state information required by each aggregation function for the group. For each tuple, the algorithm probes the hash table to find the entry for the group to which this tuple belongs and update the state information. If such entry does not exist, the algorithm creates a new hash entry and initiates the state information.

If the relation is so large that the hash table does not fit in memory, the algorithm hash partitions the relation on the grouping attributes. Since all tuples in a given group are in the same partition, the algorithm can then scan each partition independently and compute aggregation functions as described before.

It is also possible and sometimes preferable to compute aggregation using an index as long as the index covers all the attributes required by the aggregation query. The advantage is to read in a much smaller index compared to the entire relation. If the grouping attributes form a prefix of the indexed keys, the algorithm can avoid the sorting step and scan the index entries sequentially.

Key Applications

Each database query is composed of a few relational operators. The final query plan may implement each relation operator in different ways to achieve an optimal performance. All the implementation techniques are widely used in database systems.

Cross-references

- ► Access Methods
- ► Buffer Pool Management
- ► Concurrency Control
- ► Cost Model
- ► External Sorting
- ► Hashing
- ► Parallel Query Processing
- ► Query Optimization

Recommended Reading

1. Graefe G. Query evaluation techniques for large databases. ACM Comput. Surv., 25(2):73–170, 1993.
2. Mishra P. and Eich M.H. Join processing in relational databases. ACM Comput. Surv., 24(1):63–113, 1992.
3. Ramakrishnan R. and Gehrke J. Database Management Systems. McGraw-Hill, New York, 2002.
4. Selinger P.G., Astrahan M.M., Chamberlin D.D., Lorie R.A., and Price T.G. Access path selection in a relational database management system. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 1979, pp. 23–34.

Evaluation of XML Retrieval Effectiveness

► Evaluation Metrics for Structured Text Retrieval

Event

► Time Instant

Event and Pattern Detection over Streams

Mingsheng Hong, Alan Demers, Johannes Gehrke, Mirek Riedewald
Cornell University, Ithaca, NY, USA

Synonyms

Complex event processing (CEP); Event stream processing (ESP)

Definition

An event is a basic unit of information in streaming data. An event pattern is a combination of events correlated over time. Event pattern detection is an important activity in complex event processing. In this setting, the matches to the event patterns are referred to as complex events.

Historical Background

In the early 1990s, a set of pioneering work in *event systems*, such as SNOOP [3] and ODE [8], set out to define query languages for expressing event patterns. In these proposals, the data model for expressing events is not fixed. More recently, the approaches proposed by Cayuga [1,5,6] and SASE [14] for event pattern detection align more closely to relational query processing, in that each event is modeled by a relational schema, and some of the operators for expressing event pattern queries are drawn from relational algebra. Regardless of the data model for events, these systems all use some variant of NFA (Nondeterministic Finite state Automaton) as the processing model.

With the advent of internet-scale message brokering systems, *content based publish-subscribe systems*

such as [7] emerged. They are characterized by very limited query languages, allowing simple selection predicates applied to individual events in a data stream. Such systems trade expressiveness for performance – when well engineered, they exhibit very high scalability in both the number of queries and the stream rate. However, their inability to express queries that span multiple input events makes them unsuitable for event pattern detection.

Another category of systems closely related to event pattern detection is *stream databases*, such as Aurora [2], STREAM [10], and TelegraphCQ [4]. Contrary to content based publish-subscribe systems, they focus on expressiveness. Such systems have very powerful query languages, typically including a rich functionality and extending SQL with provisions for sliding window and grouping features. Though powerful, stream databases are not designed for detecting event patterns, and their query languages can be awkward for expressing event pattern queries. Moreover, there is little work on scaling up stream database engines with the number of concurrent queries that are reasonably sophisticated.

Foundations

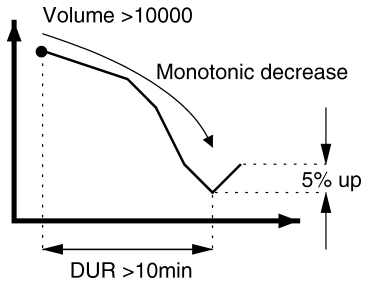
Event Pattern Query Model

An event stream is a potentially infinite sequence of events. Each event has a timestamp value, and its payload content is encoded by a relational tuple. Events are processed in the timestamp order by an event processing system. They can be filtered, transformed, and correlated to form event patterns.

The computational model of matching event patterns over event streams naturally extends that of matching regular expression patterns over character streams in the following aspects. First, each stream event can contain multiple attribute-value pairs conforming to a relational schema, where the value domains are potentially infinite. In comparison, each character in a character stream provides for the same single attribute a value drawn from a finite alphabet. Second, given the multiple attributes for each stream event, event patterns can perform relational unary operations on the events, including filtering, projection and renaming of attributes. This provides expressive power especially to event correlation based on sequencing, where the attribute values of multiple stream events can be compared. Finally, event pattern detection involves a temporal aspect, in that stream events have timestamps, which event patterns reason about. In comparison, regular expression processing only involves character orderings in the streams, which can be viewed as a weaker notion of time.

