Event Composition and Detection in Data Stream Management Systems

Mukesh Mohania[1], Dhruv Swamini[2], Shyam Kumar Gupta[2], Sourav Bhowmick[3], and Tharam Dillon[4]

[1] IBM India Research Lab, I.I.T., Hauz Khas, New Delhi
[2] Dept of Computer Science and Engg, I.I.T. Delhi, Hauz Khas, New Delhi
[3] Nanyang Technological University, Singapore
[4] Faculty of Information Technology, University of Technology Sydney, Australia

Abstract. There has been a rising need to handle and process streaming kind of data. It is continuous, unpredictable, time-varying in nature and could arrive in multiple rapid streams. Sensor data, web clickstreams, etc. are the examples of streaming data. One of the important issues about streaming data management systems is that it needs to be processed in real-time. That is, active rules can be defined over data streams for making the system reactive. These rules are triggered based on the events detected on the data stream, or events detected while summarizing the data or combination of both. In this paper, we study the challenges involved in monitoring events in a Data Stream Management System (DSMS) and how they differ from the same in active databases. We propose an architecture for event composition and detection in a DSMS, and then discuss an algorithm for detecting composite events defined on both the summarized data streams and the streaming data.

1 Introduction

The data in Data Stream Management System (DSMS) is delivered continuously, often at well defined time intervals, without having been explicitly asked for it [9, 10]. The data needs to be processed in near real-time, as it arrives because of one or more of the following reasons – it may be extremely expensive to save the raw streaming data to disk; the data is likely to represent real-time events, like intrusion detection and fault monitoring, which need to be responded to immediately. Another major challenge handling streams is because of their delivery at unreliable rates, the data is often garbled, and they have limited processor resources. It is likely to be subjected to *continuous queries (CQ)* – which need to be evaluated continuously as data arrives, in contrast to the *one-time queries,* which are evaluated once over a point-in-time snapshot of the data set. The streaming data being infinite in size, and if the need for storage be, it has to be summarized or aggregated [11].

Active functionality [1, 2] in a database enables automatic execution of operations when specified events occur and particular conditions are met. Active databases enable important applications, such as alerting users that a certain event of importance has occurred, reacting to events by means of suitable actions, and controlling the invocation of procedures. Most of the research efforts on incorporating this functionality have focused on active capabilities in the context of relational database systems [2]. However, due to the nature of streaming data, pointed out earlier, active functionality cannot be easily incorporated on DSMS. Old aggregated data needs to be referred to, from time to time, for events evaluation and prove very expensive if the system was to make a disk access for the same each time. Also, the system would be required to handle detection of events on streaming data in real-time which is not an issue dealt with in case of traditional databases.

In this paper, we deal with the problem of referencing the old data to respond to user-specified events in real time. As stated in [6], certain applications require reference to data, not only when it arrives, but also after it is summarized (or aggregated). The work illustrates a monitoring application for which access to the entire historical time series is required. Similarly, for event detection in streaming databases, there could be a need to use the past data for evaluation of events. Consider the field of financial data, where the value of various stocks keeps changing continuously. A user may be interested in re-computation of DowJones Average when any two of IBM, GE or Boeing stock prices change by 1% in an hour during the day. Assuming that the aggregation of the data is done every 10 minutes, the system would be required to compare the values to past data. As another example, consider the problem of monitoring constraints on the data, as declared by the user. They could be of the following types – referential integrity (foreign key), primary key, domain constraints etc. For example, consider two relation schemas R_1 and R_2, such that the attributes of R_1 reference to relation R_2. As new data arrives for R_1, it would be required to check it against attribute values of R_2 to ensure data integrity. This information would have to be retrieved from the disk, which would be very time-expensive. Our performance results show that events (primitive or composite) in DSMS can be detected from the data streams and/or from the aggregated data in near real-time.

