Development of an Event Stream Processing System for the Vehicle Telematics Environment

Jongik Kim, Oh-Cheon Kwon, and Hyunsuk Kim

ABSTRACT—In this letter, we present an event stream processing system that can evaluate a pattern query for a data sequence with predicates. We propose a pattern query language and develop a pattern query processing system. In our system, we propose novel techniques for run-time aggregation and negation processing and apply our system to stream data generated from vehicles to monitor unusual driving patterns.

Keywords—Event streams, pattern matching, telematics.

I. Introduction

Due to increasing requests for real-time processing of large amounts of data coming from various data sources, the data processing paradigm has shifted from stored, static, and offline storage to streamed, dynamic, and online data sources.

Many studies have been conducted on stream data processing. To represent a query over streams, stream query languages such as CQL [1] and TelegraphCQ [2] have been proposed. However, they lack constructs to represent nonoccurrence of events. More recently, complex event processing systems over streams, SASE [3] and SASE+ [4], have been proposed, where a data item arriving from a stream is called an event. Event query languages in these systems can address a sequence of events and negation predicates for nonoccurrence of events. However, they suffer from the lack of run-time processing techniques for negation predicates. They can evaluate negation predicates only with the aid of a relational database system.

Manuscript received Feb. 23, 2009; revised May 4, 2009; accepted May 20, 2009.

This work was supported by the IT R&D program of MKE/IITA, Rep. of Korea [2007-S025-02, Vehicle & Driver Management System Technology Development].

Jongik Kim (phone: +82 63 270 4634, email: jongik@chonbuk.ac.kr) is with the Department of Computer Science & Engineering, Chonbuk National University, Jeonju, Rep. of Korea.

Oh-Cheon Kwon (email: ockwon@etri.re.kr) and Hyunsuk Kim (email: hyskim@etri.re.kr) are with the IT Convergence Technology Research Laboratory, ETRI, Daejeon, Rep. of Korea.

doi:10.4218/etrij.09.0209.0087

In this letter, we present an event stream processing system making the following contributions:

i) We propose a query language that extends existing event languages to provide a convenient syntax for the ANY sequence constructor and new semantics for aggregations.

ii) We develop novel techniques for processing aggregations and negations using run-time stacks without an RDBMS.

iii) We apply our technique to the telematics environment.

II. VDMS-PQ

In this section, we briefly introduce our query language named vehicle and driver management system pattern query (VDMS-PQ). We adopt language features, such as negations in sequences and parameterized predicates from the SASE [3], while we uniquely define two aggregation semantics and a convenient syntax for ANY constructor.

VDMS-PQ consists of three parts: PATTERN, WHERE, and WITHIN. PATTERN specifies a sequence of event arrivals from streams; WHERE specifies constraints for events in PATTERN using predicates; and WITHIN specifies a time sliding window. We demonstrate the features of our language using the following two example queries.

```
Q1: PATTERN x:START, NOT(y:START), z:STATUS
      WHERE x.car_id=y.car_id=z.car_id && z.distance> 500
      WITHIN 200 minutes
Q2: PATTERN x:START, y:ANY(STATUS, DTC)
      WHERE x.car_id =y.car_id && x.start_time > 21 &&
      (y.speed > avg*(y.speed) OR y.code = SOME_ERR)
```

Data related to engine start time comes from the data source, START, which includes the driver ID, car ID, and engine start time. The data source, STATUS, generates data related to the

vehicle status such as car ID, speed, RPM, and driving distance. Finally, the data source, DTC, has attributes related to vehicle diagnosis information which comes from a vehicle whenever the vehicle has an error.

Q1 finds vehicles that drive more than 500 km without restarting. The PATTERN clause of Q1 specifies that the engine start information of a car has arrived, no other engine start data of that car is expected, and the status of the car will arrive sometime after. The negated event, NOT(y:START), means a vehicle is not restarted. When this pattern occurs, the driver could be warned of overdriving.

Q2 finds vehicles that start at night and either exceed the speed limit or have an engine error. In VDMS-PQ, we can assign one variable to alternative data sources in an ANY constructor. We can use all the attributes of the data sources in ANY with the variable in the WHERE clause. In the WHERE clause of Q2, we use *avg**, which is an aggregation function that computes the average of all the data items that have arrived so far. By using *avg* instead of *avg**, we have alternative aggregation semantics. Here, *avg* computes the average of only the data items that belong to the query results. We have two types of aggregations in our query language. *avg*, *sum*, *count*, *min*, and *max* are type 1 aggregations, and their * versions are type 2 aggregations.