Event patterns are usually expressed in an event algebra. Many such algebras have been proposed. One representative is the Cayuga algebra [5]. The Cayuga algebra is specifically designed for large-scale event pattern detection. In addition to unary operators for selection predicates and aggregates, the expressive power of this algebra comes from two binary operators. The first binary operator, *sequence*, can correlated two input events based on a join predicate. This join predicate involves timestamps, and can optionally involve other attributes in the stream schema. The second operator, *iteration*, is an generalization of the sequence operator. It is able to produce an event pattern involving arbitrarily many input events by iteratively concatenating input events with the pattern built so far. To allow users to interact with the system in a user-friendly way, a SQL-style query language, referred to as Cayuga Event Language (CEL) [6], has been developed. Figure 1 shows an event pattern query expressed in CEL. This event pattern query

```
SELECT Name, MaxPrice, MinPrice, Price AS FinalPrice
FROM
  FILTER{DUR > 10min}(
    (SELECT Name, Price_1 AS MaxPrice, Price AS MinPrice
    FROM FILTER{Volume > 10000}(Stock))
      FOLD{$2.Name = $.Name, $2.Price < $.Price}
    Stock)
    NEXT{$2.Name = $1.Name AND $2.Price > 1.05*$1.MinPrice}
  Stock
```



Event and Pattern Detection over Streams. Figure 1. Event pattern query to find stock price pattern.

corresponds to a particular trend in stock prices. Intuitively, this query searches for the given price pattern for any company. The pattern starts with a large trade (`Volume > 10,000`), followed by a monotonic decrease in price (`FOLD` clause), which lasts for at least 10 min (`DUR > 10min`). Then the price rebounds with a sudden increase by 5% (`NEXT` clause). The `NEXT` and `FOLD` constructs respectively correspond to the two binary operators in the Cayuga algebra introduced above. `NEXT` matches the next event in the stream that satisfies a given condition (same company name and 5% higher price in the example). `FOLD` is the iterated version of `NEXT`, i.e., it continues matching the next event that satisfies a certain property until a stopping condition is satisfied.

SASE [14] uses a query algebra similar to the Cayuga algebra, where sequence and iteration are the key primitives. SASE also supports *negation*-style event patterns, where a pattern is matched by the absence of an input event, rather than the presence. In addition to online event stream processing supported by Cayuga and SASE, work on sequence database systems has focused on matching event patterns offline over archived data [11,12,13]. Sequence is again the key primitive for expressing event patterns in the SEQ query algebra [13] and the SQL-TS query language [12].

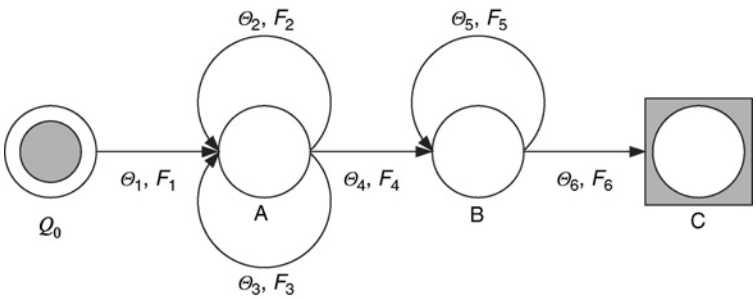
Event Pattern Query Processing

As is mentioned earlier, an event pattern query is typically implemented by a state machine. Continuing the above query example, this approach is described in the context of Cayuga. Each Cayuga automaton is an extension to the classical non-deterministic finite automaton [9] in the following aspects. First, each automaton edge is associated with a predicate, and for an incoming event, this edge is traversed if and only if the predicate is satisfied by this event. This mechanism implements the selection predicates in the event patterns. Second, when patterns are matched, to be able to generate witness events with concrete content instead of boolean answers, each automaton instance needs to store the attributes and values of those events that have contributed to the pattern instance.

Figure 2 shows the automaton for the example query in Fig. 1. The two middle states correspond to the `FOLD` and `NEXT` operators, respectively. The predicates θ_i associated with automaton edges originate either from `FILTER` conditions (θ_1, θ_6 in the example) or from join conditions of the `FOLD` and `NEXT` operators. Specifically, θ_1 implements the filter predicate `vol` $>$ 10,000. When an input stock event e satisfying θ_1 occurs, a new automaton instance I is created under state A, remembering the content of e.

