Efficient Pattern Detection Over a Distributed Framework

Ahmed Khan Leghari(✉), Martin Wolf, and Yongluan Zhou

Institute of Mathematics and Computer Science (IMADA),
University of Southern Denmark, Odense, Denmark
{ahmedkhan,zhou}@imada.sdu.dk, mawo09@student.sdu.dk

Abstract. In recent past, work has been done to parallelize pattern detection queries over event stream, by partitioning the event stream on certain keys or attributes. In such partitioning schemes the degree of parallelization totally relies on the available partition keys. A limited number of partitioning keys, or unavailability of such partitioning attributes noticeably affect the distribution of data among multiple nodes, and is a reason of potential data skew and improper resource utilization. Moreover, majority of the past implementations of complex event detection are based on a single machine, hence, they are immune to potential data skew that could be seen in a real distributed environment. In this study, we propose an event stream partitioning scheme that without considering any key attributes partitions the stream over time-windows. This scheme efficiently distributes the event stream partitions across network, and detects pattern sequences in distributed fashion. Our scheme also provides an effective means to minimize potential data skew and handles a substantial number of pattern queries across network.

1 Introduction

Complex event detection is a growing field and a lot of work has been done to efficiently detect sequence patterns [1–12]. Much of the past implementations and research are aimed to process complex events on a single machine. Some work has also been done to detect complex events in distributed environment [1,13–15]. The focus of that work is partitioning the event stream on certain keys or attributes, such as stock symbols. If no such partitioning attribute exists in the query, or very few partitioning attributes/keys exist, then the event stream can not be efficiently parallelized, and data being sent to various processing nodes becomes very skewed, causing some machines to process much more data than others, and ultimately leading to unfair distribution of processing load and overall performance degradation.

Consider the case of two arch rival football teams playing in the world cup final, where each team is supported by thousands of emotional supporters in the stadium. To manage any health related emergency each person watching the game in stadium is made equipped with a sensor that is continuously sending the vital details about each person's physical conditions as events, and each

M. Castellanos et al. (Eds.): BIRTE 2013 and 2014, LNBIP 206, pp. 133–149, 2015.
DOI: 10.1007/978-3-662-46839-5_9

event consists of (heart_rate, breathing_rate, body_temperature). As there is no natural partitioning key or demarcation of events carrying information about heart rate, breathing rate, and body temperature, hence, the best option is to partition them on the basis of time windows. Moreover, if we assume that the numbers of events generated by sensors are associated with every individual's heart rate, then a sudden and dramatic change in the game would greatly affect the number of events generated throughout the game. To handle such situations where the number of generated events change dramatically, and possibly outperform a single machine, a better option is to process them in a distributed fashion.

Consider another case of event stream partitioning, if there are limited number of partitioning keys in a query, then the resultant stream partitions can be too large to be processed on a single machine, and it would again require further repartitioning. Our proposed scheme does not experience such problem as the volume of data being processed by a machine, and the degree of the parallelization is subject to the available machines.

Our proposed approach is based on partitioning the incoming data stream on time-windows, it efficiently distributes fast data stream among multiple machines, and detects sequence patterns [3] running queries on machines. The essence of this approach is that, it can be used to parallelize processing of any sort of event stream without need of any partitioning key or attribute in the pattern query. Simultaneously, while distributing the stream partitions, it ensures a fair and uniform data distribution among multiple machines, avoiding any possible data skew.

The work presented in this paper is summarized as follows.
(i) An event stream partitioning strategy based on time-windows, that does not take into account event stream attributes. *(ii)* A pattern sequence detection strategy that distributes stream partitions across number of machines, and executes pattern queries over stream partitions. *(iii)* Optimizations are proposed for efficient distribution of overlapping stream partitions across number of machines by removing duplicate events from consecutive partitions. *(iv)* A technique is discussed to prevent data skew while distributing stream partitions to multiple machines. *(v)* A simple cost model is proposed to evaluate the execution cost of pattern queries, based on the structure and complexity of the pattern queries.

The remainder of the text is organized as follows. Section 2 discusses problem description, and provides problem statement, system and cost model. Section 3 provides motivations of the research, challenges and an overview of the existing research work. Section 4 provides some background details. Section 5 briefly explains the algorithms, proposed strategy and optimizations to the approach. Section 6 presents the implementation details, evaluation and outcomes of the experiments. Section 7 concludes the paper and discusses future research avenues.

2 Problem Description

2.1 Definitions and Semantics

Event. A data stream consists of various events, each event can be identified through it's event_id with respect to it's arrival time t and associated values termed as *attr_values* [16,17]. *Time-window.* A time-window can be described as a time-interval/passage of time, between time t_1 and time t_2 such that $t_2 \geq t_1$ [18]. The length of time-window is represented in units of time, as described in WITHIN clause of query-1. *Sequence pattern.* A sequence pattern is a sequence of events occurring in a sequential order. Such as (A; B; Q), a sequence query detects all instances of event B that follows event A, and all instances of event Q that follows B. *Sequence query.* A sequence query Q_s detects all the occurrences of relevant events in a time-window as depicted in Query-1.

