ESTREAM: AN INTEGRATION OF EVENT AND STREAM PROCESSING

by

VIHANG GARG

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE AND ENGINEERING

THE UNIVERSITY OF TEXAS AT ARLINGTON

December 2005

To my Family, Friends and my advisor.

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my advisor, Dr. Sharma Chakravarthy, for giving me an opportunity to work on an interesting and challenging Estream system and constantly providing me guidance and support throughout the duration of this thesis. I am also grateful to Dr. Gautam Das and Dr. Jung-Hwan Oh for serving on my committee. I would like to thank Raman Adaikkalavan and Balakumar Kendai for helping me throughout the design and implementation of this work. I also thank Vamshi Pajjuri, Sunit Shrestha, Dhawal Bhatia, Srihari Padmanabhan and all friends in ITLAB for their invaluable help and advice during the course of development of this system.

This work was supported, in part by NSF (grants IIS-0123730, ITR 0121297 and IIS-0326505).

Last but not the least i would thank my parents Mr. Mahesh Kumar Garg and Mrs. Chhaya Garg, my brother Tarang and my fiancee Deepti for their constant love and support. Without their encouragement and endurance, this work would not have been possible.

November 4, 2004

ABSTRACT

ESTREAM: AN INTEGRATION OF EVENT AND STREAM PROCESSING

Publication No. ______

VIHANG GARG, M.S.

The University of Texas at Arlington, 2005

Supervising Professor: Sharma Chakravarthy

Event and stream data processing models have been independently researched extensively and are utilized in diverse application domains. Advanced applications require both event and stream processing, which is currently not supported in the same system. Although there are a number of similarities and differences between them, a synergistic integration of their strengths will be better than the sum of their parts.

In this thesis, we present EStream, an integrated event and stream processing system for monitoring changes on stream computations and for expressing and processing complex events on continuous queries (CQs). We introduce attribute-based constraints for reducing uninteresting events that are generated from CQs. We discuss the generalized specification of CQs, complex events, and rules. We also discuss stream modifiers, a special class of stream operators for computing changes over stream data.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS iii

ABSTRACT iv

LIST OF FIGURES ix

Chapter

1. INTRODUCTION 1
   - 1.1 EStream Applications 1
   - 1.2 Motivation 1
   - 1.3 Motivating Examples 3
     - 1.3.1 Car ADN 3
     - 1.3.2 Network Fault Management 6
   - 1.4 Contributions 7
2. RELATED WORK 9
   - 2.1 Data Stream Management Systems 9
     - 2.1.1 Aurora 9
     - 2.1.2 STREAMS 10
     - 2.1.3 The COUGAR Sensor Database System 11
     - 2.1.4 Fjord: Architecture for Queries Over Streaming Senor 13
   - 2.2 Event Processing Systems 14
     - 2.2.1 ODE 14
     - 2.2.2 SAMOS 15
     - 2.2.3 Sentinel or LED 16
   - 2.3 Event and stream integration systems 17

2.3.1 HiFi 17

2.3.2 Tiny DB 18

2.3.3 Financial applications 19

2.4 Summary 19

3. THE MAVSTREAM AND THE LOCAL EVENT DETECTOR (LED) 21

3.1 MavStream 21

3.1.1 Operators 22

3.1.2 Instantiator 23

3.1.3 Scheduler 24

3.1.4 MavStream Server 25

3.1.5 Data Flow computational model 26

3.2 Local Event Detector 27

3.2.1 Event types supported by LED 28

3.2.2 Parameter Context 31

3.2.3 Rule Priority 32

3.2.4 LED Computational Model 32

3.3 Data Flow Model Vs. Event Detection Graph 33

3.3.1 Inputs and Outputs 33

3.3.2 Event Operators and Stream Operators 34

3.3.3 QoS Support 34

3.3.4 Reusability 35

3.4 Summary 35

4. DESIGN 37

4.1 Issues 39

4.1.1 Event as a Single Tuple or a Collection of Tuples 39

4.1.2 Event and Continuous Query Definition and Mapping 40

4.1.3 How shall the two models be integrated . . . 40
4.1.4 Address Space Issue . . . 41
4.2 Continuous Event Query (or CEQ) . . . 41
4.3 Masks . . . 43
4.3.1 Where to add masks . . . 44
4.4 Integration Design . . . 46
4.4.1 Event Generator (EG) Interface . . . 46
4.4.2 Consumer/Producer Approach . . . 51
4.4.3 Rules and Composite event Generation . . . 53
4.5 The Integrated Model . . . 55
4.5.1 Continuous Query (CQ) Processing Stage . . . 56
4.5.2 Coupling Event and Stream Processing . . . 56
4.5.3 Event Processing Stage that Detects Events . . . 57
4.5.4 Rule processing stage . . . 57
4.6 Future Queries . . . 57
4.7 Summary . . . 58
5. STREAM SERVER EXTENSIONS . . . 59
5.1 Stream Modifiers . . . 59
5.1.1 Non-Windowed Stream Modifiers . . . 62
5.1.2 Windowed Stream Modifiers . . . 62
5.2 XFilter operator . . . 65
5.3 Group By Operator . . . 65
5.4 Summary . . . 67
6. IMPLEMENTATION . . . 68
6.1 Input Processor . . . 69
6.1.1 CEQ . . . 70

6.1.2 Query plan generator 72
6.1.3 Event container 73
6.2 Rule and Event Manager 74
6.3 Query Processor 74
6.3.1 Event Generator Operator and Masks 74
6.4 Stream Modifiers and XFilter operators 80
6.4.1 Instantiator extensions 80
6.4.2 Operator Data 81
6.5 GroupBy Operator: 81
6.5.1 Algorithm 82
6.5.2 Issues in using the JAVA HashTable: 83
6.6 Future Queries 85
6.7 Summary 86
7. CONCLUSION AND FUTURE WORK 87
REFERENCES 89
BIOGRAPHICAL STATEMENT 95

LIST OF FIGURES

Figure		Page
3.1	MavStream Architecture	22
3.2	MavStream Computational Model	26
3.3	LED Architecture	27
3.4	LED:Event Detection Graph Model	33
4.1	EventGenerator Operator	50
4.2	Producer/Consumer buffer approach	52
4.3	Event Generator Operator	53
4.4	Four Stage Integrated Model	55
6.1	EStream Architecture	69
6.2	Input Processor	70
6.3	Events generated with and without masks.	78
6.4	Average action execution latency	78
6.5	Stream Modifiers and XFilter	80
6.6	GHashTable design	85

CHAPTER 1

INTRODUCTION

1.1 EStream Applications

Data intensive applications such as network monitoring, financial applications, enemy intrusion detectors, RFID processing, temperature monitoring, etc., are emerging. They have a continuous, unpredictable and unbounded flow of data, referred to as streams. These applications can be classified as monitoring applications, as they monitor complex conditioned from multiple data streams and combine the events generated by them in order to take some actions. For example, financial applications are interested in monitoring the stock feeds from different feeds for detecting predefined scenarios with respect to share prices to detect an opportunity for trading. Likewise network monitoring applications are interested in monitoring packets generated by the network containing information of the state of the network to detect conditions which may indicate a malfunction or congestion in a telecommunication network and then take corrective measures. There are many other potential applications of EStream, which want to detect complex event compositions consisting of simple events based on conditions computed over stream data.

1.2 Motivation

A significant of research has been done in the field of data stream management systems (or DSMSs) recently. A number of architectures as well as techniques for optimizing stream processing – from scheduling strategies [1, 2, 3] to load shedding [4, 5] – have

been developed. A number of issues have been addressed from architecture [6, 7, 8, 9, 10], recovery, distributed processing to Quality of service [4, 5, 11].

Similarly, event processing has received a lot of attention in the last decade. Analysis of polling and event based (or asynchronous) systems have concluded that event based systems are more appropriate for monitoring applications. A number of techniques and architectures [6, 7, 8, 9, 10] for condition monitoring have been developed. Several event processing languages for specifying composite events have been proposed and triggers have been successfully incorporated into relational databases. Different computational models [12, 13, 14, 15, 16] for event processing such as Petri nets [13, 16], extended automata [17, 18, 19] and event graphs [12, 14, 20] have been proposed and implemented. Various event consumption modes [12, 13, 14, 15, 16] have been explored.

Although both of these approaches are extremely useful in their individual domains, we argue that neither of them makes a complete system for addressing some real world scenarios that have come about due to advances in technology. A large class of non-traditional applications such as process control, threat assessment and analysis, air traffic control, computer integrated manufacturing, and cooperative problem solving often need to react(often subject to timing constraint) to a variety of conditions defined on processed sensor data. Hence there is a critical need for integrating a generic capability for DSMSs which is able to detect events on continuous queries, has a well defined semantics, is efficient and can be tailored to the needs of DSMSs. By integrating an expressive event processing system with stream processing, we argue that the events thus defined for continuous queries are not constrained to the limited event types such as insert, delete and update but can be on any event for which a continuous query can be defined. For example, an attribute value can be monitored from multiple stock feeds and opportunities for program trading can be identified based on the correlation of stock prices within a

sector (e.g., buying shares of an oil company if the share prices of the other oil companies constantly and significantly rise relative to it, over some period of time).

The current DSMSs are passive. They either rely on operating system events [21, 22], or on the user application to continuously check for the conditions (or polling for conditions) on the output stream [23, 24] to become true. Polling wastes resources, which are of crucial concern in sensor applications as sometimes microcomputers, used for processing sensors have limited power resources. It also transfers the onus of determining the frequency of polling to the user or application developer. Polling frequency is likely to be dependent on a variety of parameters such as frequency of update, timeliness (the time window within which the condition needs to be detected) etc., and hence the integration of the event and stream processing will not only increase the application domain of stream processing but also transfer the complexity of monitoring event from the user application to the EStream system.

1.3 Motivating Examples

1.3.1 Car ADN

In a car accident detection and notification system, each expressway 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. 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 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;
```

Similarly speed reduction can also be computed with some extensions in stream processing. 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. Notifications or life saving actions have to be taken once the cars are identified, which is not supported by current stream processing model. 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 JOIN operator can be used to compute it, the number of reports is not minimized.

As illustrated above with an example, 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 and the prototype is developed as part of this thesis.

1.3.2 Network Fault Management

In telecommunication network management, Network Fault Management (NFM) [25] is defined as the set of functions that (a) detect, isolate, and correct malfunctions in a telecommunication network, (b) compensate for environmental changes, and (c) maintain and examine error logs, accept and act on error detection notifications, trace and identify faults, carry out sequence of diagnostic tests, correct faults, report error conditions, and localize and trace faults.

A telecommunication network is a multi-layered system with each layer performing specific NFM functions. Each network element (NE) in the multi-layered network reports the status of each of its components and the status of its environment (e.g., temperature) periodically (e.g., every 5 minutes). These status and alarm messages from each NE, each operation system (OS), and each link are continuously collected in a network operation center (NOC) to be further analyzed by experts to detect and to isolate faults. Once a fault is identified, sequences of actions such as generation of alarms need to be taken locally and remotely.

Currently, for each independent NFM system, due to the large volume of messages continuously reported by each NE and the complex message processing requirements, it is impossible to employ a traditional database management system (DBMS) plus trigger mechanisms as the data processing paradigm for NFM. Current NFM systems have to hard code their data processing logic (or queries) and specific monitoring rules (or ECA rules) in the system. As a result, various filters, pattern languages, regular expressions are employed to find their interesting alarm messages and group those messages into multiple subgroups based on various criteria. These subgroups are finally presented to experts to diagnose root causes or route to an event correlation system to identify causes automatically. Once the causes are identified, a ticket is placed to a trouble ticket system to trace the problem and have corresponding engineers fix it. There are several major

shortcomings of these legacy systems. First, current systems have difficulty adapting to new requirements from their customers because of the hard-coded queries. To add a new query or to add a new monitoring rule, the system has to be reconfigured partially. Second, current systems are very complicated, and their performance is poor because there is almost no query optimization. Third, there is no standard interface or language such as SQL to access those systems, which makes them hard to use and manage. Finally, it is difficult to integrate different NFM systems at different layers because of the hard-coded queries and different implementation techniques.

As there is a dramatic growth in both the volume of message stream and the number of interesting alarms, there is an increasing demand to process and manage message streams for these applications. This motivates us to present a generic solution for NFM by the integration of stream and event processing to process message stream and monitor alarms on the stream computation. The query specification for stream can be extended for the integrated system to define message processing queries along with events or alarms. The various scheduling strategies for stream processing will optimize the queries and improve performance. The same generic integrated solution can applied to all NFMs with different queries defined as per the need of users. It will be simple to integrate such NFMs systems if required at a later time.

1.4 Contributions

In this thesis, we summarize the characteristics and architecture of both threads of work to set the stage for understanding the differences and similarities. We then discuss the issues for the integration of the two computation models and extension to the continuous query definition specification to include event and rule definition capability. We introduce masks in order to extend current stream computation model to reduce the number of uninteresting events generated. We then discuss the alternatives of an

integrated architecture and finally propose the integrated system. We introduce stream modifiers, a class of stream operators for computing changes over stream data. Finally, we discuss other extensions made to the stream processing system.

CHAPTER 2

RELATED WORK

Our work is closely related to the two threads of work, event processing [12, 26, 27, 28, 13, 29, 14, 30, 17, 31, 20, 32] and stream processing [6, 33, 7, 8, 9, 34]. In this chapter we will discuss the systems developed on these two independent streams of works with respect to the limitations and capabilities required for their integration. Then we will discuss the systems which have tried to do a similar integration. To the best of our knowledge there is no such system that does a complete integration hence we will discuss systems which are based on the concept of detecting events over data streams.

2.1 Data Stream Management Systems

Most of the stream processing systems have nearly the same computational model, the data flow model and hence the computation done for processing queries is more or less same. They differ slightly with respect to inputs the systems accept, the type of queries supported, the system architecture and the optimization done for providing the quality of service requirements.

2.1.1 Aurora

Aurora's [35] prime functionality is to process streams based on the configuration set by the application administrator. Aurora is a data flow system and uses the primitive box and arrow representation. Tuples flow from source to destination through the operational boxes. Aurora's query algebra supports seven primitive operations, some of the important ones being select, aggregate, split, union and resample. This architecture sup-

ports continuous queries for real-time processing, views, and ad-hoc queries. It maintains historical storage in order to support ad hoc queries. All these query types are supported using the same set of operational blocks. Input in Aurora is through a GUI. Input begins at the top of the hierarchy and makes use of the zoom capability to further assist in the design. Quality of Service is associated with the output. It is specified in terms of a two-dimensional graph that specifies the output in terms of several performance-related and quality-related services. It is the QoS that determines how resources are allocated to the processing elements along the path of query operation. Aurora has dynamic optimization policies, which change the data flow computation graph (or network) at run time to improve the performance. Optimization is based on the types of queries running in the system. It does not optimize the whole network at once rather it does in parts, by considering a portion of the network at a time. Aurora has a Storage Management module, which takes care of storing all the required tuples for its operation. It is also responsible for queue management. Scheduler is designed to cater the needs of a large-scale system with real time response requirements. Scheduler deals with operators unlike Eddies [36] that deals with tuples for scheduling.

2.1.2 STREAMS

In STREAMS (Stanford Stream Data Management System) [10, 37] A modified version of SQL has been chosen for the query interface. It allows the user to specify sliding window queries in SQL with an explicit referral to timestamps. It assumes that with explicit timestamps, tuples will be delivered in an increasing order. It supports logical and physical windows. Logical windows are expressed in terms of tuples and physical windows are expressed in terms of timestamps. STREAMS [10, 37] support continuous queries but have not addressed the issue of ad-hoc queries. The system generates a query execution plan on the registration of a query that is run continuously. Query execution plan is

nothing but a set of operators connected by queues. Operators make use of synopsis (an internal data structure) to store intermediate results. System memory is distributed dynamically among the synopsis, intermediate queues in query plans, buffers handling incoming data streams and a cache for disk-resident data. Operators in STREAMS adhere to the update and computeAnswer model wherein an operator reads data from its input queue, updates the synopsis structures and writes results to its output queues. Operators are adaptive and take care of the dynamically changing stream characteristics such as stream flow rates, and the number of concurrently running queries. They can produce approximate answers based on the available memory. STREAMS [10, 37] has a central scheduler that has the responsibility for scheduling operators. The scheduler dynamically determines the time quantum of execution for each operator. Period of execution may be based on time, or on number of tuples consumed or produced. Different scheduling policies are being experimented.

2.1.3 The COUGAR Sensor Database System

COUGAR is specifically targeted to meet the requirements of sensor-based applications. The system is based on the characteristics of sensor and their applications. Some of the major challenges facing the COUGAR system are account for the failures of sensor and its communication, uncertainty of sensor data and distributed execution of query without global knowledge of the sensor network. COUGAR focuses on a distributed approach toward query processing wherein the workload determines the data that needs to be extracted from the sensors. COUGAR is based on the Cornell university's PREDATOR [23, 24] object relational database system. Sensor data is considered as a combination of stored data and sensor data. Stored data is represented as relations and sensor data is represented as time series data based on a sequence model. Long running sensor queries are supported by this system. Sensor queries are defined as an acyclic

graph of sequence and relational operators. In COUGAR , signal-processing functions are represented as Abstract Data Type (ADT) functions. Sensor ADT's are defined for sensors of the same type (e.g. temperature sensor, seismic sensor, etc). Public interface to an ADT corresponds to the signal processing function supported by a type of sensor. Sensor queries are SQL like queries with a little modification wherein ADT can be included in the SELECT or WHERE clause of the query. Query processing takes place on a database front end whereas the signal-processing functions are executed on the sensor nodes involved in the query. On each sensor a lightweight query execution engine is responsible for executing signal processing functions and sending data back to the front end. COUGAR assumes that there are no modifications to the stored data during query execution which is guaranteed using Two Phase locking. COUGAR also mentions the need of event integration as a topic of research to be addressed but does not provide any details on it. In order to support monitoring event based queries they provide long-running queries which polls the sensor data after the periodicity defined in the query.

Consider the following example query in COUGAR:

```
  SELECT   AVG(R.Concentration)
    FROM   ChemicalSensor R
   WHERE   R.loc IN region
  HAVING   AVG(R.concentration) > T
DURATION   (now, now+3600)
   EVERY   10;
```

This query checks for the concentration of the chemical every 20 seconds and outputs the tuples, which have average concentration above a threshold. There is a delay of 10 seconds for the query and as a result, to provide real time response we need an event which will trigger the moment the concentration goes above the threshold.

2.1.4 Fjord: Architecture for Queries Over Streaming Senor

Fjord [9] is sensor data processing architecture for data intensive sensor-based applications. It provides a low-level database engine support required for sensor centric data-intensive systems. The main focus of the system is to provide an efficient, adaptive and power sensitive infrastructure. This system supports the Berkeley Highway lab to monitor traffic conditions with the help of sensors that are deployed on Bay Area freeways. Fjord's operators export an iterator like interface and are connected together via local piper or wide area queues. It provides support for integrating streaming data that is pushed into the system with disk-based data that is pulled into the system. Each machine involved in the query runs a single controller in its own thread. Controller accepts message to instantiate operators, connect local operators via queues to other operators that may be running locally or remotely. Queues also export an iterator like interface irrespective of whether the operators are local or remote. This way it makes the operator ignorant of the nature of their connection to remote machine. Each query has its own thread, which is multiplexed between local operators via procedure calls in case of a pull base architecture or via a special scheduler that also controls the input and output of data through the operators. Operators are the primary functional unit of the system. Each operator owns a set of input and output queues. It reads data from the input queue, performs the required operations and directs them to the output queue. No processing takes place in the queues. Stale data are discarded from queues based on the requirements of the applications. It supports non-blocking operators such as selection and projection and blocking operators such as join and aggregate. A main memory symmetric hash join has been implemented which maintains a hash table for each relation. Window based operations are supported for blocking operators. Considering the nature of streams, Fjord provides an optimization by combining multiple queries. This way a significant amount of computation and memory can be shared thereby improving the overall performance

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

2.2 Event Processing Systems

The event processing systems, on the other hand have different computational models such as the Event Detection Graph (EDG), Petri Nets and Finite Automaton. All the computational models have certain restrictions and capabilities. In this section a system based on each event computational model is discussed with respect to its architecture and the capability to be integrated with a stream processing system in a manner that it does not significantly affect the QoS of stream processing, which are tuple latency and memory utilization.

2.2.1 ODE

Ode [38, 17] is a database system and environment based on the object paradigm. The database is defined, queried and manipulated using the database programming language O++, which is an upward compatible version of C++. Ode provides active behavior by the incorporation of constraints and triggers [38, 17]. Constraints and triggers are defined declaratively within a class definition and consist of a condition and action. Constraints are used for maintaining object consistency and are applicable to all instances of the class in which they are declared. Triggers, on the other hand, are used for other purposes and are applicable only to those instances of the class in which they

are declared. The computational model for ODE is an extended finite automaton for composite event detection and triggering of constraints and triggers. The extended automaton makes a transition at the occurrence of each event in the history like a regular automaton and in addition handles attributes of the events to compute a set of relations at the transition. The drawback of finite automata as the computational model is that for each event occurrence each constraint and trigger has to be evaluated, i.e., each finite automaton constructed has to be checked to see if there are any transitions. In case of the event and stream integrated architecture where each stream tuple is converted to an event, it leads to excessive checking, which introduces delay in detecting the event and hence is not suited as it will not optimize tuple latency, an essential requirement of stream processing. In this implementation a suite of finite automatons are generated if an attribute is specified, for each unique value of the attribute. For detection of such an event all the automatons should be satisfied. This further increases the tuple latency and also memory utilized. There is also no specification of priority in the case of constraints and triggers and they seem to be activated in an arbitrary manner due to which events with priority cannot be defined.

2.2.2 SAMOS

The combination of active and object-oriented characteristics within one, coherent system is the overall goal of SAMOS (Swiss Active Mechanism Based Object- Oriented Database System). Samos [39] addresses event specification and detection in the context of active databases. The computational model for SAMOS is modified colored Petri Nets called SAMOS Petri Nets to allow flow of information about the event parameters in addition to the occurrence of an event. Memory utilization for SAMOS is increased in case of primitive events participating in more than one composite events, (e.g., in E=(E1;E2) and in EE=(E1,E3)). To represent the Petri Nets for the two composite events, E1 has

to be duplicated. The duplication of Petri Nets equal to the number of common event expressions that E1 participates in. Since all duplicates must also be represented in the data structure this might lead to excessive storage requirements thus increasing memory utilization. In Samos only the chronicle context is supported. To implement contexts a different Petri net has to be generated. If contexts are introduced in Petri Nets then they cannot be built unless the context information is specified beforehand. Petri Nets also do not support explicit and temporal events. The limited capability of event detection and excessive memory utilization are the drawbacks of integrating this system with the stream system.

2.2.3 Sentinel or LED

Sentinel [40, 41, 42, 43, 44] is an integrated active DBMS incorporating ECA rules using the Open OODB Toolkit from Texas Instruments. Event and rule specifications are seamlessly incorporated into the C++ language. The computation model for Sentinel is an Event Detection Graph (EDG). Any method of an object class is a potential primitive event. The event occurs either at the beginning of the method or at the end of the method. Composite events are defined by applying a set of operators to primitive events and/or composite events. Events and rules are specified in a class definition. In addition, Sentinel supports events and rules that are applicable to a specific object instance alone. In that case, events and rules are specified within the program where the instance variables are declared. This ability to declare events and rules outside a class allows composition of events across classes. It can be recalled that a significant drawback of Ode was that a composite event could only be composed of events within the same class but not from different classes. This is because Ode does not support event definitions outside the class. Sentinel overcomes this drawback by allowing event definitions outside the classes too. The parameters of a primitive event correspond to the parameters of the method

declared as the primitive event and other attributes, such as the time of occurrence of the event. The processing of a composite event involves not only its detection, but also the computation of the parameters associated with the composite event. The parameters of the event (primitive or composite) are passed onto condition and action portions of a rule thus avoiding the drawback of ODE, which requires the creation of a separate automaton for the different attributes. The parameters associated with the detection of an event can be different in different contexts. Sentinel supports all the four parameter contexts specified in HiPAC namely, recent, chronicle, continuous and cumulative contexts. An event can trigger several rules, and rule actions may raise events that can trigger other rules. Sentinel supports multiple rule executions, nested rules execution as well as prioritized rule executions. Out of the three coupling modes (immediate, deferred and detached) specified in HiPAC, Sentinel currently supports immediate and deferred modes of rule execution.

Such implementation which uses an event detection graph which is similar to the data flow model of stream processing is ideal for the integration. It also minimizes space by sharing the same event node in the graph for creating multiple composite events thus eliminating the drawback of SAMOS. Events can be detected in multiple contexts using the same graph representation. The integration of an event and a stream processing is best suited with such a system.

2.3 Event and stream integration systems

2.3.1 HiFi

HiFi generates simple events out of receptor data at its Edges [45, 46] and provides the functionality of complex event processing on these Edges [45, 46]. It addresses the issue for generating the simple events by Virtual Devices [45], which interact with the

heterogeneous sensors to produce application level simple events. Then complex event processing can be done on these simple events to correlate into a sophisticated application level event. An application of this system to a library scenario is also described.

Although this system is a step in the right direction for the detection of events over sensor data, it does not define and detect events over stream queries. The events detected at Edges are simple events and cannot be defined over the result of the data preprocessed by a Continuous Query [47]. Example of events that could be detected by this system are simple events such as

1. "The book with ID 10 is on the shelf"
2. "The person with ID 7 is leaving the library"
3. "The book with ID 4 is being checked out".

2.3.2 Tiny DB

TinyDB has Event Based Queries [21, 22], which is processing of events over stream queries. They address the need for event processing over stream processing is essential. The aim of implementing event here was to save power of the microprocessors as it is a push based mechanism which saves the process of constantly polling for events. The events are initiated by low-level lying operating system events. Events are interrupt lines, which are raised on the processor or sensor readings going above or below some threshold. Events have to be explicitly defined and then registered with the Query.

This research is a step towards providing event capability for stream processing but does not integrate a complete event processing system. It lacks the capability of complex event processing and rule specification.

2.3.3 Financial applications

Financial applications like streambase [48], Apama [49], GemStone [50], etc., are systems which are used for mining for patters in data streams and raising events when some financial scenarios are detected. They also provide the capability of complex event processing over simple events. These systems are developed for a specific application domain and do not allow the user to evaluate stream queries on sensor data using the database operators. It provides a dashboard to the user to define financial scenarios to be mined in the input streams. It detects patterns and process events but does not span detection of events over events detected for other queries.

2.4 Summary

In this chapter we have discussed the various systems for stream processing. All the systems follow the same data flow model, where the data flows through the operators and may differ slightly with respect to the optimization done for stream processing and the way the query is posed to the system. The event processing on the other hand has various computational models such as Petri Nets, EDG and finite automaton and significantly differ with respect to the way the events are detected. Complex event detection on same primitive events may also require duplication of the computational model with respect to Petri Nets and finite automaton. For checking of conditions over the event attributes in the finite automaton model a separate automaton had to be constructed which increases space and time required to detect the event, which are both crucial for stream processing applications. Various modes for rule processing like immediate and chronicle are also not supported by all the models. In the computational model for Sentinel, the event detection graph is very similar to the data flow model and it supports the stream requirements for optimizing memory and time better than the other systems. Later, we overview some

systems which do event processing on sensor data. All these systems lack a complete stream capability and they simply do event processing on sensor data. TinyDB on the other hand does not implement a complete event processing capability to detect complex events and process rules.

CHAPTER 3

THE MAVSTREAM AND THE LOCAL EVENT DETECTOR (LED)

The MavStream, a data stream processing system and the LED, an event processing system are both homegrown systems and implemented in Java. In this chapter we will explain the two systems with respect to their architecture, functionality and the computational model, to lay the background before we address the design issues for the integration of these two systems in the next chapter. We also analyze the relationship between the generic stream processing data flow model on which MavStream is based and the Event Detection Graph (EDG) model, on which LED is based.

3.1 MavStream

MavStream is being developed for processing continuous queries over streams. MavStream is modeled as a client-server architecture in which client accepts input from the user, transforms it into a form understood by a server and sends the processed input to the server based on predefined protocols. MavStream is a complete system wherein a query, submitted by the user, is processed at the server and the output is returned back to the application. The various components are shown in Figure 3.1.

The MavStream server, upon receiving the query from the client, constructs the query plan object for that query, instantiates and executes the query and returns the outputs of the query to the application.The output is sent to the Run-Time Optimizer also. The Run-Time Optimizer frequently checks the QoS requirements for the query with respect to the actual output. If the QoS Specifications are not satisfied then the query takes measures to bring the output in line with the specified QoS. This is done

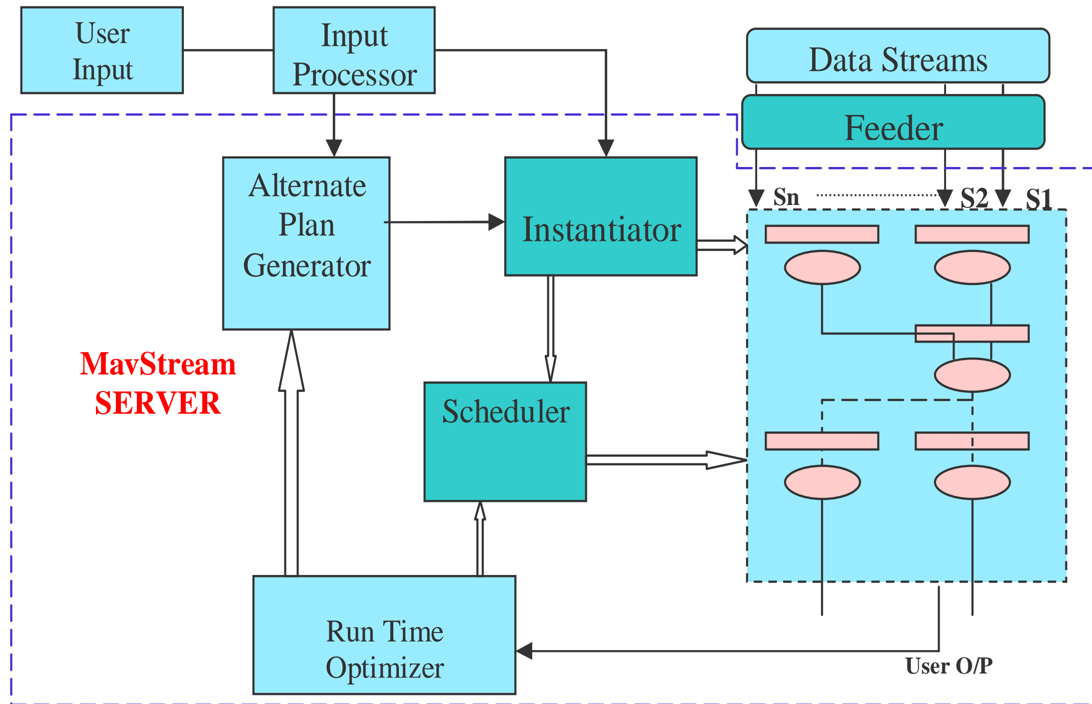

Figure 3.1 MavStream Architecture

by either changing the scheduling policy or by load shedding. It is also possible to generate an alternate plan to satisfy the QoS Specifications. The client provides graphical user interface to pose queries to the system. It sends the complete query along with the QoS requirement from user specifications to the server over a predefined protocol. Communication between client and server is command driven and protocol oriented.

3.1.1 Operators

Query processing in a traditional DBMS is not designed to produce real-time response to queries over high volume, continuous, and time varying data streams. The processing requirements of real time data streams are different from traditional applications and demand a re-examination of the design of conventional operators for handling

long running queries to produce results continuously and incrementally. Blocking operators (an operator is said to be blocking if it cannot produce output unless all the input is available) such as aggregates and joins may block forever on their input as streams are potentially unbounded. A window concept has been introduced to convert the computation of blocking operator into a non-blocking one. Tuples are processed by these blocking operators till a window is elapsed and output is produced. Operators have one or sometimes more input and one output queue. The operators are implemented as individual threads which are scheduled by the scheduler to optimize QoS requirements. The operators are suspended by the scheduler thread when the time quantum is elapsed or when they do not have any more tuples to process.

3.1.2 Instantiator

Instantiator has the responsibility of initializing and instantiating streaming operators and their associated buffers on accepting user queries from the client. Client's query is converted into a plan object, which is a sequence of operator nodes represented as a operator tree, where every node describes an operator completely. The query plan object is used as input by the instantiator. Instantiator traverses the plan object in a bottom-up fashion and instantiates all the operators and associates buffers between them. Every operator is an independent entity and expects predicate condition in a predefined form. Instantiator populates the operator instances with the predicate conditions defined in the query plan object. It also associates a scheduler with the operator to facilitate communication for scheduling. Instantiator does not start the operator and it only does the necessary initialization.

3.1.3 Scheduler

Scheduling is done at an operator level since it is not meaningful to schedule at tuple level as the number of tuples in entering the system is potentially unbounded and tuple-level granularity results in too much of overhead. Scheduler schedules operators based on its state and priority. The optimization of the query depends upon the scheduling strategy selected by the scheduler. Following are the scheduling strategies supported in MavStream:

1. Round-Robin: In this scheduling strategy, all the operators are assigned the same priority (time quantum). Scheduling order is decided by the ready queue. This policy is not likely to dynamically adapt to quality of service requirements as all operators have the same priority. However, there is no starvation in this approach.
2. Weighted round-robin: Here different time quanta are assigned to different operators based on their requirements. Operators are scheduled in round robin manner but some operators may get more time quantum over others. For example, operators at leaf nodes can be given more priority as they are close to data sources. Similarly, Join operator, which is more complex and time consuming, can be given higher priority than non-leaf Select.
3. Path capacity scheduling: Schedules the operator path which has the maximum processing capacity as long as there are tuples present in the base buffer of the operator path or there exists another operator path which has greater processing capacity than the presently scheduled operator path. This strategy is good for attaining the optimum tuple latency.
4. Segment scheduling: Schedule the segment which has the maximum memory release capacity as long as there are tuples present in the base buffer of the segment or there exists another segment which has greater memory release capacity than

the presently scheduled segment. This strategy is good for attaining less memory utilization.

5. Simplified segment scheduling: It uses the same segment strategy but the segment construction algorithm is different. Instead of breaking operator path into many segments, we break the operator path into only two segments. This strategy takes slightly more memory than the segment strategy resulting in reduced tuple latency.

3.1.4 MavStream Server

MavStream server is a TCP Server which listens on a chosen port. It is responsible for executing user query, converting a plan object into a query instantiation and processing it to give the desired output. It provides integration and interaction of various modules such as Instantiator, operators, buffer manager and scheduler for efficiently producing the correct output. It provides details of available streams and schema definitions to clients so that they can pose relevant queries to the system. It also allows new streams to be registered with the system. It initializes and instantiates operators constituting a query and schedules them. It also stops a query, which in turn stops all operators associated with the query on receiving command for query termination. Some of the commands supported by the server are given below:

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

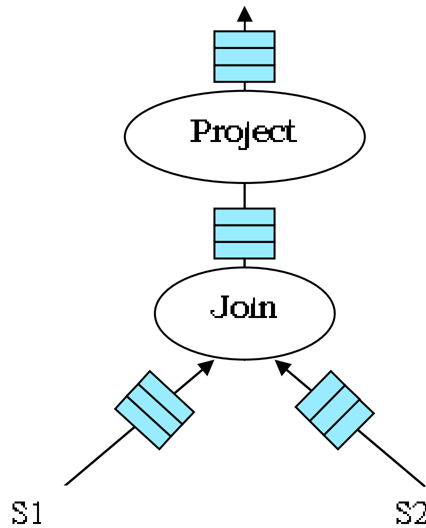

Figure 3.2 MavStream Computational Model

3.1.5 Data Flow computational model

This model is generic for stream processing with slight variations from one system to another. As suggested by the name, the data flows through the system from one operator to another until the entire query is processed. Each operator uses both the pull mechanism to retrieve tuples from its input queue(s). We will explain this model with the following example.