III. Processing of VDMS-PQ

Processing of VDMS-PQ has the following steps. First, the processor scans input streams and constructs sequences of events that satisfy the sequence in the PATTERN clause. Second, the processor tests each sequence constructed in the first step using the predicates in the WHERE clause. Finally, the processor checks the time window for each result from the second step and generates query results. The final step can be embedded in the first step, and simple predicates that compare constant values in the second step are also optimized into the first step.

Here, we describe a sequence construction method based on the technique proposed in [3]. Then, we propose novel techniques for processing negations and aggregations using the following example queries and data arrivals.

Q3: PATTERN a:A, c:C, d:D

Data arrivals for Q3: a1 d2 c3 a4 c5 d6

Q4: PATTERN a:A, NOT(b:B), c:C, d:D

Data arrivals for Q4: a1 c2 b3 a4 c5 d6

1. Sequence Construction and Negation Processing

Figure 1 shows a stack-based sequence computation for Q3. We assign a stack per each stream source in the PATTERN clause of Q3. When a1 has arrived from the stream, we push the event into stack A. The next event, d2, cannot make a sequence satisfying Q3 because d2 appears before all the events from stream C in temporal order and cannot make a sequence of A, C, and D in the PATTERN clause of Q3. Therefore, we can simply ignore d2 instead pushing it into stack D. Then, c3, a4, c5, and d6 are pushed into the corresponding stacks as in Fig. 1. When an event is pushed into a stack, the event has a link to the top element of its parent stack. For example, c3 has a link to a1 because the top element in stack A is a1 when c3 has been pushed into stack C. Likewise, c5 has a link to a4, and d6 has a link to c5. When an event is pushed into the final stack, stack D, we can construct result sequences by following the link attached to each event. In Fig. 1, sequence [a4, c5, d6] is generated by following links. Because a1 below a4 in stack A arrives before a4, [a1, c4, d6] is another result of the query. Likewise, c3 arrives before c5, and there is one event, a1, in stack A that arrives before c3. Therefore, [a1, c3, d6] is also a result of the query. We can easily compute result sequences by assuming that there is a downward link between adjacent events in the same stack as shown in Fig. 1 as gray arrows.

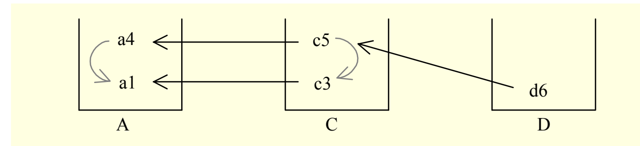

Fig. 1. Run-time stack for Q3.

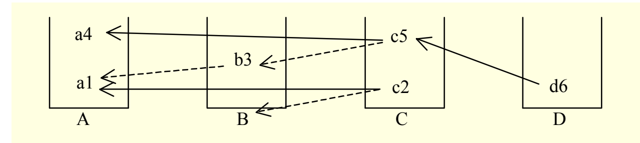

Fig. 2. Run-time stack for Q4.

To process a sequence with negations, we use another link called *neg_link*. Figure 2 shows run-time stacks for Q4 with neg_links in dashed lines. An element in a negation stack (B) has only a neg_link. An element in the child stack (C) of a negation stack has both a link and a neg_link, and an element in other stacks (A and D) has only a link.

When d6 arrives, [a4, c5, d6] is generated as a result. Although [a1, c5, d6] can be another candidate result, we drop the sequence because b3 is between a1 and c5. We can easily find it with neg_link. An element in the child stack of a negation stack can construct result sequences with elements in (neg_link$^+$, link), where '(' denotes exclusion of the first element, and ')' denotes inclusion of the last element. Here, neg_link$^+$ means following neg_link continuously until there is no neg_link. Thus, c5 can be combined with elements in (a1, a4]. When a neg_link indicates the bottom of a negation stack, we can ignore the neg_link (e.g. the neg_link of c2).

Now, we present a processing technique for negations with predicates. We assume that we have already found all the

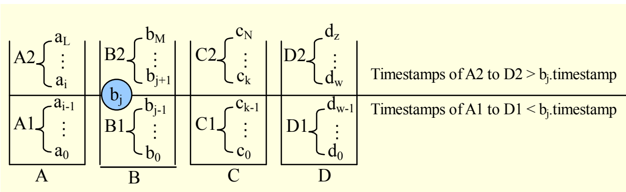

Fig. 3. Run-time stack for Q4 with predicates.

sequences such that there is no B element between A and C, and we do not take neg_links into consideration. Consider that Q4 has a predicate related to the negated variable, b, in its WHERE clause (e.g. WHERE b.attr = 3). We can remove [a1, c5, d6] from the result set only if the predicate is true.

