Efficient Strategy for Out-of-Order Event Stream Processing

Yingyuan Xiao1*, Tao Jiang1, Yan Shen1 and Huafeng Deng2

1Tianjin Key Laboratory of Intelligence Computing and Novel Software Technology, Key Laboratory of Computer Vision and System, Ministry of Education, Tianjin University of Technology, Tianjin-300384, P.R. China
2College of Information Technology, South China Normal University, Guangzhou-510631, P.R. China

Abstract

Complex event processing has been widely used in many modern applications. A key aspect of complex event processing is to extract patterns from event streams to make informed decisions in real-time. However, network latencies and machine failures may cause events to arrive out-of-order at the event processing engine. To address the problem, a number of disordered event processing techniques are proposed. In this paper, we introduce latency distance and purging time to process out-of-order event streams in real-time. Further, we present a redo strategy based on playback, with which those false pattern matches produced at the early phase can be corrected by the aid of the cloud platform. We conduct extensive experiments, and the experimental results demonstrate the effectiveness of our methods.

Key Words: Out-of-Order Events, Event Stream Processing, Latency Distance, Purging Time, Redo Strategy

1. Introduction

Sensor devices such as wireless motes and radio frequency identification (RFID) readers have been adopted in many modern applications, ranging from supply chain management to monitoring and tracking artifacts. Ever growing deployment of these devices results in a huge volume of events. Event processing over event streams has attracted a lot of attentions recently. Most event stream processing systems [1–3], both event-based and stream-based ones, assume a total ordering among event arrivals, that is, the order in which the events are received by the systems is assumed to be the same as their timestamp order. However, network latencies and machine failures may cause events to arrive out-of-order at the event processing system. Out-of-order arrival of events will result in such circumstances, either missing resulting matches or producing incorrect matches. Let us consider a popular example during tracking books in a bookstore [1,7,10] where RFID tags are attached to each book and RFID readers placed at key locations throughout the store, like book shelves, checkout counters and the store exit. If a book shelf and a store exit read the RFID tag for a book successively but the RFID tag is not read by any of the checkout counters prior to the store exit, then we can conclude that the book is being shoplifted. If the total order assumption (the later arrival of an event implies that it has a larger timestamp than the other events which have already arrived earlier) holds, the event processing system can also draw the same conclusion from the event stream generated from the RFID readers. Unfortunately, the total order assumption does not always hold in real-life scenarios. For example, the RFID tag is read by one of the checkout counters prior to the store exit but the checkout event has the later arrival than the exit event due to network traffic or possible RFID reader failure, then the event processing system may output the wrong alarm in real-time. Clearly, it is imperative for an event processing...
system to deal with both in-order as well as out-of-order arrivals of events efficiently.

Real-time processing of out-of-order event streams is a primary challenge for today’s monitoring and tracking applications. In this paper, we explore the efficient strategy for out-of-order event stream processing. The main contributions of this work include: 1) we present an efficient method for processing out-of-order event streams in real-time by introducing latency distance and purging time; 2) we propose a redo strategy based on playback to correct the false pattern matches produced at the early phase and 3) we conduct extensive experiments that demonstrate the effectiveness of our proposed approaches.

The rest of the paper is organized as follows. Section 2 reviews the related work on out-of-order event processing. Section 3 formally defines the related concepts used in this paper. An efficient method for processing out-of-order event streams in real-time is proposed in section 4. To effectively correct the false pattern matches at the real-time phase, a cloud platform-based redo strategy is presented in section 5. Extensive experiments and evaluations are reported in section 6. We conclude this paper in section 7.

2. Related Work

A number of out-of-order event processing techniques have been proposed [4–12]. In [4], the authors exploit punctuation semantics to view an infinite stream as a mixture of finite streams. Srivastava et al. [5] propose heartbeats to deal with uncoordinated streams. They focus on how heartbeats can be generated. But how heartbeats can be utilized in out-of-order event stream processing is not discussed. Babu et al. [6] propose a method for dealing with out-of-order arrival of events, called K-slack. K-slack buffers the arriving data for K time units. The biggest drawback of K-slack is rigidity of the K that cannot adapt to the variance of the network latencies that exists in a heterogeneous network. For example, one reasonable setting of K may be the maximum of the average latencies in the network. However, as the average latencies change, K may become either too large or too small. Thereby, it either introduces unnecessary inefficiencies and delays, or becomes inadequate for handling the out-of-order processing of the arriving events and produces inaccurate results. In [7–9], authors first provide three criteria of correctness for out-of-order data processing, considering latency, output order and result correctness. Then, they propose two solutions: aggressive and conservative strategies respectively to process sequence pattern queries on out-of-order event streams. Zhou et al. [10] observe that events in many real world applications have durations, and the relationships among these events are often complex. Motivated by this, they propose a hybrid solution including time-interval to solve the out-of-order events, which can switch from one level of output correctness to another. To relax the restriction of ordered input in pattern-detection models, Chandramouli et al. [11] propose a dynamic pattern matching method over disordered streams, which supports native handling of out-of-order input, stream revisions, dynamic patterns, and several optimizations. Further, Chandramouli et al. [12] exploit latency estimation in a distributed event processing system. Different from these methods mentioned above, in this paper, we introduce latency distance and purging time to process out-of-order event streams in real-time.