Each automaton instance encodes a particular event pattern built from the prefix of the event stream that has been processed so far. When an automaton instance reaches the final state C, a match to the entire event pattern specified in the query is found, and will be output by this automaton.

To continue the explanation of the example Cayuga automaton in Fig. 2, θ_2 and θ_3 respectively implement the two join predicates associated with `FOLD` in the query shown in Fig. 1. These two predicates on the two self-loop edges associated with state A together build a monotonically decreasing sequence in the prices of a particular stock. Specifically, after the occurrence of event e, when a later stock event e' together with I satisfies θ_2; i.e., e' and I have the same stock name s (say), θ_3 is evaluated to check whether this event pattern built so far can be extended. For this reason, θ_2 serves as a criterion which, when satisfied, concatenates the next input event from `Stock` to the event pattern built up so far, and θ_3 serves as a criterion



Event and Pattern Detection over Streams. Figure 2. Cayuga automaton example (source: [6]).

which, when satisfied, continues the extension of the event pattern being built. The event building process proceeds similarly to state *B* and *C*. The functions F_i are responsible for transforming event stream schemas and automaton state schemas.

Event pattern queries expressed in SASE and SQL-TS are processed in a similar way. In SASE, NFA is one of the run-time operators. Operator re-ordering is performed as part of the query optimization to produce efficient query plans. For example, a selection predicate above an NFA operator can be pushed inside the NFA operator to discard irrelevant input events earlier, thus improving system throughput. In SQL-TS, to optimize pattern search, the query engine exploits the inter-dependencies between the elements of a sequential pattern.

Evaluating one event pattern query efficiently is relatively easy. However, it is challenging to efficiently evaluate a large number of concurrent event patterns. Multi-Query Optimization (MQO) techniques have been developed to share the computation among concurrent event patterns. For example, in Cayuga, the event processing engine achieves this goal by exploiting the relationship of the query algebra to the automata-based query execution, and the commonality among queries. Specifically, each query is first translated into a set of automata. These automata are then "merged" with existing ones in the engine. During the merging process, two optimization techniques are used. First, two automata with the same prefix of states can merge these states, thus sharing computation and storage. This is similar to the technique of finding common subexpressions in relational query processing. Second, some of the filtering predicates on the automaton edges can be managed efficiently by indexes in a way similar to the techniques for processing multiple selection operators [7]. These two techniques enable a throughput of thousands of events per second, even for tens of thousands of active event pattern queries.

Key Applications

Event pattern detection targets a large class of both well-established and emerging applications, including supply chain management for RFID (Radio Frequency Identification) tagged products, real-time stock trading, monitoring of large computing systems to detect malfunctioning or attacks, and monitoring of sensor networks, e.g., for surveillance. These *event monitoring applications* need to process massive streams of events in (near) real-time. There is great interest in these applications as indicated by the establishment of sites like http://www.complexevents.com, which bring together major industrial players like BEA, IBM, Oracle, and TIBCO.

Future Directions

One important future direction is to integrate event processing systems with stream databases. In the past they have evolved along different paths, and are designed for different query workloads, as is described in the Historical Background section. It would be beneficial to integrate these two categories of stream processing systems, for two reasons. First, there is much overlap in their functionality. For example, both categories support stateless operations such as filtering and projection, and some forms of stream join. Second, there are a class of stream applications that demand functionality from both categories. There are two major challenges in the integration. At the logical level, a unified query algebra is needed for expressing queries in both categories. At the physical level, it is desirable to evaluate both categories of queries in the same engine. This is a challenging task since current stream database engines, like relational database engines, are usually based on operator trees, while event engines are based on variants of NFAs.

Cross-references

- ► Active Database
- ► Architectures and Prototypes
- ► Complex Event Processing
- ► Continuous Query
- ► Publish/Subscribe over Streams
- ► Stream Processing

Recommended Reading

1. Brenna L., Demers A., Gehrke J., Hong M., Ossher J., Panda B., Riedewald M., Thatte M., and White W. Cayuga: a high-performance event processing engine. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 2007, pp. 1100–1102.
2. Carney D., Çetintemel U., Cherniack M., Convey C., Lee S., Seidman G., Stonebraker M., Tatbul N., and Zdonik S. Monitoring streams – a new class of data management applications. In Proc. 28th Int. Conf. on Very Large Data Bases, 2002, pp. 215–226.
3. Chakravarthy S., Krishnaprasad V., Anwar E., and Kim S.K. Composite events for active databases: semantics, contexts and detection. In Proc. 20th Int. Conf. on Very Large Data Bases, 1994, pp. 606–617.
4. Chandrasekaran S., Cooper O., Deshpande A., Franklin M.J., Hellerstein J.M., Hong W., Krishnamurthy S., Madden S.R., Raman V., Reiss F., and Shah M.A. Telegraph CQ: continuous