```
SELECT * from S1,S2,
WHERE S1.Attribute1=S2.Attribute1;
```

In figure 3.2 the tuples from streams S1 and S2 are sequentially ordered (either by timestamp or sequence number). The tuples flow upwards in the Query tree and the result is produced in the output buffer. The intermediate buffers hold the tuples till they are consumed, which is similar to the buffers of Aurora [35]. In MavStream we do not support snapshot queries. Some systems like [9, 35] may support snapshot queries by maintaining a synopsis at the operators or the buffers. All the operators execute as individual threads and have to be scheduled only when tuples are present in their

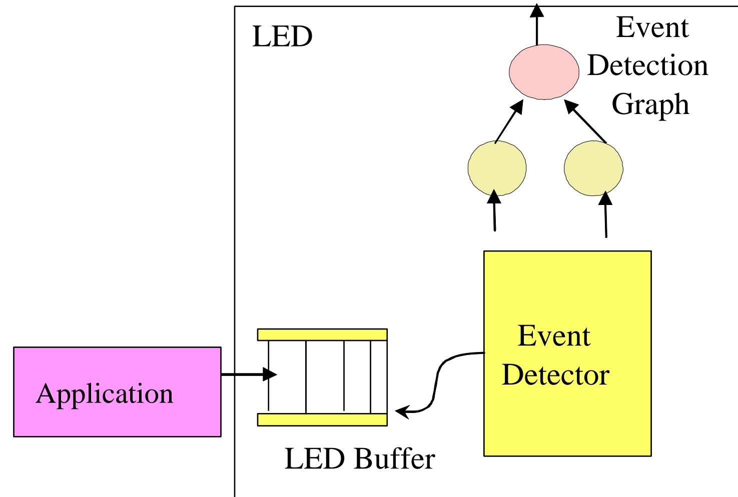

Figure 3.3 LED Architecture

input queue. Various scheduling strategies [1] such as Chain scheduling [51], Segment scheduling [52, 1], path capacity scheduling [52, 1], etc., have been studied. The queries are scheduled in a way to optimize the memory utilization (since it is a main memory system) and/or tuple latency (since it supports real time response).

3.2 Local Event Detector

The Java LED is based on ECA (event-condition-action) rule paradigm. A rule consists of:

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.
3. Action: specifies the operations that are to be performed when the event occurs and the corresponding conditions evaluate to true.

The major components of LED are shown in the figure 3.3:

The application explicitly defines the events using the APIs provided by the ECAAgent. The EDG is constructed for the user defined events. When the events are raised by the applications, an event object is created and placed in the LED buffer. LED is a *single threaded* application. It consumes one event object at a time from the notify buffer and raises the primitive events defined on it. The primitive events detected are propagated upwards in to EDG to detect complex events defined on them. The event detector thread also executes all the rules defined on the event that are raised and only then returns to the LED buffer to consume more events. Event Detection is based on *time based semantics*. When simple events are detected a timestamp is attached to them. The complex event detection is implemented on the timestamps of the constituent events.

3.2.1 Event types supported by LED

1. **Primitive Events:** An event is an occurrence of interest at a specific point in time. Primitive events are the elementary occurrences and are classified into domain-specific, explicit and temporal events. Domain-specific events are specific to application domain. For example, a database domain-specific event may occur when a tuple is updated, inserted or deleted, in object-oriented systems events can be related to methods and each invocation or method call can be an event occurrence. Explicit events are explicitly defined by an application and raised by the user. The parameters of the explicit event are also specified by the user. Temporal events correspond to absolute and relative temporal events. The absolute temporal event is an event associated with an absolute value of time. For example, 11 A.M. on October 28, 2005 is an absolute event. The relative temporal event corresponds to a specific point on the time line, which is an offset from another time point (specified either as absolute or as an event).

2. **Composite Events:** A composite event is an event that is composed of primitive events and/or other composite events by applying Snoop [53, 54] event operators such as OR, AND, SEQUENCE, NOT, etc. In other words, the constituent events of a composite event can be primitive events and/or previously defined composite events.

   Event Operators: The event operators are used to construct composite events. Some of these event operators and their semantics are described briefly in the following section. The upper case letter E, which represents an event type, is a function from the time domain on the Boolean values. The function is given by E (t) =True if an event type E occurs at time point t, False otherwise

   - **OR (V):** Disjunction of two events E1 and E2 denoted by E1 V E2 occurs when either E1 occurs or E2 occurs. Formally,(E1 V E2) (t) = E1(t) V E2(t)
   - **Conjunction AND (Λ):** Conjunction of two events E1 and E2, denoted by E1 Λ E2 is applied when E1 and E2 both occurs (in any arbitrary order). Formally, (E1 Λ E2)(t) = (E1(t1) Λ E2 (t)) or ((E1(t) Λ E2(t1)) and t1 $\geq$ t.
   - **Sequence (<<):** The sequence of two events E1 and E2, denoted by E1 << E 2 occurs when E1 happens before E2. That is, the timestamp of occurrence of E1 is less than the timestamp of occurrence of E2. Formally,(E1 << E2)(t) = E1 (t1) << E2 (t) and t1<t
   - **NOT (¬):** The not operator, denoted by ¬(E2)[E1, E3] detects the non-occurrence of the event E2 in the closed interval formed by E1 and E3. ¬(E2)[E1, E3](t) = (E1(t1) Λ E2(t2) Λ E3(t)) and t1 $\leq$ t2 $\leq$ t
   - **Aperiodic Event Operators (A, A*):** The Aperiodic operator A is used to express the occurrence of an aperiodic event in the half-open interval formed by E1 and E3. There are two variants of this event specification. The non-cumulative variant of an aperiodic event is expressed as A(E1, E2, E3) where

E1, E2 and E3 are arbitrary events. The event A is signaled each time E2 occurs during the half-open interval defined by E1 and E3. A can occur zero or more times. Formally, A (E1, E2, E3) (t) = E1(t1) Λ E2(t2) Λ E3(t)) and ($t1 < t2 \leq t$ or $t1 \leq t2 < t$) There are situations when a given event is signaled more than once during a given interval (e.g., within a transaction), but rather than detecting the event and firing the rule every time the event occurs, the rule has to be fired only once. To meet this requirement, there is an operator A* (E1, E2, E3) that occurs only once when E3 occurs and accumulates the occurrences of E2 in the half-open interval formed by E1 and E3. This constructor is useful for integrity checking in databases and for collecting parameters of an event over an interval for computing aggregates. For example, the highest or lowest stock price can be computed over an interval using this operator. Formally,

A*(E1, E2, E3) (t) = (E1 (t1) Λ E3 (t)) and $t1 < t$ In this formulation, E2 is not included because the occurrence of the composite event A* which coincides with the occurrence of E3 is not constrained by the occurrence of E2. However, the parameters of A will contain the parameters of E2.

- **Periodic Event Operators (P, P*):** The periodic operator, denoted by P (E1, [t], E3) is used to express a periodic event that repeats itself within a constant and finite amount of time. The event P is signaled for every amount of time t in the half-open interval (E1, E3]. Formally, P (E1, [TI], E3) (t) = (E1(t1) Λ E3(t2)) and $t1 < t2$ and $t1 + x * TI = t$ for some $0 < x < t$ and $t2 \leq t$ where TI is a time specification.

3.2.2 Parameter Context

Four parameter contexts - recent, chronicle, continuous and cumulative, were introduced to provide a mechanism for capturing meaningful application semantics and reduce the space and computation overhead for the detection of using the semantics described above. The contexts are defined by using the notions of initiator and terminator for events. An event that initiates the occurrence of a composite event is termed the initiator of the composite event. An event that completes the detection of a composite event is denoted as the terminator of the composite event. For example, a composite event (E1; E2; E3) has E1 as initiator and E3 as terminator.

1. **Recent:** In the recent context, the most recent occurrence of the initiator (when there are multiple instances of the same event) for the detection of event is used. When the event occurs, the event is detected and all the occurrences of events that cannot be initiators of that event in the future are deleted. In this context, not all occurrences of a constituent event will be used in detecting a composite event. Furthermore, an initiator of an event will continue to initiate new event occurrences until a new initiator occurs.
2. **Chronicle:** In the chronicle context, the initiator-terminator pair is unique for an event occurrence. The oldest initiator is paired with the oldest terminator for each event. When an event occurs, the occurrences of the events used for detecting it are deleted. The event occurrence can be used at most once for computing the parameters of the composite event.
3. **Continuous:** In the continuous context, each initiator of an event starts a separate detection of that event. A terminator event occurrence may detect one or more occurrences of the same event. The initiator and terminator are discarded after an event is detected.

4. **Cumulative:** In the cumulative context, all occurrences of an event type are accumulated as instances of that event until the event is detected. When the event occurs, all the occurrences that are used for detecting that event detection are discarded.

3.2.3 Rule Priority

In addition to the parameter context, there is also a priority assigned to each rule. The default priority of a rule is a priority of 1. The priority increases with the increase in the numerical value. That is, 2 has a higher priority than 1, 3 is a higher priority than 2 and so on. Rules of the same priority are executed concurrently and rules of a higher priority are always executed before rules of a lower priority. It is possible that a rule raises events that in turn could fire more rules. This results in a cascaded rule execution. Furthermore, rules can be specified either in the immediate coupling mode or the deferred coupling mode. Both priority and coupling mode of a rule have to be taken into account for scheduling the rule for execution.

3.2.4 LED Computational Model

LED uses an event graph for detecting composite events as shown in Figure 3.4. Each node in the event graph represents either a primitive event or a composite event. Primitive event nodes are leaf nodes from which composite event nodes are constructed. The primitive event node contains an instance-based multiple rule list and an event subscriber list, while the composite event node contains only one rule subscriber list and one event subscriber list. An instance-rule list is a collection of rule subscriber lists for classes and instances. The rule subscriber list and event subscriber list keep the associated rules and composite events, respectively. The event signature HashTable provides references to the primitive event nodes. When a primitive event occurs, it is

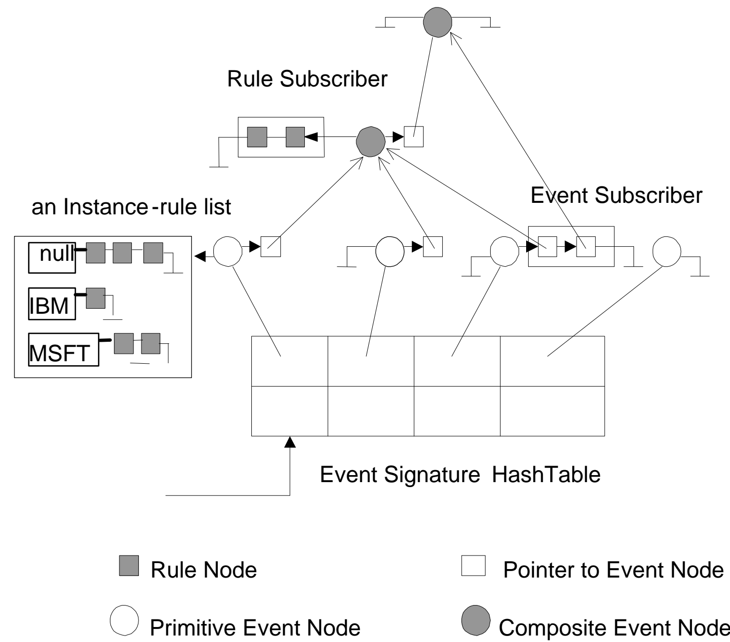

Figure 3.4 LED:Event Detection Graph Model

notified to the leaf node and the occurrence is propagated to the internal nodes similar to a data-flow computation.

3.3 Data Flow Model Vs. Event Detection Graph

In this section we will discuss the two computational models and the reasons for choosing these models for the integration of event and stream processing.

3.3.1 Inputs and Outputs

Inputs (or data sources) to an event processing model are a sequence of events (or event histories) ordered by their time of occurrence. Most event sequences considered by event processing models are generated by system clock, etc. The input rate of an event sequence is not assumed to be very high or even bursty. The outputs of event operators form event sequences, which are ordered by their occurrence timestamps. On

the other hand, inputs to the data stream applications are data streams. Input tuples in a data stream can be ordered by an attribute and not necessarily by a timestamp (e.g., sequence id of a TCP packet in a TCP packet stream) as in the case of event sequences. 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.

3.3.2 Event Operators and Stream Operators

Event operators are not quite different from the operators supported by current stream processing models. Event operators are mainly used to express and define the computation on events (or event objects) and to reduce the number of output events through consumption modes, and they solely use the timestamp of an event for detecting composite events. The attributes of the event are passed as Parameters but they are only used in the computation of rules (checking for condition and action part of a rule). On the other hand, current stream processing operators are mostly modified relational operators, which focus on how to express and define the computation on the tuple attributes.

3.3.3 QoS Support

The notion of QoS is not present in the event processing literature. Although, there is some work on real-time events and event showers, 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. Thus, events are detected based on the best-effort method. The EDG is based on operators and compute sets of histories rather than instances (as in case of Petri Nets and extended automata) and hence the same operator node could be reused for defining multiple complex events

instead of creating a separate node if the operator participates in detecting more than one composite nodes. This model does the checking of the conditions on event attributes in the rules which can be defined separately for an event node hence for taking multiple different actions on different values, the same event node could be reused unlike finite automaton. This reusability of operator nodes saves both time and memory than the other event processing models and supports the QoS requirements of Stream processing.

3.3.4 Reusability

The EDG and data flow model are exactly the same, except that EDG works on different set of assumptions (timestamps and non-burst inputs). The EDG model supported the domain where the rate at which events generated were limited, so whenever a primitive event is detected it is propagated higher up in the EDG and only when all events dependent on it are detected, the next primitive event raised will be detected by the event detector. In stream processing, the number of events generated are large so to seamlessly integrate event and stream processing, this delay introduced between detection of two primitive events can be avoided by pipelining the detection of events at each level of the EDG by introducing buffers/queues between event nodes. The same buffer/queue that is used in the data flow model for stream processing can be reused for this purpose and also the scheduling and load shedding strategies can be reused to optimize the processing of EDG when it is pipelined.

3.4 Summary

In this chapter, we have discussed the architecture of MavStream and its various components. We have also discussed the main components of the Local Event Detector. We have explained the computational model of both the systems, which follow the same graph design with stream tuples and events propagating from the leaf to the root nodes for

each of the models, respectively. We also analyzed the relationship between the generic stream processing data flow model and the Event Detection Graph (EDG) model, on which these systems are implemented. Both the computational models complement each other. The inputs and outputs of both the models are ordered. Although operators of event processing currently perform on timestamps, yet can easily be extended to attribute based semantics, similar to stream operators. Extensions to event side such as pipelining of event detection in EDG can be easily done by reusing the components of stream side such as scheduler, load shedder, buffer management, etc. In the next chapter we present the design of the integration of the two systems.

CHAPTER 4

DESIGN

In this chapter we discuss the design of the EStream system. We first highlight the issues for generation of event for stream tuples followed by the discussion of extensions to the continuous query specification to include ECA (event, condition and action) rules. We further discuss a concept of attribute based constraints on generation of events. Finally, we discuss the approaches for the integrated architecture and elaborate on our design.

Consider the car ADN example illustrated in Section 1.3, which we use to explain the issues that arise for integrating the stream and the event processing systems. The accidents will be detected when the conditions for *i)* IMMOBILITY, *ii)* SPEED REDUCTION, and *iii)* SAME SEGMENT are satisfied. We have already seen that condition 1 can be expressed as a continuous query on the stream side. The continuous query that detects whether the car has been stopped can be represented as

Immobile:

```
SELECT  carId, Xpos, count(*)
FROM    CarLocStr [2min sliding window]
WHERE   true
GROUP BY carId, Xpos
HAVING  count(*) > 3;
```

This query will group the tuples based on the carId and position (Xpos). Since cars produce tuples every 30 sec, this query will count the number of times for which a car is in the same position for at least three consecutive tuples or 90 seconds.

Condition 2 can also be represented in the stream side if the stream side has some operator in addition to the RDBMS operators, which can evaluate complex conditions on stream attributes. We have introduced XFILTER which is similar to SELECT and can not only evaluate conditions with syntax: *Attribute Operator Constant*, but can also evaluate conditions such as *Attribute1 Operator Attribute2* on a single tuple. The full description of this operator is given in the next chapter. The query which will calculate whether the car has reduced its speed by more than 30 percent can be represented in SQL as:

Decrease

```
SELECT CarId ,Xpos, MIN(speed) as min_Speed, MAX(Speed) as max_Speed
FROM CarLocStr [2min sliding window]
WHERE True
GROUP BY CarID;
HAVING min_Speed < (.7*max_Speed);
```

In our implementation GROUP BY operator is implemented without the HAVING condition. HAVING can be evaluated as a separate SELECT or XFILTER operator, which evaluates on the output tuples produced by the GROUP BY operator and does filtering on the attribute-based conditions which are specified in the HAVING clause. The above query can be represented by a GROUP BY and an XFILTER operator with the XFILTER operator consuming the tuples produced by the GROUP BY and filtering tuples based on the condition given in the HAVING clause.

The SPEED REDUCTION is calculated by computing minimum and maximum speeds of each car in a time window of 2 minutes and applying the condition that each car's minimum speed is less than 70 percent of the maximum speed. In real world this implies that a car following the car that has stopped, may have come to a halt or has made a lane change to pass through.

Detection of condition 3 need be done in the event side by implementing a composite event with a sequence operator semantics. This operator will only detect the event when condition 2 happens after condition 1. Once a car is detected to have lowered its speed after another car that has stopped, an event gets triggered and in the rule associated with the event, condition 3 can be checked (i.e., whether both cars are in the same segment or not). This complex query requires computation at both event and stream side and in order to design a system that will be able to compute this query, several issues have to be addressed. Some of the issues are:

1. Whether an event shall be generated for each tuple or for a group of tuples?
2. How will the system raise the right event for a stream tuple when there are multiple queries executing in the stream side and multiple events created in the event side?
3. How shall the two models be integrated?
4. Should both systems be in the same address space or different address space?

4.1 Issues

All issues raised for the integration of the stream and event processing will be addressed in this section. We then discuss the approaches and design of our integrated system.

4.1.1 Event as a Single Tuple or a Collection of Tuples

Each generated event has a timestamp attached with it. Operators such as SEQUENCE, NOT etc., which correlate events, are implemented on the event timestamp. Similarly, each stream tuple also has a timestamp or a sequence ID associated with it and stream operators such as JOIN, AGGREGATE etc., use this attribute to evaluate results and implement the various window concepts. The output stream tuples are ordered on timestamp or sequence ID. If the stream processing has to be extended with

event processing then timestamp information for each tuple should be maintained. In case we detect a single event for a group of tuples then the timing of each individual tuple will be lost and each tuple will be assigned the timestamp of the group. The event operators will not be able to detect complex events such as SEQUENCE, NOT at the tuple granularity; hence it is essential to detect each tuple as an event. In the Car ADN example, if we detect an event for a group of cars that have stopped (Immobile) and a group of cars with decreased speeds (Decrease) then we will not be able to detect if a car in one group decreases its speed after another car in the other group that has stopped. Instead, we will only be able to detect when a group of cars has decreased speed after a group of cars that has stopped.

4.1.2 Event and Continuous Query Definition and Mapping

There should be a mapping of which event to raise for each stream tuple as there may be more than one event defined and more than one CQs executing. The mapping should uniquely identify which tuple shall raise which event on the event side. The query and the event associated with it should be uniquely identified. In our event model, each event can be uniquely identified by an event name. If we add a unique identifier to each query then there can be a mapping of the query identifier to the event name which will uniquely identify the events that need to be notified when the CQ produces an output tuple. This mapping should persist and should be visible for event generation.

4.1.3 How shall the two models be integrated

Integration of the stream and event computational models is possible if the output produced by stream can be consumed by the EDG as input. This can be addressed by calling APIs of event side for creation of event objects for stream tuples whenever events need to be generated on them. Event objects thus created can be enqueued in the LED

buffer for their detection in the EDG. The attributes and values of stream tuples should also be passed to the event side as rule conditions have to be evaluated on them. API calls can also be made for insertion of stream attribute values in the event object at the time of its creation. This is done only for primitive events as the attributes and values for composite events are automatically obtained from the constituent primitive events.

4.1.4 Address Space Issue

Communication between event and stream sides is required for the integration of the computational models. Event APIs need to be called for *i)* Creation of EDG for events defined on the CQs, *ii)* generation of event objects for output tuples of CQs, and *iii)* passing of stream attributes and values as event attributes and values.

If both the systems are in separate address spaces then some IPC (Inter Process Communication) is required for calling the APIs and marshalling and unmarshalling will be needed for passing of attribute values of stream tuples to the event side. If tuples produced by stream processing on which events have to be generated are large then overhead introduced by IPC will be significant, which will lead the system to slow down. Due to this reason we run both systems in the same address space.

4.2 Continuous Event Query (or CEQ)

We need to have a way to specify continuous queries, a way to specify events and rules, and a combination there of. The user should also have the privilege of defining both the CQ and the events together. Such queries which have both event and stream definition are referred in this thesis as Continuous Event Queries (or CEQs). If CEQ allows for the independent specification of events and continuous queries and provide a mechanism to associate events with pre-defined continuous queries then we can provide maximum flexibility. This way the CEQ specification can be used for *i)* defining CQs and

associated events along with their mapping, *ii)* defining only events (both primitive and composite) and their mapping with predefined CQs, and *iii)* defining only CQs without event specifications.In CEQ specifications the query name is given as the query identifier and association of the query name and event name is given as the mapping. The car ADN example can be represented by a generic CEQ specification as;

```
CREATE CQ Immobile AS
  SELECT carId, Xpos, count(*)
  FROM CarLocStr [2min sliding window]
  WHERE true
  GROUP BY carId, Xpos
  HAVING COUNT(*) > 3;

CREATE CQ Decrease AS
  SELECT CarId ,Xpos, MIN(speed) as min_Speed, Max(Speed) as max_Speed
  FROM CarLocStr [2min sliding window]
  WHERE True
  GROUP BY CarID;
  HAVING min_Speed < (.7*max_Speed);

CREATE  PRIMITIVE  EVENT EImmobile on Immobile
CREATE  PRIMITIVE  EVENT EDecrease on Decrease
CREATE  COMPOSITE EVENT  EAccident on
         EImmobile  SEQUENCE  EDecrease

DEFINE  RULE AccidentNotify,
        Immediate On EAccident
```

```
Condition is EImmobile.segment == EDecrease.segment
Action is
```

- Notify all cars in upstream
- Notify toll station so that waiting cars are not extra tolled
- Notify nearest police about the accident

In the above example, the continuous event query has been defined with both the CQ and event specification. Continuous queries are named (as Immobile and Decrease) and the query name has been used as the query identifier. The Events **EImmobile**, **EDecrease** are defined on the queries by using query names Immobile and Decrease. The composite event **EAccident** is defined using named primitive events EImmobile and EDecrease.

4.3 Masks

In most of the current event processing systems, events are raised for generic happening of interest (e.g., stock object is updated) and conditions further check for specific details using the parameter of the event (e.g., does the stock belong to chip manufacturing group). In Sentinel [40, 41, 42, 43, 44], instance level events permit a limited check on the instance of an object on which an event is generated (and not its attribute values). This approach leads to the generation of a large number of events and the conditions to take specific actions are checked after the events are detected and processed. This leads to excessive generation of events and waste of processing to filter events based on event parameters. Masks provide a mechanism by which attribute-based constraints can be applied to the generation of events from the output of CQs for reducing uninteresting events. This was not possible in Sentinel [40, 41, 42, 43, 44] as events were generated by the underlying system which meant that filtering had to be incorporated in the underlying system over which we have no control. LED did not do that as it meant processing

the event based on parameters before applying the event semantics at every node. In EStream, as events are generated by a CQ it is possible to apply a mask that filters a generic event into different types of events. For example, events belonging to different lanes for the car ADN example can be automated. As another example, events for specific stocks (or categories of stocks) can be generated from the same generic event. Masks provide a powerful mechanism in reducing the number of events generated, especially when the events are coupled with stream processing. The question is to analyze the best place for adding the mask. As it can be added either on the event processing side or on the stream processing side we need to discuss the pros and cons of both.

Definition 1 (Masks) *are Condition on the attributes of the event operator node that are checked before the event is detected.*

CREATE EVENT PRIMITIVE Ename ON Es MASK *maskCondition*

ES is a named CQ or a CREATE CQ statement.

maskCondition is a condition defined on the attributes of an event.

Masks types:

- **Primitive masks:** Masks defined on the primitive event nodes.
- **Composite masks:** Masks defined on the composite event node.

4.3.1 Where to add masks

Since masks help generate only interesting events (by filtering uninteresting events) in the integrated model, it is best to place them at an earlier stage than event processing. The primitive masks can be pushed down to the stream side. Before we generate an event from the stream tuple we evaluate the mask conditions on the attributes of stream. The tuples which do not satisfy the masks do not generate any events whereas those that satisfy generate events. Those tuples/objects are converted into event objects and pushed into the LED buffer. The composite mask has to be checked at the composite

node itself since a complex mask condition defined on the attributes of two or more of the constituent events and hence cannot be checked until the corresponding constituent events are detected. In this thesis, we support only primitive masks since the composite masks require attribute based computations in the event which is not currently supported. The primitive masks can be supported if they are evaluated on the stream side as it supports attribute based computations on tuples.

For example, if in the car ADN (Section 1.3) example, we have to detect accidents only on HOV lanes, we can perform that in two ways. The first and computation intensive way is to add an additional condition in the rule *AccidentNotify* of the complex event *EAccident*.

```
DEFINE Rule AccidentNotify,
        IMMEDIATE On EAccident
            CONDITION EImmobile.segment  = EDecrease.segment &&
        EDecrease.Lane = "HOV" &&   EImmobile.Lane = "HOV"
            ACTION is ....
```

In the computation intensive way, events are generated for cars in all the lanes. Composite event EAccident is also detected and AccidentNotify will be evaluated for all detections of composite events. Action is taken for those tuples which satisfy the condition of the rule and the rest of the tuples are dropped at this stage.

Masks filter the generation of uninteresting events for stream tuples whereas conditions only check whether a rule should be executed or not. Although decision for rule execution can be taken, in principle, after application of a condition, it is a round about way of filtering events. Hence masks is preferred to other ways of filtering.

The second and elegant way is to define masks on the primitive event *EImmobile* and *EDecrease*.