3. Preliminary

In the section, we formally define the related concepts used in this paper, including ordered event stream, out-of-order event, pattern matching plan, latency distance and purging time.

We use capital letters (e.g., A, E) for event types and lower-case letters (e.g., c, e, s) for event instances. Each event instance has two timestamps, an occurrence and an arrival timestamp. The occurrence timestamp of an event instance e, denoted by e.ts, reflects the time when e was generated while the arrival timestamp e.ats denotes the time when e was received by the event processing system. We use Si: <ei1, ei2, ..., ein> and S: <S1, S2, ..., Sm> to denote an event stream and a mixed event stream, respectively. Si: <ei1, ei2, ..., ein> is constructed by the event instance sequence with the same event type while S: <S1, S2, ..., Sm> is heterogeneous, being populated with event instance sequences of m different event types.

Now, we formally define ordered event stream, out-of-order event, pattern matching plan, latency distance and purging time.
4. Out-of-Order Event Stream Processing

As mentioned in section 2, out-of-order arrivals of events may cause either missing resulting matches or producing incorrect matches, so efficient pre-processing over out-of-order event streams is very important for correct pattern matching.

Suppose that a mixed event stream $S: <S_1, S_2, ..., S_n>$ is generated from heterogeneous networks and then transmitted to the event processing system where out-of-order arrivals of events may exist. Out-of-order arrivals of events together with the limited memory space make the event processing system have to quickly purge those useless events for pattern matching. Based on latency distance and purging time, we present a novel method for processing out-of-order event streams (called LDOP).

Algorithm 1. LDOP ($S, SEQ$)

Input: event stream $S$ and pattern matching plan $SEQ$

Output: pattern matching result

1: $LD = -\infty; PT = -\infty$
2: while (get next epoch) do
3:   for (each event type $E$ appearing in $w_i$) do
4:     if ($E$ does not appear in $SEQ$) then
5:       all instances of $E$ in $w_i$ are purged from memory queue;
6:     end if
7:   end for
8:   $LD_{i} = LD(S, w_{i});$
9:   for (each event $e_{ij}$ appearing in $w_i$) do
10:  if ($e_{ij}$ is missing match event or newly arrived event) then
11:    $PT_{i} = PT(e_{ij});$
12:  else /* $e_{ij}$ is a matched event */
13:    $PT_{i} = Min(PT(e_{ij}), PT);$ /* $e_{ij}$ denotes the matched event of $e_{ij}$ */
14:  end if
15: end for
16: if ($PT$ is larger than the current time) then
17:   execute pattern matching processing;
18: else
19:   purge $e_{ij};$
20: end if
21: return result;
22: end while
Specifically, LDOP first filters out those event instances whose event types do not appear in the pattern matching plan. Then, it calculates the \textit{latency distance}. Furthermore, it calculates the \textit{purging time} for each event instance. Finally, it decides on whether an event instance must be purged according to the \textit{purging time}. We formalize LDOP in Algorithm 1. In Algorithm 1, the operator $\min(PT(e_{ij}^n), PT)$ returns the smaller value of $PT(e_{ij}^n)$ and $PT$.

To reduce the sorting time on the sequence of events or the construction time of stacks [6–8], LDOP utilizes linked list to organize the matched events. The node structure of the linked list is $<\text{pointer}, \{\text{event type [event instance, ts, ats]}, \text{purging time}\}>$

\section{5. Redo Strategy Based on Cloud Platform}

In this section, we present a cloud platform-based \textit{redo strategy}, called CPRS. Like the replay of basketball game, CPRS reprocesses out-of-order event streams to correct the false pattern matches at the real-time phase by means of the powerful capacity of storage and computation.

Figure 1 depicts the cloud platform used by CPRS, which contains a master node (master for short) and some slave nodes (slaves for short). In Figure 1, $\rightarrow$ denotes the newly arrived streams, $\leftrightarrow$ represents communication between master and slaves, and $\rightarrow$ denotes the data passing between slaves.