Query 1. Pattern Sequence
PATTERN SEQUENCE $(e_3; e_4; e_5)$
WITHIN 5 min

Relevant event. A relevant event e is an event of interest iff e belongs to a sequence pattern S_p. *Stream partition.* A stream partition consists of a finite number of events that arrive in the system between t_1 and t_2. *Query semantics.* A sequence query evaluates all the events one at a time in their arrival order in a partition. Matching of events is non-greedy and contiguous in a stream partition [19], and every event is matched against a sequence pattern from t_1 to t_2 where $t_2 \geq t_1$ in a stream partition as shown in Fig. 1. *Overlapping partitions.* While partitioning the event stream on time-windows it is possible that a stream partition contained sequence of events which are also part of another pattern sequence in successive stream partition as depicted in Fig. 2. In such a case where consecutive stream partitions contain or share some of the events in the identical sequence are termed as overlapping partitions.

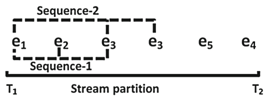

Fig. 1. Query semantics

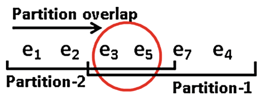

Fig. 2. Overlapping partitions

2.2 Problem Statement and Formulation

The problem is formulated as: *From an incoming event stream S, continuously create stream partitions based on n units of time, send stream partitions to multiple machines over network, and while running sequence queries over stream partitions, detect sequence patterns and simultaneously load balance the entire system.*

While detecting event sequences in distributed fashion, the stream partitions and pattern queries should be distributed among machines in such a way, that none of the machines should be overloaded, the objective function to achieve this goal is described as follows:

$$w_i \leq \frac{\sum_{i=1}^{m} w_i \alpha}{m}, \; where \; \alpha \geq 1 \; and \; w_i \alpha \leq T \tag{1}$$

where w_i specifies average workload on a machine, α denotes a slight variation of workload that can be caused by one or many new stream partitions received by a machine, m is the number of available machines and T is a certain threshold based on machine's optimum level of processing that represents maximum load to be allowed on any machine, at any time t. In this equation $\alpha \geq 1$ denotes that average workload on a machine can vary by the processing load of one or many stream partitions. This variation in average workload is caused by the arrival of a single or multiple overlapping partitions.

The objective function is to share the processing load equally among all machines. If on one or more machines $w_i > T$ at some point in time, then the load would be readjusted among available machines. But, in case if average workload on all machines is $w_i > T$ then to avoid any performance degradation a new machine should be added in the system, or T should be adjusted accordingly. The second objective function deals with minimizing communication cost and can be described as follows: Assume that sending an event e_i to a machine on network would cost C_i then the total number of events sent over network would cost:

$$\sum_{i=1}^{n} C_i \tag{2}$$

While sending stream partitions to machines, events or sequence of events which are part of overlapping stream partitions can be just sent once over the network, without causing common events to be sent in duplicates. This requires careful planning while sending stream partitions to multiple machines. Sending duplicate events can be avoided by considering the similarity of the sequence queries running on a machine. Sending common events just once in the stream partitions can reduce the overall communication cost as described below.

$$C_T = \sum_{i=1}^{n} C_i - \sum_{j=1}^{m} D_j \tag{3}$$

Here, C_T represents the total cost of sending events without sending duplicate events, C_i represents the total cost of sending events including the cost of sending duplicates, D_j represents the cost of sending duplicate events. While maintaining $w_i\alpha \leq T$ on a machine and removing duplicates from the consecutive event stream partitions, we have certain constraints described as follows. Removing duplicate events from the event stream partitions requires temporary buffering of the events in main memory.

While buffering events we have to be careful about: **(i)** hardware limits imposed by the unbounded nature of event stream, and **(ii)** timely response needed by some time critical application. So, the size of the buffer should be carefully chosen to remove maximum duplicates from the stream partitions, without affecting the response required by the time-critical applications. A careful buffering of events would allow us to send events into batches, causing efficient use of network bandwidth, saving CPU cycles, as single interrupt required by a batch vs. multiple interrupts required by events sent individually. But, there are two issues associated with creating batches of events: **(i)** Batching of events would increase the communication latency. **(ii)** After removing duplicates, some of the machines might receive less events to process then some others. If this would continue for sometime, it would cause data skewness, poor workload distribution, and hence overall less efficient resource utilization. Therefore, to keep communication latency at an acceptable level for time critical applications, the following equation is devised.

$$L_B \leq T_L \tag{4}$$

Here, L_B represents increased latency incurred while creating a batch, and T_L is the latency threshold. It means, that while creating a batch of events, the latency must not exceed a certain threshold. The latency threshold is a tunable parameter, tuned as per requirements of the time-critical applications. If latency threshold satisfies the requirement of time critical applications, then it would also satisfy the applications with flexible time constraints.