```
            CREATE  PRIMITIVE  EVENT EImmobile on Immobile
```

```
              MASK Lane = "HOV"
CREATE  PRIMITIVE  EVENT EDecrease on Decrease
              MASK Lane = "HOV"
```

By defining the masks on the two primitive events we evaluate the conditions on the stream side itself even before we raise the events. This filters a lot of uninteresting event generation.

4.4 Integration Design

We have discussed the issues for the integration of event and stream processing, introduced CEQs and described the concept of masks. In this section we elaborate on the design of the integration of event and stream models.

4.4.1 Event Generator (EG) Interface

The event generator (or EG) interface provides for the conversion of a stream output into an event object for facilitating the integration of the two models. This interface is responsible for generation of event objects for stream tuples produced by the query. An EG interface persists the mapping of events that are to be generated for the respective queries. It also persists the masks or attribute based constraints defined on the stream tuple attributes for reducing uninteresting generation of events. This interface generates event objects for only those tuples whose masks evaluate to true. The design for the EG interface is such that an application may also consume the output of a CQ directly before events are generated on it.

Functions of the event generator are given below.

- Generation of event objects for ouput of named CQs.
- Populating the attribute values of event with the attribute values of stream tuples.
- Passing the event object to the LED buffer for the event detector to detect it.

The issues of the alternatives for the design of the EG interface as a separate integration module or as an operator are discussed below.

1. **Event Generator as an Integration Module**

   This design has the EG interface is as a separate integration module but within the same address space. The EG integration module should accept tuples from all CQs apply masks and generate events on them. This architecture has the following issues.

   (a) Since the interface is responsible for mapping the CQs with event nodes and for converting the stream tuples into the event objects, the integration module should have the information of CQ identifier and the mapping for the events associates with it. It should also have the output schema of each query so that it can populate the parameter of event object with tuple attribute values. This could be done by sending the query information such as the query identifier and event mapping and the output schema to the integration module whenever a new event is defined on a CQ.

   (b) As the integration module receives output tuples from multiple queries so for each tuple, the query name should be attached to uniquely identify the corresponding event to be raised for the tuple.

   (c) Each output tuple should be enqueued to the integration module. It could be done either by having a resident logic in the output buffer of each query to send the output tuple after attaching the query name to uniquely identify it, or by polling the output buffers of all the queries.

   Although this alternative had the advantages of a layered approach in which the stream or the event system could be replaced with a similar system at a later stage without much modification, it had several disadvantages.

(a) Since the integration module is implemented as a separate module, it should have some buffer capability for queuing tuples produced by all CQs before events are generated for them. The integration module should execute as a separate thread responsible for consuming stream tuples from its buffer and producing event objects. The problem with this design is that the integration module thread will continue to execute even when there are no tuples produced by the query on which events are to be generated. Thus important CPU resources are wasted, which is crucial for DSMS applications.

(b) For each tuple to be enqueued to the integration module there is an overhead of attaching the query name.

(c) The events whose mask evaluate to false will be dropped in the integration module which could have been dropped at an earlier stage.

2. **Event Generator as an operator**

In this architecture the event generator is implemented as a stream operator. This implementation eliminates the drawbacks of the earlier approach and is considered for integration. Here the event generator operator thread is scheduled by the stream scheduler and is suspended whenever there are no tuples in the input queues. The EG operator is attached to each CQ as the root operator. This eliminates the need for attaching the query name for each tuple to uniquely identify the query. The EG operator design automatically gets the tuples produced by the CQ and here the notification of CQ output is not required. With this design of the EG interface the mask evaluation is done at the root operator of each query and tuples will be dropped at the lowest level in the integration model or at the place where they are produced.

Design

(a) Each CQ has this operator as the root operator. The tuples from the query are fed into the event generator operator if events are defined on the query. Otherwise, the tuples are dropped from the input buffer of the event generator operator. This saves the operator from getting scheduled by the scheduler as it will never have tuples to be processed if it does not have any events defined on it.

(b) The output of the CQ is produced by the child operator of the event generator and is directly fed to the application, bypassing the event generator. This avoids any increase in tuple latency for those applications which directly want to consume the output of the CQ.

(c) This operator implementation is capable of modifying the mask and defining new masks at run time (i.e., the time the query is executing). This is possible since the stream server has control over the operator thread and with the help of monitors, the server can synchronize this operator to update the masks defined or add new masks and events.

(d) Event generator stores the mapping of the mask and event names. Whenever any mask is evaluated to be true the corresponding event name are used to raise the events in LED. Once the masks are evaluated to true, the event object is constructed for the tuple and the attribute values of the stream tuple are inserted into the event object.

(e) One of the major components of event generator evaluation is the condition evaluator. FESI (Free Ecma Script Evaluator which is a powerful utility that can perform many functions in Java, condition evaluation being one of them), a java-based tool that provides the functionality of condition evaluation was selected for the purpose of condition evaluation. A lot of effort is required to build a flawless and efficient condition evaluator that can take in any valid

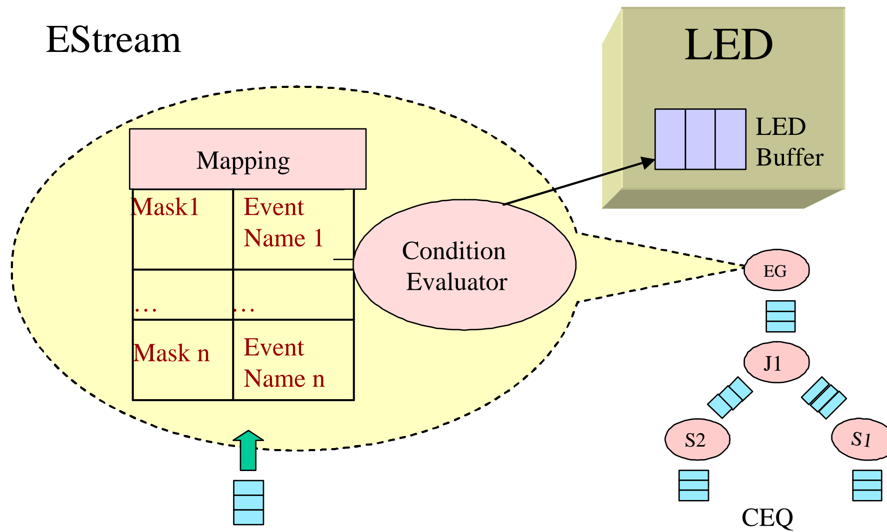

Figure 4.1 EventGenerator Operator

condition and return whether the output is true or false. Hence a decision was made to use the existing condition evaluators available for free.

FESI's condition evaluator works in the following way.

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

One of the initial requirements is to make sure that the condition string is not processed each time a tuple is evaluated. To achieve this, operands are separated out from the condition string and their position is obtained from the schema. The mapping of tuple value and its operands was done using the operand position information. Eventually, the condition evaluation is made simple and efficient. The Figure 4.1 shows the event generator as the root operator of a continuous query to generate events and enqueue in the LED buffer.

4.4.2 Consumer/Producer Approach

This design proposed the idea of having a separate producer/consumer buffer associated with each query in which the EG operator can enqueue event objects to be consumed directly by the event detector of LED.

The problem with this approach is that when multiple CQs are executing, the LED has only one event detector thread to consume event objects from all producer/consumer buffers. In event processing the composite event operators such as SEQUENCE, NOT, etc., require the events to be consumed by the event detector in the relative time order in which they were raised. When there are multiple producer/consumer buffers and the event objects are enqueued in them as and when the CQ produces the output, the event detector thread cannot determine which buffer to consume the next event from to preserve the chronological order of all the event tuples that have been produced. The order in which the events are consumed from the producer/consumer buffer will thus define the final chronological order of the event detection and the actual event detection sequence may be lost. In event processing, the time of event generation (i.e., the order of event generated) is critical. Although events are generated from separate and independent streams, the relationship between them is through the timestamp. In the car ADN example, if two streams are being processed from two lanes (or from two sides) of the same highway, slowing down on traffic in one direction (or lane) can be correlated with the other only if they are happening at the same time, otherwise they would correspond to independent happenings, which have no correlation.

In Figure 4.2, we have three CQs with a consumer/producer buffer for each. We assume that the event detector consumes events using a round robin algorithm. Event $Eo_i(t_k)$ is the i_{th} event in the buffer with global timestamp t_k, which was attached to the event when it was raised. The event detector first consumes Eo_1t_1 from Consumer/Producer buffer C_{p2}. The next object Eo_1t_3 is consumed from C_{p3}, followed by

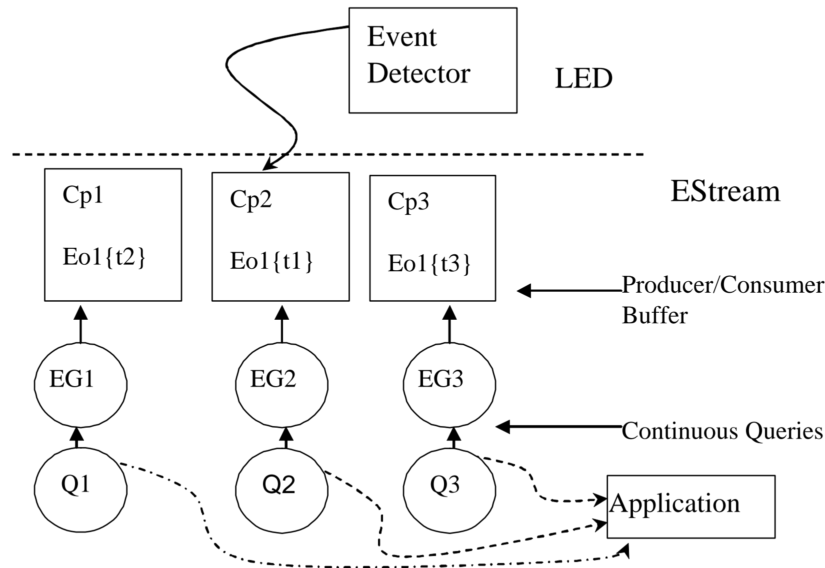

Figure 4.2 Producer/Consumer buffer approach

the event object Eo_1t_2. The final sequence of the event is Eo_1t_1, Eo_1t_3, Eo_1t_2. The events are not detected in the chronological order in which they were raised. The events are produced by the CQs based upon the characteristics of continuous queries (e.g., window parameter) and the arrival rate of the stream. Since the output tuples are not produced at a constant rate, there can be no specific order that can be applied to the event detector consumer thread such that the events that are consumed from multiple producer/consumer buffers are in a chronological order. We do not consider this approach as this is more of a global timestamp problem and our architecture has a single LED buffer as shown in Figure 4.3. The order in which the EG operators enqueue events in the LED buffer will be the order in which they will be detected. We assume that the timestamp of an event detection is closely related to the time at which the tuples in the stream arrive at the processor. We assume that QoS requirements produce the same kind of delay (or lag) in the events produced. If there is very large discrepancy between

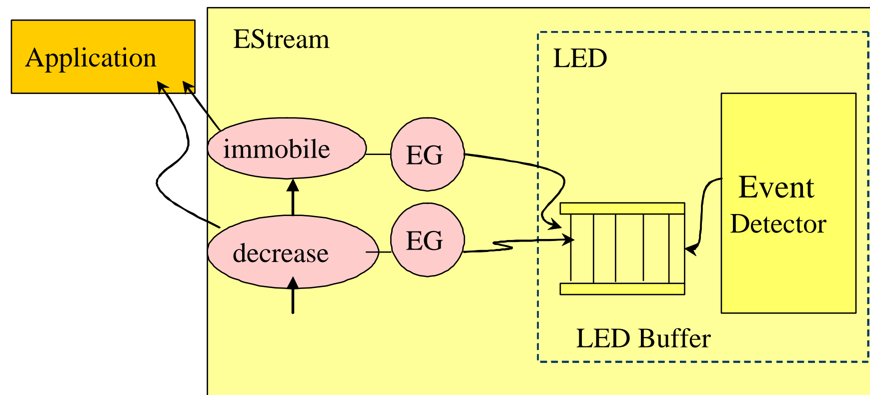

Figure 4.3 Event Generator Operator

the arrival time of tuples and the detection of the events then that will affect the overall accuracy of the system.

4.4.3 Rules and Composite event Generation

Primitive and composite events, as well as rules are generated in the EStream server as part of the ECAAgent object. The event detection algorithm used in LED are not modified. The event object contains all the parameters of the event which can be accessed by the condition and action portion of the rule.

1. **Rule Creation**

   The rules consist of a condition and action as described in the section. EStream server has the capability of associating the rules to already defined events. This dynamic association of rules is possible if the input for the rule has all the information required to uniquely identify events on which the rule has to be created. The input to the rule creation is as Follows:

   Rule Info:

   - Rule Name

- Event Name
- Condition
- Action

Each rule is created with a unique name given by the user. In the car ADN example, the rule called "AccidentNotify" is created . Using the event name the APIs of LED are invoked to create the rule. The rule has the condition and action specified by the class name and the method signature. EStream does not allow dynamic modification of rules. This is because the condition and action that have to be associated with rules are implemented as methods of classes and the class loader of JAVA does not allow dynamic JAVA classes to be added at runtime.

2. **Composite Events:**

The composite event creation is allowed only when its constituent primitive events are already defined. We have also introduced a concept of Future Queries defined in Section 4.6, which are executed at a later time than the time they are defined. If a composite event has to be created with two primitive events, one of which is created on a query that is already defined and the other on a future query, then its creation has to be delayed till the time the future query is instantiated and primitive events defined on it are created. The composite event definition specification has a unique name assigned to each composite event. The EventType is "Composite" and the operator can be any operator supported by LED namely AND, OR, NOT, etc.

- Event Name
- Event Type
- Operator
- EventName1,EventName2,EventName3

In the car ADN example the composite event is created with name EAccident and the SEQUENCE operator in the EImmobile and Edecrease events.

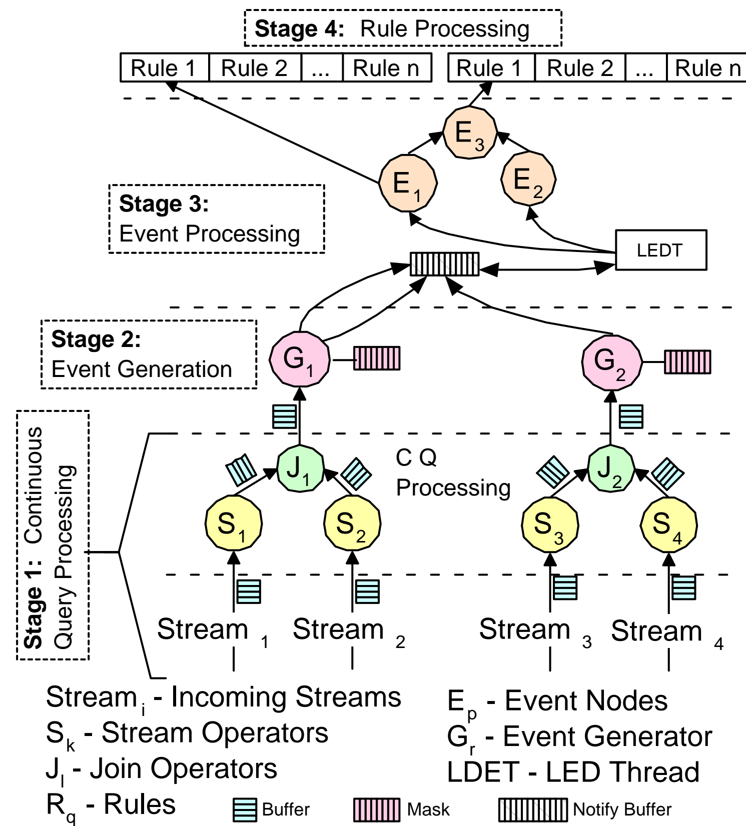

Figure 4.4 Four Stage Integrated Model

4.5 The Integrated Model

The integrated model developed after integration of the two systems has the following parts:

1. CQ processing stage.
2. Coupling stream output with event processing system.
3. Event processing stage that is used for detecting events.
4. Rule processing stage, used to check conditions and to trigger predefined actions once the events are detected.

The seamless nature of our integrated model is due to the compatibility of the chosen event processing model (i.e., an event detection graph) with the model used for stream processing.

4.5.1 Continuous Query (CQ) Processing Stage

This stage processes normal CQs, where it takes streams as inputs, and outputs computed continuous streams. In case an event is defined on the CQ, the output goes to the next stage of the integrated model through the event generator node, else the root node of the CQ directly outputs the results to the application. The scheduling strategies and other QoS delivery mechanisms (such as load shedding) can be applied at this stage. This stage has some basic stream operators such as Select, Project, Join and Aggregate. As Group By operator is an essential operator to fully implement the DBMS queries on stream data, it has been implemented to increase the computational power of the stream processing. A new class of operators called stream modifiers (Section 5.1) is added on the stream side to preprocess the data before the events are generated. The stream modifiers are responsible for computing the change in the values of some attributes between two states of the data stream. The stream modifiers are supported both as a windowed and as a non windowed operator. Another operator called XFILTER is also implemented. This stage is extended to process future queries as well.

4.5.2 Coupling Event and Stream Processing

CQs output data streams in the form of tuples. These continuous queries are associated with an event type and an optional mask in this stage. This operator can take any number of masks and for each mask, a different event tuple/object is created and sent to LED buffer. This stage supports multiple events with multiple masks to be defined for one query. Masks can be modified while the query is executing. The stream

tuples are converted into event tuples and inserted into the event object which is to be processed in next stage.

4.5.3 Event Processing Stage that Detects Events

This is the stage in which the event objects are consumed from the LED buffer and the event nodes are notified. The detection of one event may propagate to the parent node in the event graph. The detection of events is done in various consumption modes defined. When all the events are detected for each event object and rules are processed, the event detector consumes the next event from the LED buffer.

4.5.4 Rule processing stage

The rule processing stage is responsible for checking of canned conditions and taking corresponding actions if they evaluate to true. EStream gives the capability of dynamically adding rules to the events that are already defined. The LED component of EStream supports the processing of rules defined on the events once they are detected. The actions that may be taken can be arbitrary or coded to choose from a canned set of actions such as sending an email notification, sending a message on the PDA or sending an alarm.

4.6 Future Queries

The Stream side has been extended to process queries which will be instantiated and scheduled at a future time, such CQs are called Future Queries. The future queries are defined with a future time at which they have to be started and stopped. When a future query definition is received by EStream server, temporal events are created using the same LED used for event generation. Temporal events are created with the start and end times of the future query and the query is stored in the EStream server. At the start

time, an event is generated which notifies the EStream server that the future query should start. The query is then instantiated and started. When the end query event is raised, the query is stopped and the EStream server has to be purged with the constructs of the query. In the continuous query processing, one issue is that the tuples may be buffered at the intermediate queues associated with the query and so the query should not be stopped until those tuples are processed. To take care of this when the FQ stop time is reached the leaf operators are sent an *endQuery* tuple which will only be propagated to the root operator when all the buffered tuples are processed. When the root operator receives *endQuery* tuple it notifies the EStream server to delete the information stored for the FQ. For removing the constructs of the query from the scheduler, the scheduler thread is locked and then the query operators are removed.

4.7 Summary

In this chapter, we addressed the issues of the design of an integrated system that supports both of event and stream processing. We introduced CEQ specification to define events, conditions and actions along with CQs. We discussed masks for filtering of uninteresting events. We proposed the design alternative for integration of the stream and the event processing systems and the reasons for choosing a specific alternative. We further discussed the generation of composite events and rules and then presented our four stage integrated model.

CHAPTER 5

STREAM SERVER EXTENSIONS

In this chapter, we introduce Stream modifiers and the other extensions to the stream server such as GROUP BY and XFILTER operators which improve the computational power of stream processing.

5.1 Stream Modifiers

Operators in MavStream were primarily defined from relational operators (windowed versions) such as Select, Project, Join and aggregate operators. In many real word scenarios, it becomes necessary to compute various changes on the output of stream processing before it is sent to event processing. A new class of operators called `stream modifiers` is introduced for CQs in order to extend the computation of current stream processing to capture the changes of interest in an input data stream and generate events on the modified input streams. For example, a stock monitoring CEQ can be defined to monitor the live stock feed of a share and sell shares if the price of shares increases by more than 20 percent. Computation for the increase in the price of the share between successive tuples of the stock feed is not possible by using the current stream operators (e.g., Select, Project, Join, etc.,). Hence, stream modifiers can be used to modify the price attribute of the stream to the change of price between two successive tuples. The modified stream can be used for event generation with a mask defined to check if the price change is more than 20 percent. The action of the event can be to make the transaction of selling shares and receiving payment. In this section we introduce the detailed semantics of stream modifiers functions, their input and output and the algorithms to

compute them. Before we introduce the detailed semantics of a stream modifier, we need to introduce some notations:

Let a tuple be represented as: $(A_1, A_2, \cdots, A_n)$, where n is the total number of attributes in the stream schema. e.g., $(CarId, Speed, Direction, Lane) = \{1, 45, East, 3\}$

Definition 2 (SubTuple Ti $(A1, \cdots, Am)$**)** *is represented by the values of attributes* $A1, \cdots, Am$ *for the* i^{th} *state/tuple of data stream. If m is equal to n then* T_i *represents the complete stream tuple.*

For example, T1(CarId, Speed) for the data stream can be represented as $\{1, 45\}$, assuming the tuple defined above is the first tuple of the stream.

Definition 3 (State function Si(Aj)) *represents value of the* j^{th} *attribute in* i^{th} *tuple of the stream.*

Definition 4 (Stream Modifier) *A stream modifier is defined as a function to compute changes (i.e., relative change of an attribute) between two consecutive tuples/states of its input stream. A stream modifier is denoted by* $M(< A_1, A_2, \cdots, A_m > [, P < pseudo >][, O|N])$*, where M is called modifier function that computes a particular kind of change.* $< A_1, A_2, \cdots, A_m >$ *are the parameters required by the modifier function M on which the change is to be computed, m is less than or equal to n. In the following* $P < pseudo >$*, following the parameters, defines a pseudo value for M function in order to prevent underflow. The* $O|N$ *part is called modifier profile, which determines whether the oldest values or the latest values of the sub-tuple shall be given as output. If O is specified, the oldest values are output or the latest values are output if N is specified. The modifier profile is optional and the default is O.*

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, [ ] indicates optional parameters.

1. **ADiff()** is used to detect absolute changes over two consecutive states. It returns absolute change of the values of attributes $(A_1, A_2, \cdots, A_m)$, and SubTuple for the rest of the attributes based on the modifier $O|N$ profile. It is formally defined for case O as follows:

$$ADiff((A_1, A_2, \cdots, A_m >)[O])$$
$$= (\frac{s_{i+1}(A_1)-s_i(A_1)}{s_i(A_1)} \cdots \frac{s_{i+1}(A_m)-s_i(A_m)}{s_i(A_m)}) + T_i(A_{m+1}, A_{m+2}, \cdots, A_n))$$

2. **RDiff()** is used to detect the relative changes over two consecutive states. It returns relative change of the values of attributes $(A_1, A_2, \cdots, A_m)$, and SubTuple for the rest of the attributes based on the modifier $O|N$ profile. It is formally defined for case N as follows:

$$RDiff((A_1, A_2, \cdots, A_m >)[N], P < pseudo >)$$
$$= (\frac{s_{i+1}(A_1)-s_i(A_1)+pseudo}{s_i(A_1)+pseudo} \cdots \frac{s_{i+1}(A_m)-s_i(A_m)+pseudo}{s_i(A_m)+pseudo}) + T_{i+1}(A_{m+1}, \cdots, A_n))$$

3. **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 A_v, A_w, and SubTuple for the rest of the attributes based on the modifier $O|N$ profile. It is formally defined for case O as follows:

$$ASlope((A_v, A_w)[O], P < pseudo >)$$
$$= (\frac{s_{i+1}(A_v)-s_i(A_v)+pseudo}{s_{i+1}(A_w)-s_i(A_w)+pseudo} + T_i(A_x))$$

Where 1 $\leq\{v, w, x\} \leq$ n and x $\neq \{w, v\}$

Event generation may be done directly on the modified input stream or on modified result of a continuous query, so the modifier design should be flexible to support its use either ways. Hence, we have designed the modifier as an operator, which can accept either the input of a stream directly or the output of a continuous query. The specifications of stream modifiers defined above is for non-windowed modifiers which compute change between successive tuples of a stream but sometimes events need to generated for changes computed over intervals. The above stock example can be extended to sell shares if the price of shares continuously rises for 1 hour, monitored in intervals of 5 min. Here, change needs to be computed between the tuples that arrive at the start and at the end of the 5 minute window. Windowed stream modifiers have been introduce to address this issue. Design of both non-windowed as well as windowed modifiers in given below in detail.

5.1.1 Non-Windowed Stream Modifiers

These modifiers do not support intervals and directly produce the change between two consecutive tuples. The modifiers are active till the time events need to be generated and stop themselves if event generation is not required. Synopsis of a single tuple is kept, which is incrementally updated whenever an output is produced. The algorithm 1 explains non-windowed stream modifiers.

5.1.2 Windowed Stream Modifiers

The windowed stream modifiers compute the changes between the first tuple and the last tuple of the window. The windowed implementation of the stream modifier can be placed anywhere in the stream query. The CQ window specifications for the stream server assumes that the operator's window specification is the same as that of the query. If the windowed stream modifiers are implemented with this restriction then the entire query has to be defined on window specifications of the stream modifiers. Since the

Algorithm 1 Non-Windowed Stream modifier

while current tuple timestamp $<$ end time of event generation **do**

  if first tuple **then**

    tuple synopsis equals current tuple

    continue while loop

  end if

  compute change between *tuple synopsis* and current tuple

  output the modified tuple

  update *tuple synopsis* with current tuple

end while

stop modifier

---

stream modifiers produce one tuple per window, all operators in the query will be forced to process one tuple per window if windowed stream modifier are implemented at the leaf level. In such case, the entire concept of windowed operators is lost. Hence for windowed stream modifiers, we need a concept of a separate window specification which is given as an input to the operator when it is defined. The windowed stream modifier is denoted by $M(< A_1, A_2, \cdots, A_m > [, P < pseudo >][, O|N], windowSpecs)$, where the window specs define the begin and the end time of the window. The design also supports overlapped windows and hence maintain a three tuple synopsis,

1. First tuple synopsis: stores the first tuple of the current window.
2. Last tuple synopsis: stores the last tuple of the current window.
3. Overlap first tuple synopsis: stores the first tuple of the overlap window.

The algorithm 2 explains the non-windowed stream modifiers. The current tuple time stamp is compared with the end time of event generation and the operator stops if the former is less than the later. The first tuple for each window is stored in *first tuple*

synopsis and the *last tuple synopsis* is overwritten with every input. When the tuple with timestamp greater than the window bound arrives, the change is computed using *first tuple synopsis* and *last tuple synopsis*. In order to handle overlapping windows, *overlap tuple synopsis* is maintained for the first tuple of the overlapped window.

Algorithm 2 Windowed Stream modifier