The master receives event streams from the event processing system. When a stream reach the master, the master is responsible for sending the stream to a slave and meanwhile the log $(\text{SlaveID, Event type, } <\text{MinTimestamp, MaxTimestamp}>)$ is recorded at the master, where $\text{SlaveID}$ denotes the identification of the slave, $\text{MinTimestamp}$ and $\text{MaxTimestamp}$ represent the minimum occurrence time and the maximum occurrence time of arrived events. The stream received by the slave moves forward and meanwhile, the slave will send time range of each event type back to the master. When the memory overhead of the slave exceeds its maximum threshold $\text{MaxMemory}$, the master adds a new slave to process the newly coming stream. Once the time of the event stream staying on a slave is longer than its maximum threshold $\text{MaxTime}$, the slave writes them into disk and purges them from its memory. The $\text{MaxMemory}$ and $\text{MaxTime}$ of each slave are recorded at the master.

Based on the above cloud platform, we formalize CPRS in the following Algorithm 2 and 3. Algorithm 2 is executed at the master and Algorithm 3 is running at the slaves.

\begin{algorithm}[h]
\caption{CPRS-M ($S$, $SEQ$)}
\begin{algorithmic}[1]
\Require event stream $S$ and pattern matching plan $SEQ$
\Ensure pattern matching result
\For {(each time window $w_j$)}
\State select the next slave with the minimum $\text{SlaveID}$ and send the message “ready” to it;
\If {receive the message “yes” from the slave)}
\State send the event stream in $w_j$ to the slave and record $(\text{SlaveID, Event type, } <\text{MinTimestamp, MaxTimestamp}>)$;
\Else
\State select the next slave with the minimum $\text{SlaveID}$ and send “ready” to it;
\EndIf
\EndFor
\end{algorithmic}
\end{algorithm}

\begin{algorithm}[h]
\caption{CPRS-S ($S$, $SEQ$)}
\begin{algorithmic}[1]
\Require event stream $S$ and pattern matching plan $SEQ$
\Ensure pattern matching result
\State receive the message “ready” from the master;
\If {$\text{SlaveID.memory} > \text{MaxMemory}$}
\State send the message “no” to the master;
\Else
\State send the message “yes” to the master and wait to receive the stream from the master;
\EndIf
\State receive the pattern matching result and return it;
\end{algorithmic}
\end{algorithm}

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{cloud_platform.png}
\caption{Illustration of the cloud platform.}
\end{figure}
for (each event $e_{ij}$ received from the master) 
adjust time range;
if ($e_{ij}$ is missing match event)
    $PT := PT(e_{ij})$;
else /* $e_{ij}$ is a matched event */
    $PT := \min(PT(e_{ij}^*), PT)$; /* $e_{ij}^*$ denotes the matched event of $e_{ij}$ */
end if
end for
if ($PT$ is larger than the current time);
execute pattern matching processing;
else
    if ($ST(e_{ij}) > \text{MaxTime}$)
        write $e_{ij}$ and its matched events into the local disk and purge it from the memory;
    end if
end if
send the pattern matching result to the master;

In Algorithm 3, $\text{SlaveID.memory}$ denotes the memory overhead of the slave and $ST(e_{ij})$ represents the time that $e_{ij}$ stays on the slave.

6. Experimental Evaluation

Experiments are run on several PCs with Pentium (R) dual CPU 1.86 GHz and 2 GB memory, and all algorithms are implemented with Java. For evaluating LDOP, one PC, acting as the event generator, is responsible for generating event streams and sending them to the event processing system. We can also set the percentage of out-of-order events for each event type by configuring the generator. In our experiments, there are 10 kinds of event streams with different event types.

We mainly compare the proposed methods with the existing Aggressive, Conservative and K-Slack. The main performance metrics are processing time, accuracy rate (AR) and average application latency (AAL). Accuracy rate is defined by the following equation:

$$AR = \frac{|R_i \cap R_o|}{|R_i|}$$

(3)

where $R_i$ denotes the result set of pattern matches on the ordered event stream, $R_o$ represents the result set of pattern matches on the out-of-order event stream, and $|R_i|$ and $|R_i \cap R_o|$ denote the numbers of elements in $R_i$ and $R_i \cap R_o$ respectively.

Average application latency is defined by the following equation:

$$AAL = \frac{\sum(T_{out} - \max(e_{ij}.ats))}{n}$$

(4)

where $T_{out}$ denotes the output time of pattern matching events, $\max(e_{ij}.ats)$ represents the maximum arrival time of the event instances composed into the pattern matching result, and $n$ is the number of matching patterns.