dataflow processing for an uncertain world. In Proc. 1st Biennial Conf. on Innovative Data Systems Research, 2003.
5. Demers A., Gehrke J., Hong M., Riedewald M., and White W. Towards expressive publish/subscribe systems. In Advances in Database Technology, Proc. 10th Int. Conf. on Extending Database Technology, 2006, pp. 627–644.
6. Demers A., Gehrke J., Panda B., Riedewald M., Sharma V., and White W. Cayuga: a general purpose event monitoring system. In Proc. 3rd Biennial Conf. on Innovative Data Systems Research, 2007, pp. 412–422.
7. Fabret F., Jacobsen H.A., Llirbat F., Pereira J., Ross K.A., and Shasha D. Filtering algorithms and implementation for very fast publish/subscribe. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 2001, pp. 115–126.
8. Gehani N.H., Jagadish H.V., and Shmueli O. Composite event specification in active databases: model and implementation. In Proc. 18th Int. Conf. on Very Large Data Bases, 1992, pp. 327–338.
9. Hopcroft J.E., Motwani R., and Ullman J.D. Introduction to automata theory, languages, and computation. Addison-Wesley, Reading, MA, USA, 2nd ed., 2000.
10. Motwani R., Widom J., Arasu A., Babcock B., Babu S., Datar M., Manku G.S., Olston C., Rosenstein J., and Varma R. Query processing, approximation, and resource management in a data stream management system. In Proc. 1st Biennial Conf. on Innovative Data Systems Research, 2003.
11. Ramakrishnan R., Donjerkovic D., Ranganathan A., Beyer K.S., and Krishnaprasad M. SRQL: sorted relational query language. In Proc. 10th Int. Conf. on Scientific and Statistical Database Management, 1998, pp. 84–95.
12. Sadri R., Zaniolo C., Zarkesh A.M., and Adibi J. Optimization of sequence queries in database systems. In Proc. 20th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems, 2001, pp. 71–81.
13. Seshadri P., Livny M., and Ramakrishnan R. SEQ: a model for sequence databases. In Proc. 11th Int. Conf. on Data Engineering, 1995, pp. 232–239.
14. Wu E., Diao Y., and Rizvi S. High-performance complex event processing over streams. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 2006, pp. 407–418.

Event Broker

► Request Broker

Event Causality

Guy Sharon
IBM Research Labs-Haifa, Haifa, Israel

Definition

The definition of an event processing network includes a relation to express un-modeled event processing logic between events.

The fact that events of type *eT1* cause events of type *eT2* is denoted by the relation *causes(eT1,eT2)*.

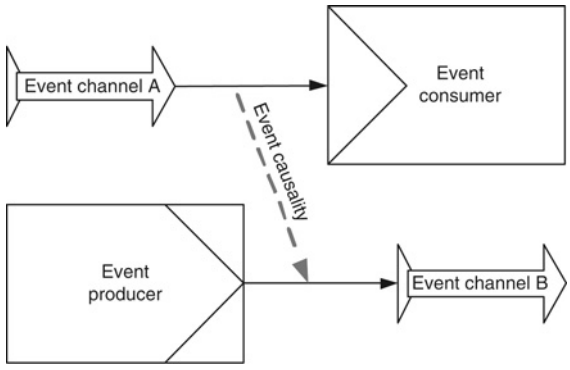
In an EPN where E is the set of edges representing event streams, EC is the set of Event Channels, C is the set of Event Consumers and P is the set of Event Producers, the relation is evaluated to be true if events of type *eT1* flow in an event stream $e1(u,v)$: $e1 \in E$, $u \in EC$, $v \in C$ and events of type *eT2* flow in an event stream $e2(m,l)$: $e2 \in E$, $m \in P$, $l \in EC$ and there is some un-modeled event processing logic between events of type *eT1* consumed by $v \in e1$ and events of type *eT2* produced by $m \in e2$.

Key Points

The event processing intent defined by an event processing network may not cover the entire flow of events through systems as there may be cases where an event is handled by an event consumer, such as a software application, and as a result the application publishes, as an event producer, a new event to an event processing network. In essence, there is some un-modeled processing logic performed by the application that results in a new event that may be processed further on. The specifics of this logic are not important for the realization of an event processing network as this logic is implemented and executed by a consumer such as an application, however, having the relation *causes(eT1,eT2)* as part of the model enables to describe the complete event-driven interactions in systems and is used in performing model analysis such as termination analysis [1–3].

In the model, event causality is represented as a red, dashed, and directed edge from the event stream being consumed to the event stream being produced (Fig. 1).

The definition of event causality in an event processing network is meant to be theory neutral and



Event Causality. Figure 1. Event causality.