```
while current tuple timestamp < end time of event generation do
  if tuple timestamp is within current window bounds then
    if first tuple then
      update first tuple synopsis with current tuple
      continue while loop
    end if
    update the last tuple synopsis with current tuple
    if current tuple timestamp is greater than next begin window time then
      update Overlap first tuple synopsis with current tuple
    end if
  else
    compute change for current window using first tuple synopsis and last tuple synopsis
    update first tuple synopsis with Overlap first tuple synopsis
  end if
end while
stop modifier
```

We have defined the algorithms of both windowed and non-windowed modifiers and in the next section we describe other operator extensions of the stream server.

5.2 XFilter operator

This operator can evaluate any complex condition on the attributes of the stream. This operator does not change the schema of input tuples but outputs tuples which satisfy the condition. This can be called an extended SELECT operator. The input to this operator is an attribute-based condition and the list of attributes on which the condition is defined. This operator uses the FESI Ecma Condition Evaluator for evaluation of the complex conditions. XFilter maps the attributes over which the condition is defined, to the position in input stream and evaluates complex conditions on it. The tuple is output or filtered depending on whether the condition evaluator evaluates the condition on the tuple attributes to true or not.

5.3 Group By Operator

This is a window aggregate Group By operator which integrates the capability of performing aggregate operations on the groups formed over the group by attributes and produces output incrementally after every window is processed. This operator supports all the window specifications for MavStream namely overlapping or landmark windows, disjoint windows etc.

Requirements:

1. The Group By operator should support window based operations.
2. It should produce results incrementally and continuously that are consumed by higher operators (if any).
3. It expects input to be ordered by timestamp for window computation and detection of window and CQ termination and also produces output in timestamp order.
4. It should support all the aggregate operators such as Count, Min, Max, Sum etc.

Syntax of Group By

SELECT Aggregate $Operator_1$ $[ColumnName_1] \cdots$

Aggregate $Operator_n$ $[ColumnName_n]$

ON *stream name*

Group By *attribute list*

The Group By operator accepts as input, the list of attributes over which grouping shall be done, list of aggregate operators and their respective attributes on which aggregation shall be done.

HAVING Condition in Group By: In our implementation Having is not an integral part of Group By and is implemented as a Select or a Xfilter operator on the output of Group By. This design saves the complexity of incorporating a separate Having condition in the Group By operator and uses the other stream operator to filter tuples from the output of Group By. This is addressed with an example:

The query below represented in SQL for evaluating Group By can have the Having condition implemented as a separate Select on the Group By operator output.

```
SELECT carId, Xpos, count(*)
FROM CarLocStr [2min sliding window]
WHERE true
GROUP BY carId, Xpos
HAVING count(*) > 3

This query works exactly as the query given above with
the HAVING condition replaced by a SELECT operator.

SELECT * From
```

```
(       SELECT carId,Xpos,count(*)
        FROM CarLocStr [2 min sliding window]
        WHERE true
        GROUP BY carId,Xpos
    ) as tempTable
WHERE count(*) >3;
```

The issues faced in the implementation of the Group By are explained in the following chapter.

5.4 Summary

In this chapter, we discussed a class of operators called stream modifiers to extend computation on the stream side before generation of events. We also introduced Group By and XFilter operators to increase the computational power of stream processing.

CHAPTER 6

IMPLEMENTATION

EStream, as shown in Figure 6.1, is implemented by integrating the Local Event Detector into the MavStream server. Both the systems are homegrown, implemented in Java and run in the same address space. EStream consists of an extended Continuous Event Query (CEQ) input processor, an instantiator, a query/operator scheduler, a query processor, an event generator, a rule and event manager, an event detector, a runtime optimizer and a load shedder. The user submits the CEQ event generator (or EG)with each query. The ECA part of CEQ is given to the rule and event manager, which generates the computational model for the events. The rule and event manager then defines rules on the event nodes specified in the CEQ. The query scheduler schedules the query which is executed by the query processor. At runtime the event generator is responsible for raising events which are enqueued in the LED buffer as event objects. The event detector consumes the event objects from the LED buffer and detects the corresponding events. For each detected event, the conditions defined on it are checked and actions are taken if the corresponding conditions evaluate to true. The runtime optimizer monitors the QoS and if the user defined performance metric is not met then it dynamically changes the scheduling strategy associated with the stream computational model.

In this chapter we explain the implementation of the input processor, which accepts the CEQ definitions from the user. We further give the implementation of the rule and event manager, which creates event nodes and rules for the user ECA definitions. We then discuss the implementation of the extensions to the query processor as well, which

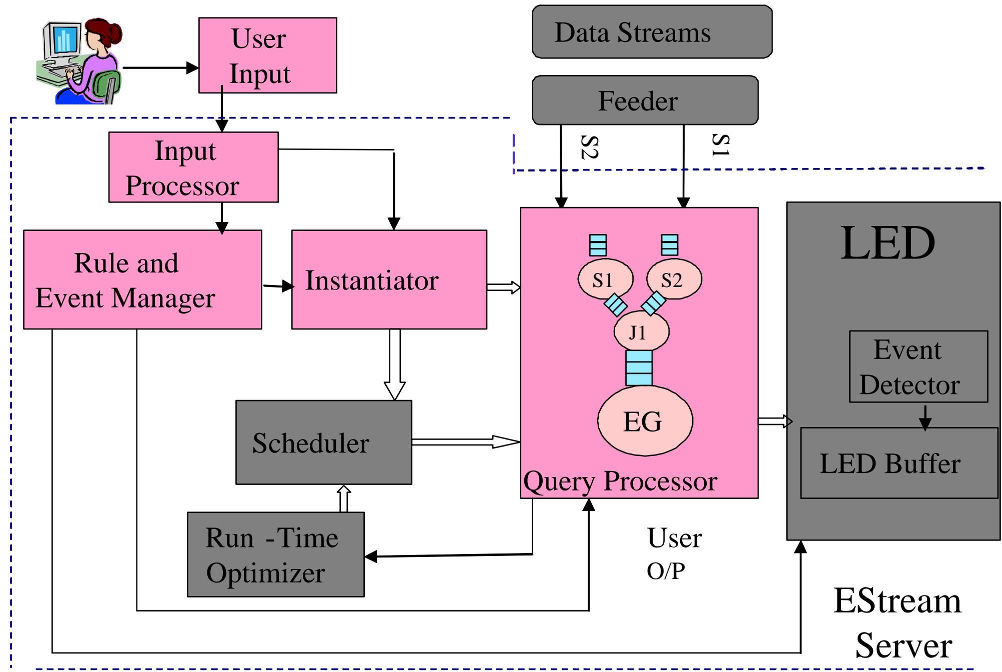

Figure 6.1 EStream Architecture

generates events and checks masks before generating events. Finally, we elaborate on the implementation of stream modifiers and other extensions such as Group By operator and future queries.

6.1 Input Processor

The input processor in Figure 6.2 accepts the CEQs from the user as an input file. The input file is parsed by Input File Parser which splits the information of CQs and the events and rules. The CQ definitions are given to the query plan generator for generating query plans and the event and rule definitions are given to the event container to be temporarily stored before they are defined. Query plan is an object generated by the system for a CQ given by the user. Once the query plans are instantiated and the event generator operators are attached to each query, the event container can be accessed

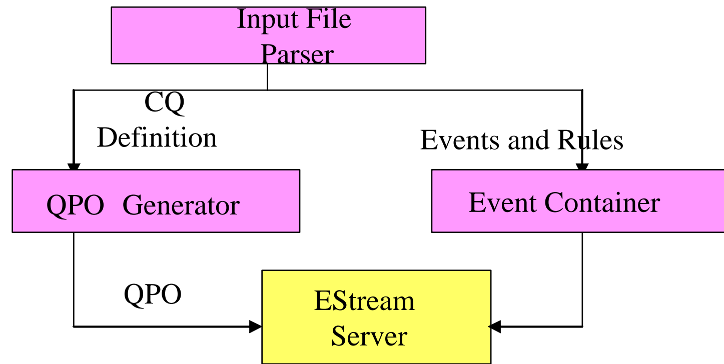

Figure 6.2 Input Processor

to define events and rules. In this section, we have explained the implementation of CEQ input file, the query plan generator and the event container data structure. Important methods of the query plan generator class and the data structure for event container are also described.

6.1.1 CEQ

The continuous event query provides the capability of defining events, rules and continuous queries. CEQ supports CQs to be defined as future queries with start and end times. CEQ specifications are such that any definition can be given alone as well as together. To provide this capability the definitions for rules and events should contain enough information to uniquely specify the CQ on which events and rules should be created. It also provides the capability to modify the masks associated with events and deleting rules associated with events. The specification of CQs, events, masks and rules are provided to the system through an input file. The user can give his input in the form of an ASCII file which will be parsed and analyzed by EStream server. Eventually this input file will be generated by a Graphical User Interface. Following is the list of headers which can be defined in the input file:

- CQ Definition
- Event Info
- Rule Info
- Modify Mask
- Delete Rule

Each header has a format in which the input should be defined. The input file can have multiple definitions for the same header in a single file.

1. **CQ definition:** The MavStream functionality has been extended to include web enabled GUI for query specification. The GUI generates an ASCII file of the user input. This ASCII file can either be generated using the GUI or can be manually written as a parameter delimited file. The CQs are defined by giving the stream operators and the information required for instantiating each stream operator. The CQ specification also requires the user to give the association between operators to form a query tree.

2. **Event Info:**The event information is defined under the Event Info header. This can be used to define both primitive as well as composite events. Primitive events are attached to a query name to uniquely identify the query with which the respective events are associated. Each event can be defined with an optional mask. In case a composite event is defined then the query name should not be given and the operator and event names on which the composite event is created have to be defined.

    - Event Name
    - CQ Name
    - Event Type
    - Mask
    - Operator

- EventName1,EventName2,EventName3

3. **Rule Info:** Rules for the events have to be defined under the Rule Info header. A rule has a rule name associated with it, a condition and an action. The condition and action are predefined methods which could be associated with the event.
   - Rule Name
   - Event Name
   - Condition
   - Action
4. **Modify Mask:** The modify mask option can be used to modify a mask defined on the event continuous query. The mask can be modified by defining the input to uniquely identify the previous mask and giving a new mask condition. The server updates the mask with the new condition defined.
   - CQ Name
   - Event Name
   - New Mask
5. **Delete rule:** This option allows the user to delete some conditions and actions associated with an event (primitive or composite). The system will drop the rule associated with the event. This becomes useful when there is more than one rule defined on the event and at a later stage one wants to drop a rule.
   - Event Name
   - Rule Name

6.1.2 Query plan generator

This query plan generator generates a query plan and gives it to the server. Each operator definition is populated in a data structure called operatorData. The operatorData is wrapped in an OperatorNode that has references to the parent and child

operators. The query plan stores the operatorNode for the root operator and is given to the server for instantiation. Since the OperatorNode has references, the server can access all the OperatorNodes of the query. The query plan generator is implemented as the QueryPlanGenerator class. The methods that create the query plan are as follows:

1. **CreateOperatorDataInfo:** This method is responsible for the creation of operator data from the user input.
2. **CreateOperatorNodeInfo:** This method is responsible for the creation of the operator node by wrapping the operatorData and associating the parent and child information with it.
3. **CreateQueryPlan:** This method calls the above two methods for the creation of the query plan.

6.1.3 Event container

The event container temporarily stores the information regarding events and rules until they can be created, following the creation of the CQ. This is implemented in the ECADefinitionContainer data structure. The ECADefinitionContainer can be described as

1. **CompositeEventInfo:** This HashTable stores the information of composite events defined by the user.
2. **PrimitiveEventInfo:** This HashTable stores the information of primitive events defined by the user. Optimization is done by mapping the primitive event definitions based on the query name on which the events have to be defined. This design avoids the event generator to be locked multiple times when more than one events are defined on a query which is already executing.
3. **Rule info:** This HashTable stores all the rule definitions given by the user.

6.2 Rule and Event Manager

This is responsible for creation of the events and rules. Since both LED and event and rule manager are in the same address space, the APIs of LED can be called for creation of event nodes and associating rules on the event nodes created. It accesses the event container for getting the definitions of the events and rules and creates the event nodes for the ECA part of the CEQ defined by the user. Creation of the events is done in an order such that primitive events are created first then composite events and finally the rules are defined on them.

6.3 Query Processor

The query processor has the implementation of all operators. In this section, we give the implementation of the event generator operator and masks. The implementation of the other operators is given in the following sections.

The event generation for each CQ is done by the event generator operator which is attached as the root operator of each query after it is instantiated. The event generator operator executes in the query processor. Stream tuples are fed to the event generator operator where they are compared against available masks and then converted into event objects. Attributes of the stream tuples are inserted as event attributes and event objects are put into the LED buffer. This is implemented by making extensions to the query processor and the instantiator by implementing the event generator operator and associating it before every query is scheduled.

6.3.1 Event Generator Operator and Masks

The main issues for the event generator operator were dynamic addition of masks and persisting the query to event mapping for evaluating masks using the condition

evaluator and generating event. The hashtable for storing the masks and the cases for addition of masks are described below.

MaskAndEventHandle HashTable: Stores the mask condition and eventHandle key value pair, which is accessed to obtain the reference of the event nodes when a mask is evaluated to be true.

Adding events and Masks: There may be various cases for the addition of events and masks. It should be noted that in the EDG model, the case where a single mask will generate two events will never occur for the same event generator. This is because the event node can be shared to create complex events instead of creating another event with the same mask. If new conditions and actions are to be added then a separate rule is defined on the same event node.

- **The event does not have a mask defined:** Here the eventHandle is added with a mask condition as "true". The eventHandle associated with the event is added to the MaskAndEventHandle HashTable with a true mask condition.
- **Modification of Mask:** The user can also modify mask at the operator execution time by giving event name and the new mask. This operator will lock itself and access the mask for the corresponding event name and update the HashTables.

The algorithm 3 explains the implementation of the event generator operator. The condition evaluator evaluates the masks on each incoming tuple and drops the tuples whose masks evaluate to false. If a mask condition evaluation is successful then the corresponding event associated with the mask is raised. Since the tuples here are assumed to be ordered by timestamp or a sequence number, we can check if the ordering attribute of the tuples is greater than the end query, and if it evaluates to true, the operator is stopped.

The checking of masks before event generation significantly reduces the number of events generated. The more the selectivity of the mask, the more will be the number of

Algorithm 3 Event Generator Operator