Initial work on active databases and time-constraints data management was carried out in the HiPAC project [1]. In this project, an event algebra has been proposed, called SNOOP [3], for defining the primitive and composite events. In [5], the authors propose a real-time event detection method for multi-level real-time systems. There are many other systems, such as ODE[4], SAMOS [12], and Sentinel, address event specification and detection in the context of active databases, however, they differ primarily in the mechanism used for event detection. The *Aurora* [10] builds up a new data processing system exclusively for stream monitoring applications. It provides with a large number of stream operators to work with, from simple stream filters to complex windowing and aggregation operators. The core of the system consists of a large network of triggers. The *OpenCQ* [7] and *NiagaraCQ* [8] systems support continuous queries for monitoring persistent data sets over a wide-area network. OpenCQ uses a query processing algorithm based on incremental view maintenance, while NiagaraCQ addresses scalability in number of queries by using techniques for grouping continuous queries for efficient evaluation.

The rest of the paper is organized as follows. The event model is outlined in Section 2. The system architecture is proposed in Section 3. The event composition and detection in the proposed system is described in Sections 4. The experimental results are discussed in Section 5. Finally, we conclude the paper in Section 6.

2 Event Syntax

An event is defined as a tuple: <event type, event_life_time, event_occ_time, attribute list>. *Event type* defines the name of events which share a common system defined meaning specified by the *eid. Event-life-time* is the time for which the occurrence of this event is of importance and *event-occ-time* is the time at which the event occurs. *Attribute list* is a flat list of typed values which carry further information about the event.

An event E (either primitive or composite) is formally being defined as a function from the time domain onto the boolean values, True and False.

$E : T \rightarrow \{\text{True, False}\}$

given by E = True *if an event of type E occurs at time point t*, False *otherwise*. The following operators are used in our system for composing primitive events.

There are two kinds of events defined – primitive and composite. The most common primitive events involve modifications to the data that occur through commands like *insert, delete, update,* etc. in relational database systems and through method invocations in object-oriented database systems. Temporal events are the other type of frequently used primitive events. More advanced systems allow the user to register compositions of such primitive events too. As mentioned above, lot of work has been dedicated to evolve event algebras that would capture the necessary compositions of events and their efficient detection. Figure 1 gives the BNF syntax of the composite event used in our system; the consequent sub-sections will describe the operators and their semantics, followed by the strategy adopted for event detection in the system. We adopt SNOOP [4] as an *Event Specification Language (ESL)* that allows specification of database, temporal, explicit and composite events.

```
composite_ev ::= <element_ev><event_op><composite_ev><time_constraint>
element_ev ::= <primitive_ev> | <atomic_condition_ev>
primitive_ev ::= <basic_update_ev> | <temporal_ev>
time_constraint ::= till<absolute_time> | in<time_span>
atomic_conditon_ev ::= <attribute_name><composite_op><value>
basic_update_ev ::= <db_op> | <ext_signals>
temporal_ev ::= <abs_time> | <interval_time> | <rel_time>
event_op ::= AND | OR | ANY | SEQ | NOT | A | P
db_op ::= UPDATE | INSERT | DELETE
time_span ::= n seconds | n minutes | n hours | n days
```

Fig. 1. BNF syntax of Event Algebra

3 System Description

In this section, we describe the proposed architecture of the event composition and detection in a data stream management system as shown in Figure 2. The detection of events is done by two separate monitoring routines, by *Event Checker* on streaming (queued) data and by *Trigger Messenger*, database inbuilt triggers on summarized data. The data is first buffered in the queue and then summarized/aggregated using application specific algorithms after a fixed interval of time or after a specified number of data points have arrived. The summarized information is then inserted into the persistent storage of the system, marked as *DB2* in the figure. When a new event is defined, the *event parser* sends the correct event definitions to the *event manager* to be stored for later retrievals.

3.1 Event Manager

The event manager stores the structural information of the events specified by the user. An *Event Specification Language* is used that allows specification of database, temporal, explicit and composite events. In our system implementation, we define events using SNOOP as event algebra [3].