6.1 Evaluating LDOP

We first test the performance of LDOP and the other three methods in terms of AAL, as a function of out-of-order percentage of events when the value of out-of-order percentage of events varies. Out-of-order percentage of positive/negative events is the ratio of the number of positive/negative events received over the number of total events received. In our experiment, the time window size is 10 time units, and the parameter $K$ in K-slack is 10. We also change the length of pattern matching plan from 3 to 7.

In Figure 2(a), we can see that as expected the increase in the out-of-order percentage of positive events does not greatly impact AAL of different methods. Because Aggressive produces results quickly, and $K$ of K-slack is constant, their AALs are nearly horizontal lines. But Conservative produces output only when its correctness can be guaranteed, so it consumes more time. LDOP purges those useless events for pattern matching, so its AAL is not very large. In Figure 2(b), we observe that except K-slack AALs of the others increase as the out-of-order percentage of negative event instances increase. This is because more out-of-order negative events will incur more re-computation.

Further, we evaluate the performance of LDOP and the other three methods in terms of processing time, as a function of out-of-order percentage of events. In Figure 3, we can see that as expected the increase in the out-of-order percentage of positive/negative events does greatly
impact the processing time of different methods except \textit{Aggressive}. The reason is that more out-of-order positive/negative events will incur more re-computation. In contrast, \textit{Aggressive} outputs results quickly, so its processing time is very small. Because \textit{Conservative} must guarantee the correctness of output, so it consumes more time. In the above test, the \textit{processing time} of LDOP is smaller than that of \textit{K-slack} and \textit{Conservative}. This is because LDOP utilizes the linked list to record the matched events, so the sorting time on the events is smaller.

Finally, we evaluate the performance of LDOP and the other three methods in terms of \textit{accuracy rate (AR)}, as a function of \textit{out-of-order percentage of events}. As shown in Figure 4, the increase in the out-of-order percentage of positive/negative events causes the decrease of \textit{AR} of four methods. The reason is that more out-of-order positive/negative events will incur more false matched events. In Figure 4, we observe that \textit{Conservative} and LDOP perform better than the other two methods. As we know, \textit{Conservative} method consumes more time to guarantee the correctness of output, and the latency of LDOP is larger than the other two methods, especially when out-of-order percentage is larger, so it obviously shows better behaves. In Figure 4(b), the \textit{AR} of all methods is much smaller than the results in Figure 4(a). This is because negative event types have more impact than the positive event types.

\subsection*{6.2 Evaluating CPRS}

In this experiment, we evaluate the \textit{throughput} and \textit{AAL} of CPRS using real data [13,14]. The experiments are conducted on a cluster which has 8 nodes. The nodes are connected by a gigabit Ethernet. Each node has 1.86 GHz, 8 G memory and 120 G disk. The algorithms are implemented using Java and compiled with Eclipse 3.6 on Ubuntu 11.10.

Figure 5(a) shows the throughput of tuples per second of CPRS. In the initial phase, output is much smaller. This is because that the processing ability is much weaker. In the middle phase, with the increasing number of concurrent slaves, throughput increases dramatically. In the final phase, due to the larger number of concurrent slaves and more frequent communication among slaves and the master, the increasing of throughput becomes gentle. In the Figure 5(b), we evaluate the \textit{AAL} of CPRS. When the out-of-order percentage of
event streams is much smaller, the AAL also becomes small. However, with the increasing percentage of out-of-order events, the AAL grows dramatically. This is because that the processors have to wait longer time and spend more time on sorting event sequences. Meanwhile, the master and slaves will spend more time on communication.

6.3 Experiment Summary

The K-slack, aggressive and conservative have different scopes of applicability. The above three methods prefer to use in an environment of constant latency, lower out-of-order percentage, and higher out-of-order percentage, respectively. Obviously, they cannot adapt to dynamic and heterogeneous environments, especially in a heterogeneous network. In contrast, the proposed methods (LDOP and CPRS) can adapt the heterogeneous network and have relatively smaller AAL and processing time.

7. Conclusions

In this paper, we address the problem of out-of-order processing over event streams. We present an efficient method for processing out-of-order event streams in real-time by introducing latency distance and purging time. For the false pattern matching events, we propose a cloud platform-based redo strategy to correct the false pattern matches at the real-time phase. We conduct extensive experiments on synthetic data and real RFID data, and the experimental results demonstrate the feasibility and effectiveness of our methods. Our work is a beneficial complementary to the existing research work on out-of-order event stream processing.

Acknowledgements

This work is supported by the Natural Science Foundation of China under Grant No. 61170174 and the Natural Science Foundation of Tianjin under Grant No. 11JCYBJC26700.

References


Manuscript Received: Jan. 24, 2013
Accepted: Oct. 7, 2013