2.3 System Model

In our model, a machine receives an incoming, unbounded sequence of events, referred to as an event stream. It is assumed that events in stream are externally timestamped, and arriving in strict order. In our model a *partition* of an event stream consists of a finite number of events that arrive in the system between the start and end of a time-window denoted as t_{start} and t_{end}. The length of a time-window is specified in the WITHIN clause of a pattern query as mentioned in Query 1.

Each pattern query before being executed is assigned a weight (see Sect. 2.4). The terms weight, load and cost are used interchangeably in the same context in our text. We assume that all the machines in the network have identical hardware resources. A single machine termed as *Stream Partitioner* in the system handles incoming event stream, partitions it and sends these partitions to other machines termed as *worker* machines over network. Each worker after receiving a stream partition executes a pattern query over the partition, and using NFA (Nondeterministic Finite Automata) evaluates the pattern sequence.

Role of Workers. In our model, at the system startup time, every worker waits in FIFO order for its turn, and receives a stream partition that turns it *busy*. Stream partitions to be sent across workers are scheduled initially on the FCFS basis and later as per current load of the worker. The current load of the workers is maintained in a weight-lookup-table that keeps record of all the workers participating in the pattern detection process. The load on the workers is determined through the cost (weight) of pattern queries being run and the number of events being processed by the worker. If all workers have identical load then a worker will randomly be selected to receive a new stream partition.

2.4 Cost Model

While maintaining the execution cost (associated wight) of pattern queries in the weight-lookup-table, the following parameters would be considered.

Length of the Time-Window. The size of a time-window determines the size of a stream partition to be sent to a worker represented in units of time. A larger partition usually means there would be more events to process, and more time is required to evaluate a pattern sequence. *Events' arrival rate.* The number of events arrive in a time-window depends on the pace of data stream, a faster data rate means there would be more events to process in a time-window, but in our system we assume that events are arriving in order and in fixed time interval.

Number of Relevant Events Arrived in a Time-Window. The number of events relevant to a sequence query may vary from one time-window to another. Multiple relevant events in a time-window require us to evaluate all the combination of relevant events, increasing the cost of pattern detection. *Number of predicates associated with the events.* A pattern query may have some predicates, that specify rules termed as attributes of the events. Events arrived in a time-window will be evaluated against one or more query predicates. Evaluation of predicates against number of events also affect the query execution time. As all of the above factors play a combined role in the execution time taken by a pattern query, hence, the cost metric can be described as follows.

2.5 Formal Definition of Cost Metric

Given a Query Q, detecting a pattern sequence P, comprised of R events, where P=($e_1, e_2, e_3 ... e_n$), M predicates, within window-size of t time units, the cost of query would be:

$$\left[\sum_{i=1}^{n}\sum_{j=1}^{m} R_i M_j\right] + \left[C\prod_{i=1}^{n} R_i\right] \tag{5}$$

Here, R_i in the left side of the addition denotes the number of relevant events arrived in a time-window, and M_j denotes the number of predicates evaluated against each relevant event. While evaluating a sequence using NFA, from all possible combinations of relevant events denoted as R_i, a fraction of pertinent combinations denoted as C would be considered.

3 Motivation and Challenges

3.1 Motivation

Centralized implementations of CEP systems like Cayuga and SASE+ can handle pattern detection queries running on a single machine [1,13,15]. Single machine CEP implementations are aimed to handle all the queries and event stream in a centralized machine. But, a distributed stream processing approach is more suitable due the reasons as follows.

(i) Processing load can be distributed among multiple machines without putting unnecessary load on a single machine. (ii) Local optimizations can be done on the data available locally. (iii) Unwanted/irrelevant events as well as noise can be filtered out on the server side while partitioning the event stream, and each machine on the network just has to process individual segment of data stream. (iv) High degree of parallelization that can not be achieved on a single machine.

Time-window based partitioning can also be reformed to enforce better security control where each machine over the network would just receive the partition that is relevant to that specific machine.

3.2 Challenges

There are various challenges to efficiently partition and distribute event stream among number of machines. *First*, how to detect patterns involving sequence operators, by partitioning the event stream without considering key attributes, and running pattern queries in parallel on different machines. *Second*, what should be the criteria to partition the event stream. *Third*, how to select a machine that would receive a stream partition. *Fourth*, if there are some inter or intra partition overlapping of pattern sequences then how to handle such situations. *Fifth*, how to load-balance this entire partitioning process to avoid a situation in which a machine receives multiple stream partitions continuously that might overwhelm its processing capacity. The subsequent sections will briefly explain the techniques and strategies to deal with the aforementioned issues.