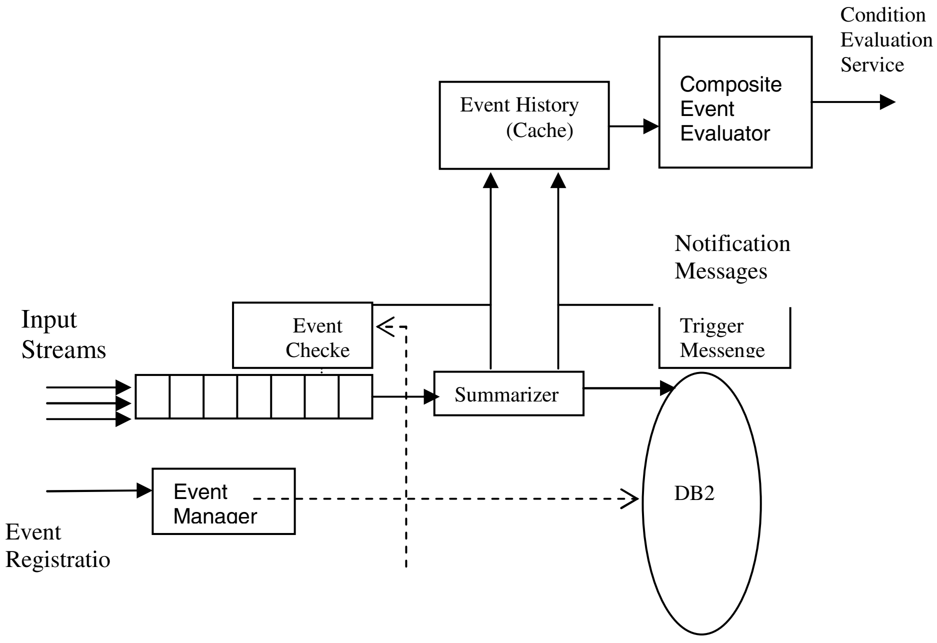

Fig. 2. Architecture of Event Composition and Detection in DSMS

When a new event is registered with the system, the event definition is extracted and corresponding event handler is initialized. If the component(s) of the event is already known to the system as triggering event(s), then the new event subscribes to it and if it is known as a triggering event, then the corresponding mechanisms for its detection are initialized in the *Event Checker* and triggers are set in *DB2*. The structural information of the event is also sent to *Composite Event Evaluator* where the occurrence of the composite events will be detected.

3.2 Data Summarization

The streaming data from the application is initially buffered in the *Data Queue* and then data is summarized. The data summarization can be time-based or data-based, i.e. it could be done after fixed intervals of time or after the arrival of a fixed number of data points in the stream. For example, a stock market may record the average value of stocks after every 10 minutes, irrespective of the number of times the value changes in that period, or the average of every 20 values can be stored. The definition of data summarization can be seen as computing materialized views. These views are then incrementally maintained as new data is summarized.

Considerable amount of work has been done in developing techniques for data reduction and synopsis construction – *sketches, random sampling, histograms, wavelets* to name a few [7, 9, 11]. Gilbert et al. have proposed QuickSAND: Quick Summary and Analysis of Network Data which builds compact summaries of network traffic data called *sketches* based on random projections. These sketches are much smaller in size and respond well to trend-related queries or to features that stand out of data. Network data can also be summarized incrementally at multiple resolutions to answer point queries, range queries and inner product queries using SWAT [11]. Gibbons and Matias have proposed two sampling based summary statistics of data – concise samples and counting samples. These are incremental in nature and more accurate compared to other techniques. The samples were actually created to provide approximate answers to hot list queries. The data summarization techniques are application-specific and hence the system would choose them according to the type of data that it must deal with. The technique selected should be such that the summary created should be amenable to answering queries and take only permissible amount of processing memory.

3.3 Event Cache

Monitoring is a continuous activity and lasts for a long period of time. For any monitoring system, there would be an upper bound on its memory requirements. If the system was to go on saving information about all event occurrences or partially completed events, the available memory space would be soon exhausted. Such situations can arise from very simple kind of events. Consider the event defined $E_1;E_2$ i.e. trigger is raised every time event E_2 occurs after event E_1 has occurred. It could happen that there are multiple occurrences of E_1 before a E_2 occurs. The system should not go on saving all these occurrences of E_1 blindly, but make use of some policy for the same to discard the irrelevant ones.