```
if no tuples in input buffer then
  Suspend operator and Notify scheduler
  { Semaphore implemented to add mask}
  if mask is to be added or modified then
    Lock the operator
  end if
  if tuple timestamp is ≥ endQuery then
    stop operator
  else
    Check all the mask conditions
    if mask is true then
      Raise the corresponding event
    end if
  end if
end if
```

events filtered. The experiment below gives the improvement by the implementation of masks for a CEQ.

The experiment was run on a machine with a single Xeon processor, 2.4GHz , 1GB RAM and Red Hat Linux 8.0 as the operating system. The data set for performance evaluation is a modified version of the dataset used by the Stanford Stream project [10, 37]. The data is stored in our database that is modified to generate synthetic data stream. This synthetic data stream is fed to this system using the feeder module, where the delays between tuples follow Poisson distribution.

In this experiment, a CEQ is executed with and without masks to observe the performance difference. Time difference at which the rule associated with the query evaluates to true and the time at which the event was generated is measured and is named as the action execution latency. The experiment is run for a datasets of 2000 tuples, 10,000 tuples and 30,000 tuples. The data rate is 100 tuples/sec. The scheduling strategy used is path capacity scheduling[52, 1]. The buffer is unbounded. The query that is evaluated is

```
CREATE CQ  AUTOMATEDMONITOR AS
    SELECT * from CarLocStr
    WHERE carId > 100

