Event Mining with Event Processing Networks

Louis Perrochon, Walter Mann, Stephane Kasriel, and David C. Luckham

Computer Systems Lab, Stanford University, Stanford, CA 94306, USA
`http://pavg.stanford.edu/`

Abstract. Event Mining discovers information in a stream of data, or events, and delivers knowledge in real-time. Our event processing engine consists of a network of event processing agents (EPAs) running in parallel that interact using a dedicated event processing infrastructure. EPAs can be configured at run-time using a formal pattern language. The underlying infrastructure provides an abstract communication mechanism and thus allows dynamic reconfiguration of the communication topology between agents at run-time and provides transparent, location-independent access to all data. These features support dynamic allocation of EPAs to machines in a local area network at run time.

1 Introduction

Event mining (EM) delivers knowledge about a complex system in real-time based on events that denote the system's activities. A system can be anything from a single semiconductor fabrication line to the interconnected check-out registers of a nation-wide retailer. Such systems may be probed to produce events as the system operates. Events are then mined in a multitude of ways: Unwanted events are filtered out, patterns of logically corresponding events are aggregated into one new complex event, repetitive events are counted and aggregated into a new single event with a count of how often the original event occurred, etc. This mining process of producing fewer "better" events out of many "lesser" events can be iterated. The presentation of the mined events to the user is virtually unlimited. EM is particularly well suited for event based systems, but is applicable to other systems as well, e.g. updates in a database can be interpreted as events. The following two applications are typical examples of EM:

Business applications: EM based real-time decision support systems constantly gather information from throughout the enterprise and immediately respond to changes in information. These systems are business event driven, where a business event represents any significant change in business data or conditions.

Enterprise network and systems management: Event patterns that may lead to a failure (e.g. an important disk filling up) or that could signal break-in attempts (i.e. connect requests to multiple targets from a single source over a short time) are detected as they occur. EM provides immediate notification of such conditions to the managers of large, mission critical networks. Automatic prioritizing of alerts and quick root cause analysis leads to reduced response time, higher up-time and allows network managers to quickly respond to critical situations.

In order to understand complex systems and efficiently deal with complex patterns of events, logging with just a time stamp is often not enough. The two following features greatly increase the power of EM:

Complex event structures: Events should be stored as complex objects together with relationships among them instead of just tuples in a relational sense. EM should support event relationships beyond time, e.g. causality: one event causes another. In today's networked real-time environments events come from multiple independent sources and not all events are ordered in respect to each other. If such a natural partial order of events is implicitly reduced to a total order in logging, information is lost and non-determinism is introduced [1].

Flexibility: Because EM happens in real-time, queries need not be hard coded, but be must be flexible, and configurable at runtime. It should be possible at any time to start a new query against an ongoing event stream, that either considers only new events, only old events, or both.

EM supporting these two features is part of Stanford University's RAPIDE project. We developed an extensive set of tools that supports logging, mining, storing, and viewing of events in real-time. RAPIDE events are related by time and cause. Each relation builds a partial order on all the events. A formal pattern language [2] supports the construction of filters and maps, constructs that aggregate simple events to complex events on a higher level of abstraction [3]. The same process can be used to query complex events, thus building a more and more abstract view of the system. Our tools are implemented and available for Sun/Solaris 2.6 and Linux and can process several hundred events per second on an Ultra 1. We are currently negotiating with pilot users in industry.

2 Event Processing Networks

The RAPIDE EM technology is based on the concept of Event Processing Networks (EPNs). Such networks consist of any number of Event Processing Agents (EPAs), namely event sources, event processors and event viewers. Fig. 1 shows an overview over the three categories, with thin arrows indicating the (logical) flow of events from sources through processors to viewers.

Event sources in our applications are typically middleware sniffers. The system middleware can be pure TCP/IP, an event communication service based on a proprietary protocol like TIBCO Inc.'s TIB or Vitria, Inc.'s Communicator, or a military standard like the MIL STD 1553. We also automatically instrument the source code of system written in Java to intercept events within the Java engine [4]. Typical examples for event processors are filters and maps. Filters pass on only a subset of their input, maps aggregate multiple events in the input to

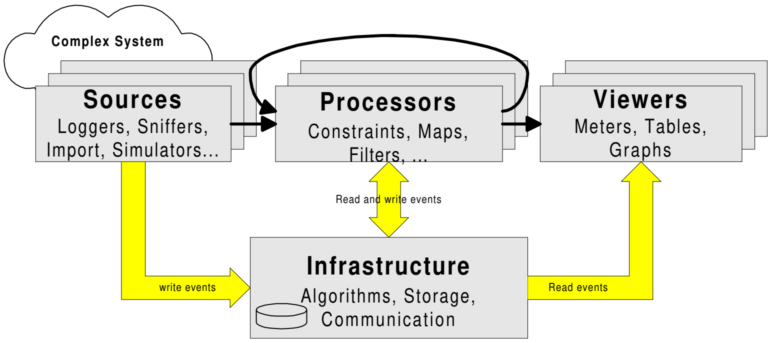

Fig. 1. Event Mining (EM)

output events thus generating events on a higher level of abstraction. Any third party event processor can be inserted into an EPN allowing for the integration with other approaches. Typical event viewers are a graphical viewer for partially ordered sets of events, a tabular viewer of event frequency or a simple gauge metering the value of an important parameter.