To deal with the problem of memory usage, we define an event-life time for every event, after which the event is considered *dead* for future consideration. This time must be user-defined, else the system-default is taken. Other solutions for the same could be to store only the recent-most occurrence of the event type, rejecting the older, valid occurrences or to permit one occurrence in one solution only. Whereas the former would function in a similar fashion as using the *recent parameter context* described in [3], the latter will not solve the problem in cases such as the example above.

When a cache entry is made with a new event, time for its removal from the cache is determined using the following equations:

$t_{life} = t_{occ} + time\_span$ *OR* $t_{life} = abs\_time$

The cache makes periodic scans of the event entries for clean-up actions and removes events with older t_{life} than the present system time.

4 Event Detection

This section deals with the specific strategies adopted by the system for event detection. The steps involved are detection of the primitive events, collection of all occurring events, composition of the same to detect complex events and de-registration of events which are not of interest any longer. We describe below all the steps in detail one by one.

4.1 Basic Event Detection

When a user registers a new event with the system, the following actions take place:

- *Assignment of a unique event identifier (eid)*
  - A new event registered with the system is assigned with an eid. The eid is used to identify this class of events in future.
- *Parsing of the event*
  - The Event Parser reads and parses the new event registered and decomposes it to sub-events such that each of them is a primitive event.
- *Update of events monitoring routines*
  - The sub-events are also assigned eids and sent to event monitoring routines *DB2* and *Event Checker* for monitoring data for their occurrence.

Monitoring applications could be hardware monitoring or software monitoring or hybrid of the two. Our monitoring technique is strictly a software monitoring system thus saving on cost and assuring portability across different platforms. Disadvantages are that since no special hardware is being dedicated to the process, it would be sharing the resources with the rest of the system.

The event recognition is done by dedicated monitoring routines at two levels – low level recognition and high-level recognition. The low level event recognition involves detection of primitive events and high-level handles detection of composite events. A special monitoring component – *Event Checker* is responsible for monitoring events in the new, arriving data, whereas the events on the summarized data are checked by setting appropriate *triggers* on the database. *Composite Event Evaluator (CEE)* is dedicated for high-level event recognition once the primitive events are detected. Events are detected on the summarized as well as on the streaming data.

- *Events on summarized data*

We make use of inbuilt triggers in the database to detect events on the summarized data. This is called *Synchronous monitoring* of events since an event occurrence is communicated explicitly to and in synchronization with the user. The triggers are installed on the data columns or objects of interest as specified in the sub-events.

- *Event on streaming data*

For monitoring the arriving data for events, we make use of *system-controlled polling* with system-defined interval where the system checks for the occurrence of the event every interval-time. We detect the events of interest by comparing two snapshots generated by two different polling time points, find out the changes i.e. the inserts, deletes and updates that have taken place.

4.2 Event Notification

The occurrence of primitive events needs to be reported to the *CEE* for detection of composite events. The event is packed with t_{occ}, t_{life} and specified attributes into a packet and sent to the *Event Cache* using message queues. Databases like *DB2* have inbuilt support for message queues (*Websphere Message Queues)* which can be exploited directly. The arrival of the events at the *Cache* needs to be co-ordinated since they would be coming from two asynchronous sources – *database* and *Event Checker.* This can be done by either setting priorities or by using semaphores to enforce mutual exclusion while writing to the *Event Cache.*

The events information is picked up by *CEE* from the *Cache* for subsequent evaluation of composite events. The details about the composition of events are given in the next sub-section. The cache must only keep information about relevant events, which would contribute to event detection in future. Each event is marked with a timestamp indicating the time when it occurred. With the knowledge of the validity interval of an event, which is either user specified or system specific, the old events are removed from the cache.