4 Related Work

Issues related with event stream processing have been studied by many [2–7,14, 15,19–33]. However, the work that is most relevant to ours is done by Balkesen et al. and Hirzel. Balkesen et al. in [14] proposes a run-based intra query parallelism scheme called RIP for scalable pattern matching. The focus of his work is to exploit multi-core architecture of a CPU. In his work an instance of Finite State Machine (FSM) is termed as a *run*. RIP distributes input events that belong to individual run instances of a pattern's FSM to different processing units or cores and each processing unit in a multi-core architecture performs pattern matching on a given sequence of inputs. As this approach is based on multi-core architecture of a single machine so it has some upper bound on the number of queries processed in per unit time. Balkesen et al. termed their RIP based approach skew-tolerant, but as their implementation is based on an isolated pattern matching engine, exploiting the features of a multi-core CPU, hence, their approach can not be compared with the data skew that could be seen across multiple machines processing real distributed pattern matching.

Hirzel proposes in [19], a partitioning scheme and a SPL operator that partitions and parallelizes the event stream [22]. It is based on partitioning the event stream on some stream attributes, referred to as *partitioning key*. The partitioning of event stream takes place using *partitionBy* clause that is identical to SQL's *Group By* clause. The partitionBy clause takes a key (or multiple keys) as its parameter to partition the event stream, and the degree of parallelization is dependent upon the number of distinct keys. If a parameter (partitioning key) is not passed to the partitionBy, or if partitionBy clause is not present in the scheme then the event stream would not be partitioned and sent to a single machine for processing. Hirzel's scheme requires some attributes to partition and parallelize the event stream.

Our work focuses on a different perspective i.e. time-windowing of event stream. The approach we propose is suitable to partition and parallelize any kind of event stream without taking key attributes and using any partitionBy or GroupBy clause.

5 Proposed Method

5.1 Central Theme

Our proposed event stream partitioning method is based on the principle of divide and conquer. A machine after receiving an event stream, partitions it on time-windows, and sends these partitions to individual workers over a network. Each worker after receiving a stream partition, executes a pattern query (or number of queries) over that partition to detect pattern sequences. Assume that Query 1 is registered as a pattern query to detect a pattern sequence over an incoming event stream S, and the pattern sequence specified in the query is comprised of events as shown in Fig. 3 by dotted rectangular. Algorithm 1 illustrates that upon receiving the event stream, the system will first detect event e_3, that is the first event in the required pattern sequence. Detection of e_3 at time t would be marked as start of the time-window or t_{start} and event e_3 and all subsequent relevant events from that point in time would be sent to a worker that would further evaluate the pattern sequence. Figure 3 shows that multiple identical events can arrive at any time, so each e_3 would be considered as a start of a new time-window or t_{start} (depicted by dotted line arrows), and would be sent to a worker that would further detect the pattern sequence. All the events that arrive after the detection of the first event until the end of five minutes time-window would be considered as a partition. After the arrival of the last event in that window that point in time is marked as t_{end} denoting end of time-window, and could be start of a new time-window. As shown in Fig. 3 by solid line arrows that detection of the first event e_3 at t_{start}, and end of a time-window at t_{end} is a logical partition on time.

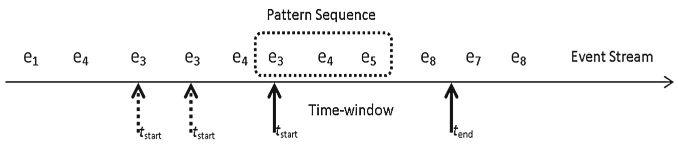

Fig. 3. Detection of an startup event

Algorithm 1. CreatePartition /*Partitioning the event stream */

```
1: Input: $S$, denotes an event stream, $e_i$ denotes the starting event of a pattern
   sequence, $t_{window}$, is size of the partition
2: Output: $P$, denotes a partition
3: while $e_i$ is detected in $S$ do
4:     Let $t_{start}$:= $e_i.ts$
5:     Let $t_{end}$:= $t_{start}$+$t_{window}$
6:     Start partition $P$
7:     Let m:=Getmachine()
8:     Send event $e_i$ in partition $P$ to machine $m$
9:     if $t \geq t_{end}$ then
10:        End partition
11:        break while
12:    end if
13: end while
```

Algorithm 1 deals with creation and termination of a single partition, multiple partitions can be managed by calling the same operation multiple times. Each partition if completes without any overlapped t_{start} would be of identical size (in terms of time) while query 1 is being run, but the number of tuples (or events) in each partition might vary due to some external factors which could affect the occurrence of events. Machines across the network will initially receive partitions in round robin fashion and then as per their respective processing load. For efficient use of network bandwidth, unwanted or irrelevant events which would not be part of any partition, or not be the start of any sequence, would be simply skipped by the machine partitioning the event stream.