In Fig. 3, if b_j is the element that makes the negation predicate true, we should drop sequences containing b_j and all the sequences that stem from [A1, C2, D2] because the b_j element is between A1 and C2 in the order of arrival time. We can divide negation predicates into two cases: 1) b is related to constants; 2) b is related to other variables (e.g. b.attr1 > a.attr2). Variations of our technique can be applied to these cases.

2. Aggregation Processing

Figure 4 shows the overall aggregation processing of our system. When an event arrives, we aggregate the attribute values of the event for type 2 aggregations. If the event participates in a result sequence, then we aggregate attribute values of the event as type 1 aggregations. We aggregate the event only once for the first result sequence.

When an event is removed from a run-time stack, we move the event into the pending stack. Whenever a new event arrives from a stream, we disaggregate each event in the pending stack whose timestamp is outside the time window and remove it from the pending stack. The dashed line in Fig. 4 shows the disaggregation process.

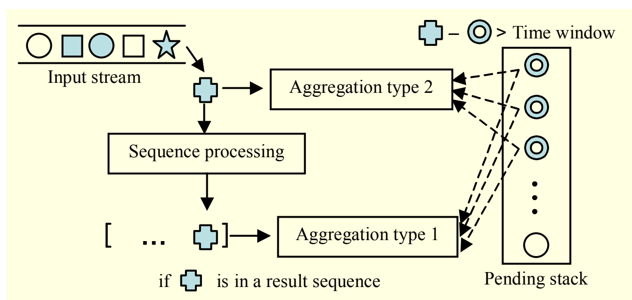

Fig. 4. Aggregation processing scheme.

3. Preliminary Performance Results

In the experiment, we focused on the performance of negation processing and compared our technique with that of SASE [3] and SASE+ [4]. We used the query Q1 and ran experiments varying the input size from 10,000 events to 70,000 events. Because SASE uses an RDBMS for processing negation, it is necessary to insert every event into an RDBMS, which is an expensive operation. Figure 5 shows the experimental results. In Fig. 5, SASE* is SASE without RDBMS insertion overheads. Our technique outperforms SASE about four times. Moreover, our technique is superior to SASE even though we removed RDBMS insertion overheads from the SASE system.

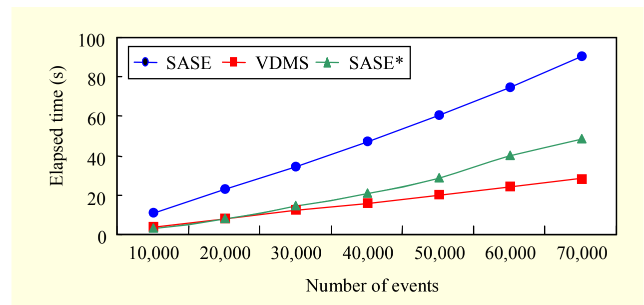

Fig. 5. Experimental results for the query Q1.

IV. Conclusion

We presented a pattern query language for event streams and an event stream processing system. We developed new techniques for negation sequences and aggregations, and we demonstrated its efficiency by experiment. We are continuing to develop our system in the following directions. We will extend our language to be used in general, and SASE+ can be a reference model of the extension of our language. The proposed technique for negation processing is unique and efficient, but it may produce many redundant result sequences, so we will improve negation processing. We plan to apply our technique to postal delivery vehicles and commercial vehicles as a part of the VDMS project.

References

[1] A. Arasu, S. Babu, and J. Widom, "CQL: A Language for Continuous Queries over Streams and Relations," *Proc. Int. Workshop on Database Programming Languages*, 2003, pp. 1-19.

[2] S. Chandrasekaran et al., "TelegraphCQ: Continuous Dataflow Processing for an Uncertain World," *Proc. Conf. Innovative Data Syst. Research*, 2003, p. 668.

[3] W. Eugene, Y. Diao, and S. Rizvi, "High-Performance Complex Event Processing over Streams," *Proc. ACM SIGMOD*, 2006, pp. 407-418.

[4] J. Agrawal et al., "Efficient Pattern Matching over Event Streams," *Proc. ACM SIGMOD*, 2008, pp. 147-160.