    CREATE EVENT " ResidentialSpeedingTicket" on AUTOMATEDMONITOR
    MASK "true"
    CREATE RULE " SpeedingTicket"
    CONDITION Speed > 30
    ACTION "CalculateTicketBasedOnZone"
```

The query named AUTOMATEDMONITOR is to monitor the speed of cars which are public cars with CarId > 100 to be within the speed limit in the Residential area. If the speed of the car increases by 30mph then a ticket event "ResidentialSpeedingTicket" will be generated and the owner will be mailed the ticket.

In the Figure 6.3 and Figure 6.4, we observe that with the application of masks the number of events generated is considerably reduced thus reducing the traffic in the LED buffer. This also prevents the event detector to consume event objects from the LED buffer and drop them at the rule evaluation time. With the application of masks only those events whose rules evaluate to true are generated. This result in a decrease

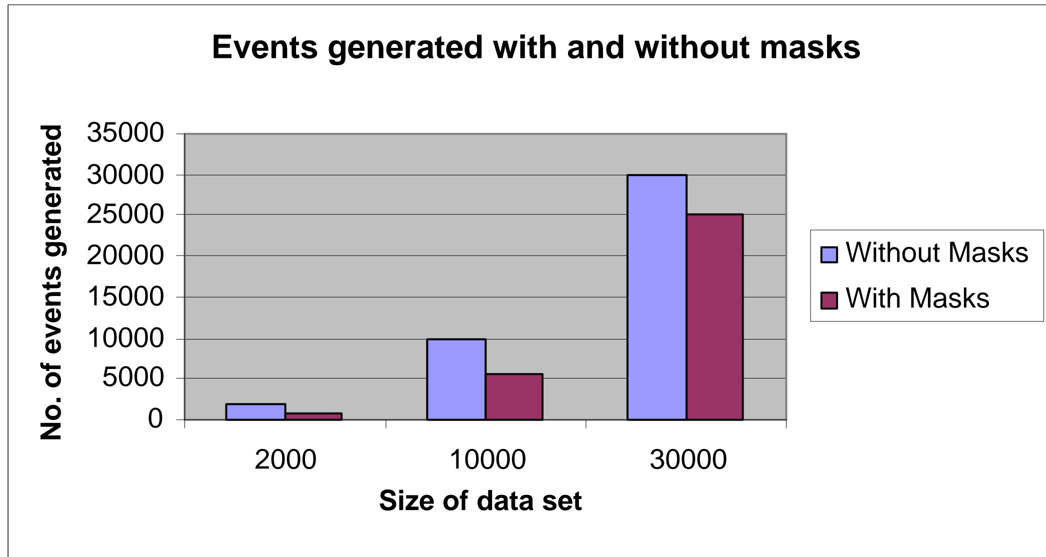

Figure 6.3 Events generated with and without masks.

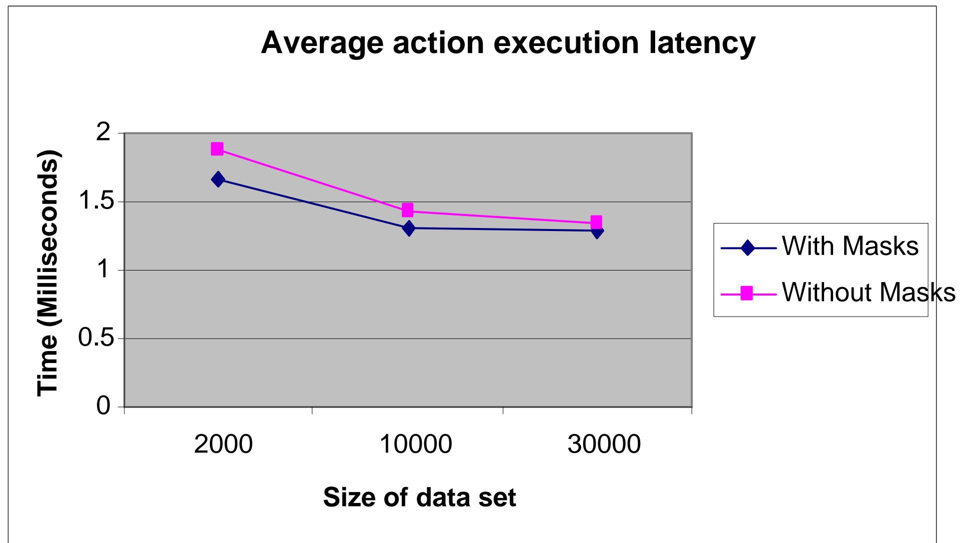

Figure 6.4 Average action execution latency

of the average event execution latency for various data sets and can be seen from the Figure 6.4.

Many classes have been created and extended for this implementation. Some of the major classes and their methods are explained below

1. **EventGenerator:** This class is responsible for storing the masks and the EventHandles which are raised once the mask condition is satisfied by the stream tuple. The masks and Events are stored in the
   (a) *addMaskAndEventHandle*: This accepts the mask and eventHandle associated with the mask. This is responsible for populating the above HashTables.
   (b) *runOperator*: This method implements the abstract method for super class Operator. This method is executed by the scheduler whenever the event generator operator is scheduled. Masks are checked in this method, event objects are created and corresponding events are raised in this method.
2. **EventOperatorLock** This class is responsible for locking the event generator operator if the query execution has started. It is essential to lock the event generator in case masks have to be modified or new events have to be added to already executing CQs.
   (a) *getLockToAddMask:*This synchronized method sets the semaphores in the EventGenerator.
   (b) *releaseLock:* This method releases the lock of the EventGenerator.
3. **GenerateEvent:** This class makes the API calls to the LED for creation of the events. Events once generated have an eventHandle with which they can be referenced.
   (a) *generatePrimitiveEvent:* This method generates primitive events and returns an eventHandle associated with the event.

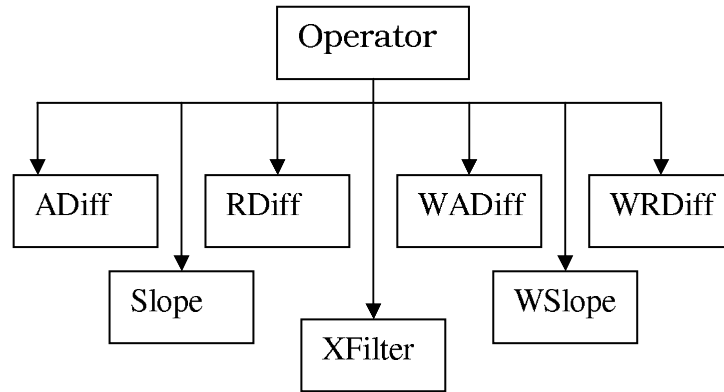

Figure 6.5 Stream Modifiers and XFilter

(b) *generateCompositeEvents:* This method generates composite event and also returns eventHandle. It accepts the names of constituent events, the name of the event that shall be created and the operator on which the event will be created.

6.4 Stream Modifiers and XFilter operators

The stream modifiers and XFilter are implemented as operators. All the operators extend from the parent abstract class Operator as seen in the Figure 6.5. Other than implementing these operators as classes, extensions had to be done to integrate these new operators with the server. Extensions are done to the instantiator for instantiating these operators and also to the data structure called operatorData which stores the parameters of the operator and is used for creating query plan. Extensions done to the data structure and instantiator are explained below.

6.4.1 Instantiator extensions

The instantiator has been extended for implementing these operators. Separate methods have been written to instantiate each operator. These methods create an object

of the operator and populate the operator object with the parameters given by the user. The methods are given below:

- InstantiateAdiff
- InstantiateWAdiff
- InstantiateRdiff
- InstantiateWRdiff
- InstantiateSlope
- InstantiateWSlope
- InstantiateXFilter

6.4.2 Operator Data

This data structure is extended with the following constructs to store the input of the user before the operators are instantiated:

1. **GroupByAttributes:** This is a vector of the attributes on which Group By operator is to be grouped.
2. **XFilterAttributes:** This stores the vector of attributes on which XFILTER has to evaluate the condition.
3. **ModifierAttributes:** This stores attributes of the modifier in a vector.
4. **ModifierProfile:** This stores the modifier profile integer which represents whether the SubTuple should contain the old or the new state.

6.5 GroupBy Operator:

Group By is created by extending the operator class. It has a window concept and is a blocking operator. It supports various aggregate functions of Sum, Aggregate, Count, etc., implemented as methods within the operator. In this section we have discussed the

algorithm for Group By operator and the issues in using a HashTable implementation of JAVA for this operator.

6.5.1 Algorithm

Some data structures are introduced before giving the algorithm. The operator maintains groups for each window in a HashTable called GHashTable. Each group of the GHashTable has a GroupHandle (GH) attached to it. First we describe the data structures and then we discuss the algorithm.

1. **GHashTable:** Stores all the tuples in the current window hashed by the values in their grouping attributes. Each bucket in HashTable stores the tuple in the following form. $(< T1, t1 >)$, $(< T2, t2 >) \cdots$ $(< TN, t3 >)$, where Ti represents the i_{th} tuple and t1, t2$\cdots$tn represent the tuple timestamps. This operator uses two GHashTables which are described below:
   - **currentGHashTable:** This HashTable is used to construct the groups for the current window.
   - **overlapGHashTable:** This HashTable is used to temporarily store the tuples while the groups are constructed for the overlap part of the next window. In case the operator processes disjoint windows then this HashTable will be empty.
2. **GroupHandle, GH(one for each group):** Stores the output tuple of each group and the maximum timestamp (GH.maxTS) among all the tuples in the group.

The overview of 4 is given below:

- Tuples read from the stream, are inserted into the corresponding group in the currentGHashTable whose values match the values of Group By attributes in the tuple.

- Those tuples whose timestamp is greater than the start time of next window are grouped in currentGHashTable as well as OverlapGHashTable.
- When tuple timestamp is greater than the current window bounds then the groups in currentGHashTable are processed. Aggregate operators are then computed on each group in currentGHashTable.
- GH is computed for each group which is sorted based on the timestamp of each GH.
- The sorted lists of GH are given as output for each window. The currentGHashTable is purged and contents of overlapGHashTable are put in the currentGHashTable. The tuples are processed for the next window in similar manner until the end query is reached.
- In case of disjoint window, overlapGHashTable is not maintained and grouping is done in currentGHashTable.

6.5.2 Issues in using the JAVA HashTable:

To implement the groups by JAVA HashTable class, each bucket in the table should represent a group. In JAVA, if two key value pairs with the same key are inserted into the HashTable then they are hashed to the same bucket. If the bucket already contains some value for the same key then JAVA overwrites the value with last value hashed. Hence, if we have multiple tuples in a window with same values for attributes on which we have decided to do Group By, then only the last tuple will be stored as the rest of them will be overwritten. To avoid this problem, we have implemented our own HashTable in which if the same key and different value tuples are inserted then all the values associated with the key get stored in a vector inside the bucket to which they are hashed. It is also possible in JAVA that more than one key may be hashed to the same bucket. This can be avoided by associating each bucket with a list of Vectors representing different groups

Algorithm 4 Group By Operator

if current tuple timestamp < End Query **then**
  if current tuple timestamp is within current window bounds **then**
    Group the tuple in *currentGHashTable*
    if current tuple timestamp > next begin window time **then**
      Group tuple in *overlapGHashTable*
    end if
  else
    Apply aggregate operators on groups in *currentGHashTable*.
    Sort GH obtained for each group based on timestamp.
    Output the sorted GH List.
    Exchange the *currentGHashTable* and *overlapGHashTable*.
    Clear the *overlapGHashTable*.
  end if
else
  stop operator
end if

being hashed to the same bucket. Each vector in the list contains tuples for only one group. In the Figure 6.6, Group By is done on first two attributes of the stream. Two tuples are hashed to the same bucket as they have the same key. In GHashTable the first two tuples are inserted in the vector for Group1. We assume that third tuple may also be hashed into the same bucket. Although the bucket is the same, the key is not the same so a new group (Group 2) is created to insert the third tuple.

Sum,Max,Min,Count and Average: These methods compute the aggregate values for each group when the window for the group is elapsed.

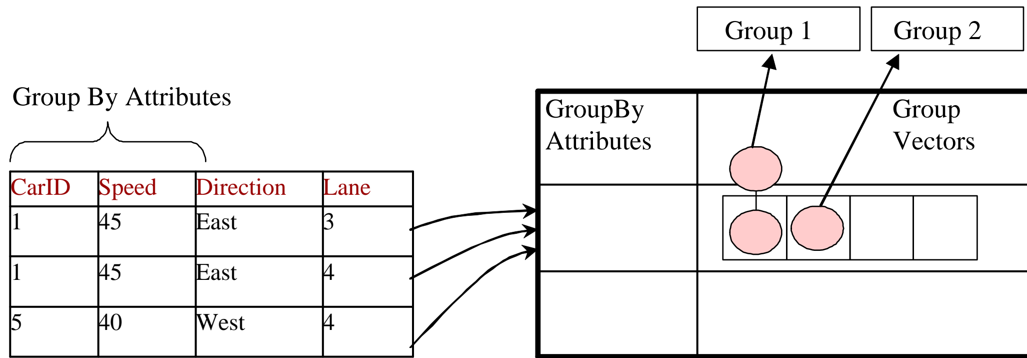

Figure 6.6 GHashTable design

6.6 Future Queries

The future queries are implemented by setting temporal events for the start and end time of the query. A user query is checked before it is instantiated that whether it is defined as a future query or not. If it is, then the instantiation is delayed for the time till the LED temporal event for start time of the query is raised. The class below gives the implementation of the query as a future query.

It is to be noted that the same local event detector that is integrated with the MavStream is used for executing future queries. The temporal events supported by the LED are used for starting the instantiation and execution of a CQ. Events for starting and stopping the CQs are generated in the same LED.

1. **FutureQueryGenerateEvent:** This class calls the API of the LED for generating the event for the future query.
    - *SetStartEvent:* This method accepts a time at which the query is to be scheduled and query name as the query identifier. It sets the temporal event for the start of the query. The action part of the rule defined for the event calls the method for instantiating and scheduling the query at the query start time.

- *SetEndEvent:* This method accepts a time at which the query is stopped and the query name as query identifier. It sets the temporal event for stopping the query. The action part of the rule defined for the event calls the method for stopping the query and deleting query constructs.

6.7 Summary

In this chapter we discussed the extensions to various modules of EStream system. We first discussed the input processor and the CEQ implementation. Then we explained the extensions to query processor with the query generator and masks. We further elaborated on the implementation of stream modifiers, XFILTER and Group By operators.

CHAPTER 7

CONCLUSION AND FUTURE WORK

In this thesis, we have addressed the design and implementation of EStream, a system that supports the need of advanced applications that require not only stream processing but also event processing. We have given an integrated model to enhance the current stream processing data model horizontally and vertically to combine event and stream processing systems:

1. To detect composite event patterns through enhanced continuous queries with event processing component. Query specification extensions provides a way to define stream queries, a way to define events and rules and a combination there of.
2. To perform complicated computations (i.e., capture complicated change patterns) through a set of named continuous queries with a family of stream modifiers. Windowed stream modifiers with seperate window specifications are introduced to compute change in the stream state over windows.
3. To filter uninteresting events based on user defined attribute-based conditions. Multiple events can be generated from the same stream by defining unique constraints.
4. To define and execute a large number of rules.
5. Extend the stream computational model with stream operators and provide support for future queries.

Future work can be in the area of event detection. EStream detects events on point based semantics, where when one primitive event is detected, all composite events and rules associated with the it are also detected and then only the next primitive event will be detected. Event detection can be extended to support interval based semantics,

where event model can be extended with queues and event detection with scheduler to avoid this bottle neck. EStream supports attribute-based constraints on event detection only for primitive events and complex events can be defined using operators based only on time-based semantics. The current event processing model can be extended with attribute-based semantics to support attribute-based constraints on complex event and also complex event operators which evaluate on attribute-based semantics. Semantic window [55] can also be implemented on the stream side to enhance the window specifications of stream processing.

REFERENCES

[1] Q. Jiang and S. Chakravarthy, "Scheduling Strategies for Processing Continuous Queries over Streams," in *Proc. of BNCOD*, Jul. 2004.

[2] B. Babcok *et al.*, "Operator scheduling in data stream systems," *The VLDB J.*, vol. 13, pp. 333–353, 2004.

[3] D. Carney *et al.*, "Operator Scheduling in a Data Stream Manager," in *Proc. of VLDB*, Sep. 2003.

[4] N. Tatbul *et al.*, "Load Shedding in a Data Stream Manager," in *Proc. of VLDB*, Sep. 2003.

[5] B. Babcock, M. Datar, and R. Motwani, "Load Shedding for Aggregation Queries over Data Streams," in *Proc. of ICDE*, Mar. 2004.

[6] Q. Jiang and S. Chakravarthy, "Data Stream Management System for MavHome," in *Proc. of ACM SAC*, Mar. 2004.

[7] D. Abadi *et al.*, "Aurora: A New Model and Architecture for Data Stream Management," *VLDB Journal*, vol. 12, no. 2, Aug. 2003.

[8] J. Chen *et al.*, "NiagaraCQ: A Scalable Continuous Query System for Internet Databases," in *Proc. of SIGMOD*, 2000.

[9] S. Madden and M. J. Franklin, "Fjording the Stream: An Architecture for Queries over Streaming Sensor Data," in *Proc. of ICDE*, 2002.

[10] R. Motwani *et al.*, "Query Processing, Resource Management, and Approximation in a Data Stream Management System," in *Proc. of CIDR*, Jan. 2003.

[11] A. Das, J. Gehrke, and M. Riedewald, "Approximate Join Processing over Data Streams," in *Proc. of SIGMOD*, 2003.

[12] S. Chakravarthy *et al.*, “Design of Sentinel: An Object-Oriented DBMS with Event-Based Rules,” *Information and Software Technology*, vol. 36, no. 9, pp. 559–568, 1994.

[13] S. Gatziu and K. R. Dittrich, “Events in an Object-Oriented Database System,” in *Proceedings of Rules in Database Systems*, Sep. 1993.

[14] A. P. Buchmann *et al.*, *Rules in an Open System: The REACH Rule System.* Rules in Database Systems, 1993.

[15] S. Chakravarthy and D. Mishra, “Snoop: An Expressive Event Specification Language for Active Databases,” *Data and Knowledge Engineering*, vol. 14, no. 10, pp. 1–26, Oct. 1994.

[16] S. Gatziu and K. R. Dittrich, “SAMOS: An Active, Object-Oriented Database System,” *IEEE Quarterly Bulletin on Data Engineering*, vol. 15, no. 1-4, pp. 23–26, Dec. 1992.

[17] D. L. Lieuwen, N. H. Gehani, and R. Arlein, “The Ode Active Database: Trigger Semantics and Implementation,” in *Proc. of ICDE*, Mar. 1996, pp. 412–420.

[18] N. H. Gehani, H. V. Jagadish, and O. Shmueli, “Event Specification in an Object-Oriented Database,” in *Proc. of SIGMOD*, San Diego, CA, June 1992, pp. 81–90.

[19] N. H. Gehani and H. V. Jagadish, “Ode as an Active Database: Constraints and Triggers,” in *Proc. of VLDB*, Sep. 1991, pp. 327–336.

[20] H. Engstrom, M. Berndtsson, and B. Lings, “Acood essentials,” University of Skovde, Tech. Rep., 1997.

[21] S. R. Madden *et al.*, “The Design of an Acquisitional Query Processor for Sensor Networks,” in *Proc. of SIGMOD*, 2003.

[22] ——, “TAG: a Tiny AGgregation Service for Ad-Hoc Sensor Networks,” *In Proc. of OSDI*, Dec. 2002.

[23] P. Bonnet, J. E. Gerhke, and P. Seshadri, "Towards Sensor Database Systems," in *Proc. of MDM*, Jan. 2001.

[24] Y. Yao and J. E. Gehrke, "Query Processing in Sensor Networks," in *Proc. of CIDR*, Jan. 2003.

[25] Q. Jiang, R. Adaikkalavan, and S. Chakravarthy, "NFM^i: An Inter-domain Network Fault Management System," in *Proc. of ICDE*, Apr. 2005.

[26] U. Schreier *et al.*, "Alert: An Architecture for Transforming a Passive DBMS into an Active DBMS," in *Proc. of VLDB*, 1991.

[27] O. Diaz, N. Paton, and P. Gray, "Rule Management in Object-Oriented Databases: A Unified Approach," in *Proc. of VLDB*, Sep. 1991.

[28] N. H. Gehani, H. V. Jagadish, and O. Shmueli, "COMPOSE: A System For Composite Event Specification and Detection," AT&T Bell Laboratories, Tech. Rep., Dec. 1992.

[29] 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.

[30] E. N. Hanson, "The Design and Implementation of the Ariel Active Database Rule System," *IEEE TKDE*, vol. 8, no. 1, 1996.

[31] P. Seshadri, M. Livny, and R. Ramakrishnan, "The Design and Implementation of a Sequence Database System," in *Proc. of VLDB*, 1996, pp. 99–110.

[32] A. Dinn, M. H. Williams, and N. W. Paton, "ROCK & ROLL: A Deductive Object-Oriented Database with Active and Spatial Extensions," in *Proc. of ICDE*, 1997.

[33] S. Babu and J. Widom, "Continuous Queries over Data Streams," in *ACM SIGMOD RECORD*, Sep. 2001.

[34] M. F. Mokbel *et al.*, "PLACE: A Query Processor for Handling Real-time Spatio-temporal Data Streams," in *Proc. of VLDB*, Sep. 2004.

[35] H. Balakrishnan *et al.*, "Retrospective on Aurora," *VLDB Journal: Special Issue on Data Stream Processing*, 2004.

[36] R. Avnur and J. M. Hellerstein, "Eddies: Continuously Adaptive Query Processing," in *Proceedings, International Conference on Management of Data (ACM SIGMOD)*, May 2000, pp. 261–272.

[37] A. Arasu *et al.*, "Linear Road: A Stream Data Management Benchmark," in *Proc. of VLDB*, Sep. 2004.

[38] N. Gehani and H. Jagadish, "Ode as an Active Database: Constraints and Triggers," in *Proc. of VLDB*, Barcelona, Spain, 1991, pp. 327–336.

[39] S. Gatziu and K. R. Dittrich, "SAMOS: An active, object-oriented database system," *IEEE Quarterly Bulletin on Data Engineering*, vol. 15, no. 1-4, pp. 23–26, December 1992.

[40] H. Lee, "Support for Temporal Events in Sentinel: Design, Implementation, and Preprocessing," Master's thesis, Database Systems R&D Center, CIS Department, The University of FLorida, Gainesville, 1996.

[41] S. Chakravarthy *et al.*, "Design of Sentinel: An Object-Oriented DBMS with Event-Based Rules," *Information and Software Technology*, vol. 36, no. 9, pp. 559–568, 1994.

[42] R. K. Honnavalli, "Design and implementation of an event based shop floor control application on sentinel – an active oodbms," Master's thesis, Industrial and Systems Engineering Department, Aug 1995.

[43] H. Liao, "Global events in sentinel: Design and implementation of a global event detector," Master's thesis, Database Systems R&D Center, CISE, University of Florida, E470 CSE Building, Gainesville, FL 32611, January 1997.

[44] H. Lee, "Support for temporal events in sentinel: Design, implementation, and preprocessing," Master's thesis, Database Systems R&D Center, CISE, University of Florida, E470 CSE Building, Gainesville, FL 32611, Aug 1996.

[45] S. Rizvi, S. R. Jeffery, S. Krishnamurthy, M. J. Franklin, N. Burkhart, A. Edakkunni, and L. Liang, "Events on the edge," in *Proceedings, International Conference on Management of Data (ACM SIGMOD)*, 2005, pp. 885–887.

[46] O. Cooper, A. Edakkunni, M. J. Franklin, W. Hong, S. R. Jeffery, S. Krishnamurthy, S. Rizvi, and E. W. 0002, "Hifi: A unified architecture for high fan-in systems." in *Proc. of VLDB*, 2004, pp. 1357–1360.

[47] A. Arasu, S. Babu, and J. Widom, "The CQL Continuous Query Language: Semantic Foundations and Query Execution," *Stanford Technical Report*, Oct. 2003.

[48] "Stream processing engine," Oct. 2005. [Online]. Available: http://www.streambase.com/www/misc/speintro.html

[49] "Apama's technology," April 2005. [Online]. Available: http://www.apama.com/technology/index.html

[50] "Real Time Event Management in Financial Markets," 2005. [Online]. Available: www.gemstone.com/company

[51] B. Brian *et al.*, "Chain: Operator Scheduling for Memory Minimization in Stream Systems," in *Proc. of SIGMOD*, 2003.

[52] V. K. Pajjuri, "Design and implementation of scheduling strategies and their evaluation in mavstream," Master's thesis, Information Technology Laboratory, CSE Dept., The Univ. of Texas at Arlington, 2004. [Online]. Available: http://itlab.uta.edu/ITLABWEB/Students/sharma/theses/Vamshi.pdf

[53] S. Chakravarthy, J. D. Yang, and S. Yang, "A formal framework for computing composite events over histories and logs," University of Florida, E470-CSE, Gainesville, FL 32611, Tech. Rep. UF-CIS TR-98-017, November 1998.

[54] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim, "Composite Events for Active Databases: Semantics, Contexts, and Detection," in *20th International Conference on Very Large Databases (VLDB)*, 1994, pp. 606–617.

[55] Q. Jiang, R. Adaikkalavan, and S. Chakravarthy, "Towards an Integrated Model for Event and Stream Processing," *TR CSE-2004-10, CSE Dept., Univ. of Texas at Arlington*, 2004.

BIOGRAPHICAL STATEMENT

Vihang Garg was born in Uttar Pradesh, India, in 1981. He received his B.S. degree in Computer Science and Engineering from Agra University, India, in 2003. In the Fall of 2003, he started his graduate studies in Computer Science at The University of Texas, Arlington. He received his Master of Science in Computer Science and Engineering from The University of Texas at Arlington, in Dec 2005.