5.2 Processing Multiple Distinct Pattern Queries

A pattern detection system detects patterns with varying complexity, and each distinct pattern has a different evaluation cost, as some patterns can be more expensive to evaluate than others. Hence, to handle variety of queries it becomes necessary to carefully distribute the processing workload across multiple machines. Figure 4 depicts multiple varying length pattern sequences, to detect pattern sequence 1, upon detecting the first event i.e. IBM, this point is marked as the start of the window or t_{start}, and the end of the time-window would be marked as the t_{end}. After the start of time-window all the events including the first event would be sent to a worker for further processing. The *start*

Algorithm 2. GetMachine Algorithm

```
1: Input: LM, a priority queue of all machines involved in pattern detection
2: Output: m, a worker machine with minimum load
3: LM:=Prioritize(LM)
4: m:=LM.getNext()
```

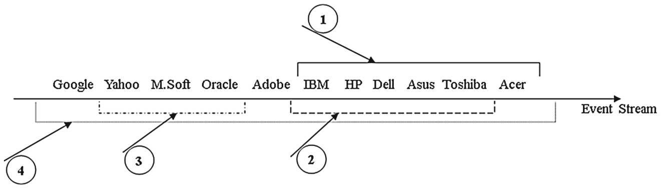
1
Google Yahoo M.Soft Oracle Adobe IBM HP Dell Asus Toshiba Acer
Event Stream
4
3
2

Fig. 4. Processing multiple distinct pattern queries

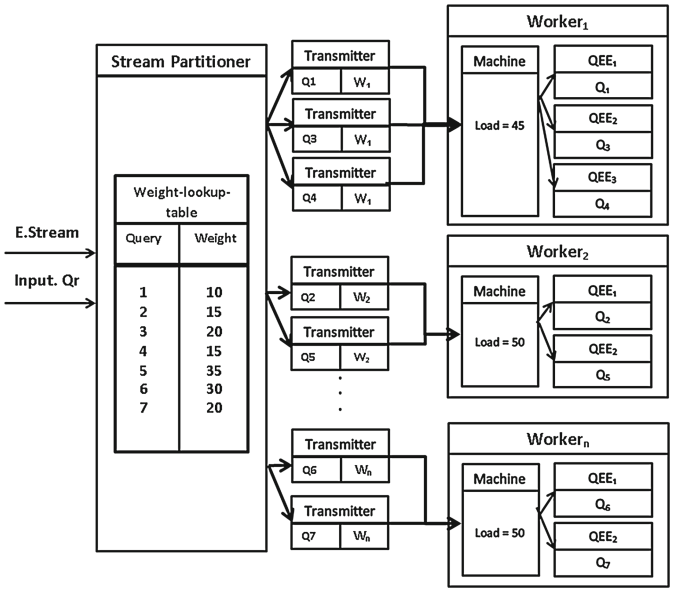

Fig. 5. Distribution of load among multiple machines

time and *end time* of the window sets the boundaries for the stream partition. Refer to Fig. 5, the stream partitioner after detecting the first relevant event of a sequence, partitions the event stream, and the transmitter sends the event stream partition to a worker. The worker after receiving a stream partition executes a pattern query over partition using a Query Execution Engine (QEE). The stream partitioner keeps a weight-lookup-table and ensures before sending a stream partition that the recipient worker should have less load then the other workers. The weight-lookup-table maintains processing load of all the workers in a priority queue, Algorithm 2 takes this priority queue and returns a worker with highest priority (with minimum load) to receive a partition.

5.3 Processing Multiple Overlapping Partitions

During processing of a huge number of complex patterns, it is possible that while partitioning the event stream some of the stream partitions might overlap each other as shown in Fig. 2, in such scenario, a careful decision can increase throughput, and cause efficient use of network bandwidth. While sending overlapping partitions to machines, there are three possible choices: *(i)* To send entire partitions (including common events in each) to different machines, this would lead to transfer duplicate events and cause waste of network bandwidth, specially when large partitions having multiple events in common sent over the network. *(i)* To send entire partitions (including common events in each) to a single machine. As, both partitions (including individual sets of common events) are sent, hence it is also a waste of network bandwidth. *(iii)* To send partitions sharing events with one another to a single machine, without sending common events twice. The third approach is simple, and effective than the first and second to avoid duplicate events to be sent across multiple machines, as it would send common events just once, and save network bandwidth. To efficiently select the most suitable worker machine to process overlapping partitions, we would maintain an in memory list of workers, with respective partition overlaps (if there is any) as depicted in Table 1. Before sending an overlapped partition to any worker, the partitioner checks the list for number of events in overlap, and number of most recent overlapping partitions sent to a worker. The new overlapping partition would be sent to the worker with highest number of overlapping events, Iff there is no other worker who received a non-overlapping partition in its most recent turn.