Data needs to be stored persistently because agents may want to access past events, even long after they have happened. Also the number of objects currently under consideration may easily exceed the size of the available main memory, thus EM requires some way of storing objects temporarily to disk. RAPIDE EM includes a shared data store that keeps track of all the objects. New objects are written into the data store from where agents and viewers read them. A communication service notifies other EPAs when new objects are added.

Events flow through the EPN in real-time and are displayed in viewers as soon as they are created, limited only by the speed of the underlying infrastructure. Processed events are displayed in viewers shortly after the underlying events have been created by the event source. EPNs are dynamic in that all EPAs can be added and removed at runtime. Newly added agents can either ignore all previous events and just start with the current event at the time they are added, or they can try to catch up all events from the beginning. As EPNs are distributed, EPAs can reside on machines distributed across a network.

3 Real Time Pattern Queries

The RAPIDE pattern language allows the user to describe patterns of events. A RAPIDE pattern matcher searches for all occurrences of a pattern of events in a partially ordered set. A typical example would search for all A events that cause both a B and a C event, with B and C independent of each other. In RAPIDE this pattern could be specified as: $A \rightarrow (B \sim C)$. In OQL, clumsily enhanced with a * operator denoting one or several repetitions of the path expression, this query would look like:

```
select tuple(e.ID, f.ID, g.ID)
from event e, e.(successor)* f, e.(successor)* g
where e.type='A' and f.type='B' and g.type='C' and
 (NOT f.(successor)*=g) OR (NOT f.(successor)*=g) OR f=g)
```

Executing this query from scratch whenever a new event is added to the set is very inefficient, as potentially the whole set has to be traversed. Also, computing the complete transitive closure of the successor relation as a derived relationship is not feasible in a real-time setting. The algorithms we use instead were originally inspired by [5]. Our pattern matching algorithm searches whatever it can on the available data and keeps partially completed results around if possible. When new data arrives, only the partial results which might possibly benefit from the new event need to be reinvestigated.

We call this process incremental query execution or *incremental queries*. Incremental queries return new hits as new records are inserted and optimize re-execution of an ongoing query when new objects are inserted. This optimization is a trade off between storing all partial results on the one hand and rebuilding all partial results on the other hand. Incremental queries are similar to materialized views with real-time constraints. For our purpose, we can think of incremental queries as a repository of ongoing queries, along with some state information on these queries. Every time a new event is added, the queries of this repository would be allowed to run on that element only and the requesting client would be notified if there are new hits. Incremental queries require two interdependent modules:

Notification (call-backs, triggers) that notifies interested clients of insertions and updates on specific objects in the database, and

Dynamic Adaptation that modifies the current query execution plan depending on the newly inserted object and runs the query. This must be done efficiently, e.g. the query tree should be executed in such a way that a minimal amount of work is redone. We believe that these two elements are useful even beyond implementing pattern matching for EM: e. g. for rule processing in real time expert systems [6].

The commercial OODBMS that we looked at had very limited support for notification: most of them require polling the database for new events. With polling, throughput does not scale with the size of the database because searching time for any new object is not constant. One way out is to partition the database. However, big partitions do not help much, and small partitions increase the number of partitions which adds the overhead of keeping track of them. Worse, references between partitions are slower than references within a partition, reducing throughput. In addition, polling has to be done by all readers individually, increasing the load in an event processing network. Overall, our experiments with polling lead to a throughput of only a few objects per second. Hence having readers poll the database for new events is completely unrealistic for our purposes. Only one of the commercial OODBMS we looked at offered call-backs (triggers) on certain changes in the database, but so slow that we could only hope to notify a few objects per second. But clearly, efficient call-backs with minimal delay is critical. To support our requirements, we added our own notification mechanism.

4 Mining Event Patterns

Given an infrastructure for building large databases of events and their temporal, causal, and data attributes, along with a formal pattern language for expressing relationships between events in a compact and expressive way, then event mining is the process of extracting patterns from large sets of events in real time.

Our initial experiments in this area focus on using statistical analysis of stored relationships between events (causality, equivalent data parameters) to identify common yet complex behaviors implied by the events.

The patterns extracted via event mining may then be used to initiate further event processing. For example, they may be used to filter out normal event behavior of a system, so that variations of it may be examined. Or, the patterns extracted may be aggregated into higher level events, to allow views of the event activity at a more abstract level.

A critical factor for real-time event mining is the need to process each new event in constant time. Otherwise incoming events will eventually start to queue up and lead to a big back log. Heuristic methods that are effective and efficient enough are one area of future research.

References

[1] Pratt, V. R. Modeling concurrency with partial orders. Int. J. of Parallel Programming, 1986. 15(1): p. 33–71.

[2] RAPIDE, Rapide 1.0 Pattern Language Reference Manual. Stanford University, Stanford, 1997.

[3] Luckham, D. C. and Frasca, B., Complex Event Processing in Distributed Systems. Computer Systems Laboratory Technical Report CSL–TR–98–754. Stanford University, Stanford, 1998.

[4] Santoro, A., et al., eJava - Extending Java with Causality. In Proceedings of 10th International Conference on Software Engineering and Knowledge Engineering (SEKE'98). 1998. Redwood City, CA, USA.

[5] Fidge, C. J., Timestamps in message-passing systems that preserve the partial ordering. Australian Computer Science Communications, 1988. 10(1): p. 55–66.

[6] Wolfson, O., et al., Incremental Evaluation of Rules and its Relationship to Parallelism. In Proceedings of SIGMOD'91 Conference on the Management of Data. 1991. Boulder, CO: ACM Press.