Design and implementation of windowed operators and scheduler for stream data. Master's thesis, UT Arlington, 2004.

[108] U. Srivastava and J. Widom. Memory-Limited Execution of Windowed Stream Joins. In *Proceedings, International Conference on Very Large Data Bases*, Sep. 2004.

[109] W. Stallings. *High-speed networks and Internets. Second Edition.* Prentice Hall, 2002.

[110] M. Sullivan and A. Heybey. Tribeca: A system for managing large databases of network traffic. In *usenix*, June 1998.

[111] H. Takagi. *Analysis of Polling system.* MIT, Cambridge, MS, 1986.

[112] H. Takagi. Queuing analysis of polling models. In *ACM Computing Surveys (CSUR)*, volume 20, pages 5–28, March 1988.

[113] H. Takagi. *Queueing analysis, Vol. 1.* North-Holland, Amsterdam, 1991.

[114] N. Tatbul, U. Cetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker. Load shedding in a data stream manager. In *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB)*, September 2003.

[115] D. B. Terry, D. Goldberg, D. Nichols, and B. M. Oki. Continuous queries over append-only databases. In M. Stonebraker, editor, *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data, San Diego, California, June 2-5, 1992*, pages 321–330. ACM Press, 1992.

[116] T. Urhan and M. Franklin. Xjoin: A reactively-scheduled pipelined join operator. *IEEE Data Engineering Bulletin*, 23(2):27–33, June 2000.

[117] T. Urhan and M. Franklin. Dynamic pipeline scheduling for improving interactive performance of online queries. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*, Sept 2001.

[118] S. D. Viglas and J. F. Naughton. Rate-based query optimization for streaming information sources. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2002.

[119] S. D. Viglas and J. F. Naughton. Rate-based query optimization for streaming information sources. In *SIGMOD*, 2002.

[120] S. D. Viglas, J. F. Naughton, and J. Burger. Maximizing the output rate of multi-way join queries over streaming information sources. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*, 2003.

[121] J. Widom and S. Ceri. *Active Database Systems: Triggers and Rules.* Morgan Kaufmann Publishers, Inc., 1996.

[122] N. Wilschut and P. M. G. Apers. Pipelining in query execution. In *Conference on Databases, Parallel Architectures and their Applications*, 1991.

[123] C. Yang and A. Reddy. A taxonomy for congestion control algorithms in packet switching networks. *IEEE Network*, July/August 1995.

[124] Y. Yao and J. E. Gehrke. Query Processing in Sensor Networks. In *Proceedings, Conference on Innovative Data Systems Research*, Jan. 2003.

[125] C. Zaniolo et al. Stream Mill, Available [Online]: http://wis.cs.ucla.edu/stream-mill/index.html.

[126] J. Zhou and K. A. Ross. Buffering database operations for enhanced instruction cache performance. In *SIGMOD Conference*, pages 191–202, 2004.

[127] S. Zilberstein and S. Russell. Optimal composition of real-time systems. *Artif. Intell.*, 82(1-2):181–213, 1996.

BIOGRAPHICAL STATEMENT

Qingchun Jiang graduated from the University of Texas at Arlington with his Ph.D degree in computer science and engineering in August 2005. His research interests are mainly focused on data management and processing, including database management systems, data stream management systems, query processing, resource allocation, and QoS management. His other research interests include network management and information retrial and integration.

Before he joined The University of Texas at Arlington to pursue his Ph.D in August, 2001, He worked in the group of Network Fault Management in Bell-Labs (China), Lucent Technologies as a Member of Technical Staff after he received his Master of Engineering in computer science and engineering from Harbin Institute of Technology in July 1997.