In Table 1 it is shown that worker 2 is the one with highest partition overlap, but as there is worker 3 whose most recent partition was non-overlapping or zero, hence, the partition would be sent to worker 3 as it must have finished or could be in the middle of processing it's most recent partition. The above solution to send overlapping partition is suitable for most of the cases. To handle a special case where a stream partition has no or zero overlap with respect to the partitions sent to all workers, and all workers have an equal partition count, such situation would be handled using Algorithm 3. This algorithm while sending partitions across workers considers an upper bound of processing capabilities of each worker termed as threshold. The threshold is based on the hardware configurations of the machines. A worker with minimum load is selected through calculating the current load, plus load of the query. Let m_1 denote the most recent machine that has received a partition, and m_2 denotes a machine with minimum processing load; let T be the maximum threshold for machine load, p be the overlapping partition, and Q be a pattern query to detect a pattern over p, then using this algorithm if the current processing load on m_1 plus the new processing load to be assigned to it is less than T then it will receive the next stream partition otherwise a suitable machine will be sought.

Optimization to the approach. As many of the real world applications require a rapid detection of event patterns such as seismographic patterns, there are some applications which do not require a real-time response, and their processing can be delayed up to a certain time interval. Sending individual events to a worker is not an efficient method that leads to poor CPU and bandwidth utilization. Algorithm 4 groups individual events into batches and sends them as a single

Table 1. Worker selection for overlapping partitions

Worker_id	Partition overlap (#of overlap events)	Partition count (# of partitions already received)
1	5	2
2	6	3
3	2	0

batch for further processing. In Algorithm 4, we assume that each pattern query has a priority associated with it, this priority can be assigned to a pattern query as per specific needs of an application. We have set a user defined priority threshold for all the queries in the system. When a pattern query has a priority equal to or higher than the priority threshold it requires that relevant events should be sent without any delay, so, after detection of the very first event such as e_j, it would be checked that is there any existing batch ready to be sent to the worker m, if such a batch exists then e_j would be included in the batch and the batch would be sent without any delay. If there is no existing batch, then e_j (as a start of the new partition) would be sent without any delay to the concerned worker. But, when a query has priority less than the priority threshold, it means that the nature of the application associated with the query can ignore a certain delay and we will start creating a batch of events.

Algorithm 3. Machine selection for sending overlapping partition

```
1: if $p$ is detected then
2:    if $m_1$.totalLoad+Q.load<T then
3:       send $p$ to $m_1$;
4:    else
5:       send $p$ to $m_2$
6:    end if
7: end if
```

The size of the batches would be based on a user defined threshold set as per time critical needs of the applications. Creating batches of events can be very useful when there is an overlapping between partitions, as certain events in overlapping partitions can be just sent once to a worker, leading to a reduced network traffic. The efficiency of sending events into batches is highly dependent on the batch size, which is further dependent on the priority associated with the query.

6 Experimental Evaluation

Our present work is an evaluation of our proposed method, in future the focus of our work would be to extend our experiments to process real life large data

Algorithm 4. BatchOfEvents /* Creating batch of events */

```
1: Input: $Q_i$ is the $i^{th}$ query, $e_j$ is the first event in the relevant partition, $p_k$ is the
   priority associated with the query, $B_T$ is batch-size threshold
2: do
3: if $p_k \geq p_t$ */ $p_t$ is some priority threshold */ then
4:    if Any BoE alreday exists then
5:       Attach $e_j$ to BoE
6:    else
7:       CreatePartition() /* Call the partition procedure */
8:    end if
9: else
10:    if $p_k < p_t$ */ $p_t$ is some priority threshold */ then
11:       Start batching events (BoE)
12:       BoE.end :=BoE.currentSize+$t_{window}$
13:       while $BoE.end \leq B_T$ do
14:          Add $e_j$ to BoE /* Add every event to the batch of events */
15:       end while
16:       Let m:=GetMachine() /*Call GetMachine() Procedure */
17:       Send BoE to $m$ /* Send batch of events to machine $m$ */
18:    end if
19: end if
20: while(true)
```

sets increasing queries and machines. In our present study a system based on the strategies and algorithms mentioned in Sect. 5 has been developed, and experiments have been conducted on a cluster of four machines, each running Linux on a dual core 2.6 GHz CPU and 4.8 GB of main memory, 4.6 GB of secondary storage. Our setup was based on a single stream partitioner, and three worker machines, all connected through a local area network. We expect that result would differ on an overlay network due to a higher latency and communication cost.