4.3 Composition of Summarized and Streaming Data Events

Event Algebra provides the necessary expressive power and allows composition of events. Composite events, though more complex in nature, are more useful to the user. The *Composite Event Evaluator* stores the structural information of the registered composite events as well as the data-structures needed to describe the different event-types. The approach taken for event composition is different from earlier works of Ode [4] and SAMOS [12]. SAMOS defines a mechanism based on Petri Nets for modeling and detection of composite events for an OODBMS. They use modified colored Petri nets called SAMOS Petri Nets to allow flow of information about the event parameters in addition to occurrence of an event. It seems that common sub-expressions are represented separately leading to duplication of Petri Nets. Also the use of Petri nets limits the detection of events in chronicle context only. Ode used an extended finite automaton for the composite event detection. The extended automaton makes transitions at the occurrence of each event like a regular automaton and in addition looks at the attributes of the events and also computed a set of relations at the transition. The definition of 'AND' operator on event histories does not seem to produce the desired result; the automaton for the operator constructed according to the specification given by [5] does not seem to reach an accepting state. Since most of the operators in Ode are defined in terms of the 'AND' operator, it makes their semantics also questionable.

We use event trees to store the structure of registered composite events. The composing primitive events are placed at the leaves of the tree whereas the internal nodes represent the relevant operators. The information about the occurrences of primitive events is injected at the leaves and flows upwards. The advantages of using event trees over the previously mentioned methods is that in case of common sub-events, event trees can be merged together and hence reduce storage requirements. For example, let events be A::= E0 AND (E1;E2), B::= (E1;E2) OR E3 and C::= E2;E3. Clearly E1;E2 is common to events A and B and hence their trees can be coalesced. Also, the event trees can be used to detect events in all four parameter contexts.

There could be two distinct ways of keeping the event trees – as a single, consolidated structure for all events, or as specialized graphs, one for each event. A single event graph minimizes on redundancy but makes garbage-collection difficult. Specialized graphs carry an overhead due to multiple copies of the structure but make garbage collection very simple. There are advantages and disadvantages to both and the choice would depend on the application. The choice of the kind of tree used for storage would depend on the application and resources available. The algorithm followed for composite evaluation using event trees is given in Figure 3.

As mentioned earlier, the system must also carry out *garbage collection* to prevent the system from getting clogged with semi-composed events. Hence, a validity interval must be associated with each composite event, either user-specified or system-defined.

```
ALGORITHM Composite Event Detection
    Construct an event graph for each rule with nodes as operators and leaves as
    primitive events. The primitive event nodes are the source and the rule nodes
    are sinks. Edges are from constituent events to composite event.

    For each occurrence of a primitive event
            Store its parameter in the corresponding terminal node 't';
            activate_terminal_node(t);

PROCEDURE activate_terminal_node(n)
    For all rule-ids attached to the node 'n'
            signal event;
    For all outgoing edges i from 'n'
            propagate parameters in node 'n' to the node_i connected by edge i
            activate_operator_node(node_i);
    Delete propagated entries in the parameter list at 'n'
```

Fig. 3. Algorithm for event detection using event trees

4.4 Deregistration of Events

After some time, the user may no longer wish to be notified of certain events. Hence arises the need for facility of deregistration of events – removal of the event structure from the *CEE*. If the event was a primitive one, then it requires a check if any other event is subscribed to it. If not, then the event can be removed from the triggers of the database and from the monitoring routines, else only the subscribing information will be notified.

5 Performance Results

The system was tested for simulated real-time click-stream data using web logs from IIT-CSE web-site (www.cse.iitd.ernet.in). The *access_log* from the web server was used to create continuous messages from *dataGenerator.java*. The *Threads.java* calls the Event Manager which allows user to register events that have to be monitored for. The information about the events registered is also sent to the Receiever, which form the event trees for event composition. It goes on to set the corresponding triggers on the DB2 and set monitoring routines from them which would check for the events on the queued data. As and when an event is detected on either of these, a message is sent to *Receiver.java* about the same. The Receiver maintains the event trees and computes the events following the receipt of sub-events and notifies the user on detection of any registered events.

The system was registered with 20 composite events and tested on a month log data. The number of events missed vs. the window size (number of tuples after which summarization is done), was plotted. There was a slight degradation in the accuracy of detecting events, with the change in window-size from 50,70 to 100. The miss rate went up from 0.68% to 0.693% which is almost ignorable. However, this was accompanied with reduction in memory space used by the database. The following Figure 4 depicts the same.