The experiments have been performed running sequence queries consisting of random sequences of English alphabet with varying size of time-windows, and without predicates. Processing sequence patterns involving predicates would be extended in the future work. Each query detects a pattern similar to the Query 1 discussed in Sect. 2.3. Initially, a synthetic stream of five thousand events was generated, and to perform multiple experiments, the same stream was used to construct ten thousand, fifteen thousand, twenty thousand and twenty five thousand events. In each experiment pattern queries were divided in two categories i.e. high priority queries, and low priority queries. High priority queries were assigned priority than the other half of the queries, to mimic real world scenario where some applications need a time critical response.

6.1 Results and Discussion

While conducting experiments, it was observed that the major factor that affects the hardware resources and processing time is not just the event stream itself,

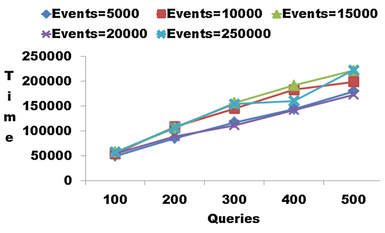

Fig. 6. Processing time of various stream inputs

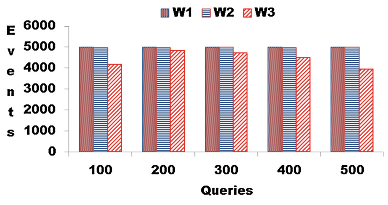

Fig. 7. Distribtion of processing load

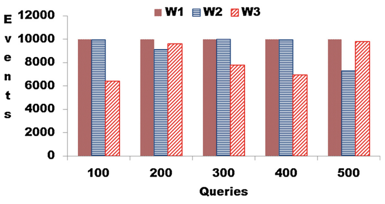

Fig. 8. Distribtion of processing load

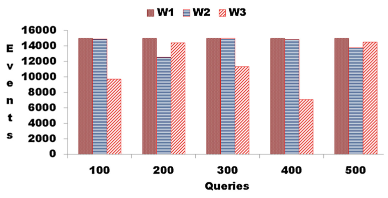

Fig. 9. Distribtion of processing load

but, the complexity of patterns and queries being processed by workers. The complexity and structure of the pattern and the number of predicate decide the evaluation time required by a sequence query. In the first experiment we tested our approach on different sets of input streams, and running one hundred to five hundred queries on each input set. Figure 6 depicts that during all experiments the processing time(ms) shows a linear growth behavior. The reason for this linear growth is that the event streams were distributed fairly well among workers, and there was no worker machine that received huge processing load that would have caused it to spend higher amount of time increasing the overall processing time of the job. To observe the load balancing, possible data skew, and duplicate removals (in overlapping partitions) we conducted many experiments. Figures 7, 8, 9, 10 and 11 depict distribution of processing load among workers, it can be observed that processing load (number of events processed) are distributed fairly well in workers. But, some of the workers have slightly lower processing load, this is due to the reason that overlapping partitions are sent to the workers with maximum partition overlap as described in Sect. 5.3. Sending duplicate events can be avoided while creating partitions by increasing the size of the batches and sending overlapping partitions continuously to some specific worker(s). But, creating larger size batches would effect the time critical nature of some of the applications as well as potentially result in data skew, and load balancing issues. Removing duplicate events to cut down the communication cost is useful in cases where conservation of network bandwidth is of primary concern, but as we have just single stream partitioner with fixed hardware resources, creating larger size of batches can cause a performance bottleneck at the partitioner side. So, the present load distribution among workers

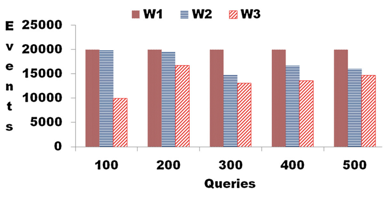

Fig. 10. Distribtion of processing load

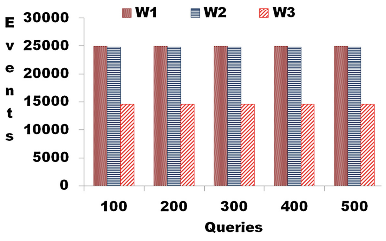

Fig. 11. Distribtion of processing load

shows a mix of load balancing and duplicate removal. It must be noted that in the figures the number of events processed by each approach are higher than the actual number of events given as input, this is because an event can be associated with multiple sequence patterns, and hence, processed individually for each sequence pattern by respective pattern query.

7 Conclusion and Future Work

Partitioning and distribution of event stream on partitioning keys are potentially prone to data skew, as the parallelization relies on the number of keys. In this paper we propose a stream partitioning scheme that is not dependent on the partitioning attributes, it efficiently partitions and distributes the event stream, and elegantly load balances the entire process of detection of sequence pattern across multiple machines. In future we intend to extend our work to include predicate handling, improved load balancing and want to repeat our experiments on large real life data sets. We also want to introduce multiple stream partitioner to avoid performance bottleneck and any single point of failure.