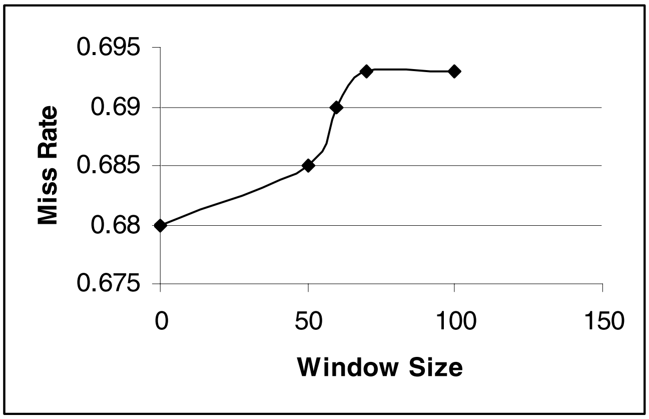

Fig. 4. Miss Rate (of events) vs. Window Size used for summarization

6 Conclusions

This paper has described an architecture and algorithms for detecting composite events in a data stream management system. These events can trigger active rules defined over data streams for making the system reactive. These rules are triggered based on the events detected on the data stream, or events detected while summarizing the data or combination of both. Integration of active rules in data stream management system is important in many real-time applications, such as monitoring a single portfolio that has equities from several stocks exchanges, monitoring the fraud transactions, etc. In this paper, we have described the event model considered in our system implementation and discuss the functionalities of each component of the architecture. We have discussed the various approaches for summarizing the data and then how to notify the events, if generated from this summarization to the composite event evaluator for composing the events. We have done some experiments to measure the miss rate of these events with respect to the varying window size. We have observed that the miss rate is almost negligible as we increase the window size.

References

1. S. Chakravarthy et al. HiPAC: A research project in active time-constrained database management – final technical report. Technical Report XAIT-89-02, Reference Number 187, Xerox Advanced Information Technology, July 1989.
2. U. Schreier, H. Pirahesh, R. Agarwal, and C. Mohan. Alert: an architecture for transforming a passive DBMS into an active DBMS. In *Proc. of the 1991 Intl. Conf. on Very Large Data Bases,* pages 469-478, Sept. 1991.
3. S. Chakravarthy and D. Mishr. Snoop: An Expressive Event Specification Language for Active Databases. *University of Florida CIS Tech. Report,* Sept. 1991.
4. N. Gehani, H. V. Jagadish, and O. Shumeli. Composite Event Specification in Active Databases: Model and Implementation. In Proc. *18th International Conference on Very Large Data Bases,* pages 100-111, Vancouver, Canada, 1992.
5. 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 Largee Databases (VLDB94),* pages 606-617, September 1994.
6. P. Bates. Debugging Heterogeneous Distributed Systems Using Event-Based Models of Behavior. *ACM Transactions on Computer Systems*, 13(1):1-31, February 1995.
7. L. Liu, C. Pu, and W. Tang. Continual queries for internet scale event-driven information delivery. *IEEE Trans. On Knowledge and Data Engineering*, 11(4):583-590, Aug. 1999.
8. J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *Proc. of the 2000 ACM SIGMOD Intl. Conf. on Management of Data*, pages 379-390, May 2000.
9. S. Babu and J. Widom. Continuous queries over data streams. *ACM SIGMOD Record*, 2001(3):109-120.
10. D. Carney, U. Cetinternel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams – a new class of data management applications. In *Proc. 28th Intl. Conf. on Very Large Data Bases*, Hong Kong, China, August 2002.
11. Bulut and A. K. Singh. SWAT: Hierarchical stream summarization in large networks. In *IEEE International Conference on Data Engineering*, page to appear, 2003.
12. S. Gatziu and K. Dittrich, 'Events in an Active Object-Oriented Database', In Proceeding of the 1st International Workshop on Rules in Database Systems, Springer-Verlag, pages 23-39, 1994.