References

1. Diao, Y., Immerman, N., Gyllstrom, D.: Sase+: An Agile Language for Kleene Closure Over Event Streams. ACM Press, New York (2007)
2. Agrawal, J., Diao, Y., Gyllstrom, D., Immerman, N.: Efficient pattern matching over event streams. In: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, pp. 147–160. ACM (2008)
3. Mei, Y., Madden, S.: Zstream: a cost-based query processor for adaptively detecting composite events. In: Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data, pp. 193–206. ACM (2009)
4. Wu, E., Diao, Y., Rizvi, S.: High-performance complex event processing over streams. In: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data, pp. 407–418. ACM (2006)
5. Liu, M., Ray, M., Rundensteiner, E.A., Dougherty, D.J., Gupta, C., Wang, S., Ari, I., Mehta, A.: Processing nested complex sequence pattern queries over event streams. In: Proceedings of the Seventh International Workshop on Data Management for Sensor Networks, pp. 14–19. ACM (2010)

6. Ramakrishnan, R., Cheng, M., Livny, M., Seshadri, P.: What's next? sequence queries. In: Proceedings of International Conferene Management of Data. Citeseer (1994)
7. Liu, M., Li, M., Golovnya, D., Rundensteiner, E.A., Claypool, K.: Sequence pattern query processing over out-of-order event streams. In: IEEE 25th International Conference on Data Engineering, ICDE 2009, pp. 784–795. IEEE (2009)
8. Law, Y.N., Wang, H., Zaniolo, C.: Query languages and data models for database sequences and data streams. In: Proceedings of the Thirtieth International Conference on Very Large Data Bases, vol. 30, pp. 492–503. VLDB Endowment (2004)
9. Seshadri, P., Livny, M., Ramakrishnan, R.: Sequence query processing. ACM SIGMOD Rec. **23**, 430–441 (1994). ACM
10. Zuo, X., Zhou, Y., Zhao, C.-H.: Elastic non-contiguous sequence pattern detection for data stream monitoring. In: Yin, H., Tino, P., Corchado, E., Byrne, W., Yao, X. (eds.) IDEAL 2007. LNCS, vol. 4881, pp. 599–608. Springer, Heidelberg (2007)
11. Gao, C., Wei, J., Xu, C., Cheung, S.: Sequential event pattern based context-aware adaptation. In: Proceedings of the Second Asia-Pacific Symposium on Internetware, p. 3. ACM (2010)
12. Gyllstrom, D., Agrawal, J., Diao, Y., Immerman, N.: On supporting kleene closure over event streams. In: ICDE, vol. 8, pp. 1391–1393 (2008)
13. Demers, A.J., Gehrke, J., Panda, B., Riedewald, M., Sharma, V., White, W.M., et al.: Cayuga: a general purpose event monitoring system. In: CIDR, vol. 7, pp. 412–422 (2007)
14. Balkesen, C., Dindar, N., Wetter, M., Tatbul, N.: Rip: run-based intra-query parallelism for scalable complex event processing. In: Proceedings of the 7th ACM International Conference on Distributed Event-Based Systems, pp. 3–14. ACM (2013)
15. Brenna, L., Gehrke, J., Hong, M., Johansen, D.: Distributed event stream processing with non-deterministic finite automata. In: Proceedings of the Third ACM International Conference on Distributed Event-Based Systems, p. 3. ACM (2009)
16. Chakravarthy, S., Krishnaprasad, V., Anwar, E., Kim, S.K.: Composite events for active databases: semantics, contexts and detection. VLDB **94**, 606–617 (1994)
17. Peng, S., Li, Z., Li, Q., Chen, Q., Pan, W., Liu, H., Nie, Y.: Event detection over live and archived streams. In: Wang, H., Li, S., Oyama, S., Hu, X., Qian, T. (eds.) WAIM 2011. LNCS, vol. 6897, pp. 566–577. Springer, Heidelberg (2011)
18. Zdonik, S., Sibley, P., Rasin, A., Sweetser, V., Montgomery, P., Turner, J., Wicks, J., Zgolinski, A., Snyder, D., Humphrey, M., Williamson, C.: Streaming for dummies (2004)
19. Hirzel, M.: Partition and compose: parallel complex event processing. In: Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems, pp. 191–200. ACM (2012)
20. Schultz-Møller, N.P., Migliavacca, M., Pietzuch, P.: Distributed complex event processing with query rewriting. In: Proceedings of the Third ACM International Conference on Distributed Event-Based Systems, p. 4. ACM (2009)
21. Sadoghi, M., Singh, H., Jacobsen, H.A.: Towards highly parallel event processing through reconfigurable hardware. In: Proceedings of the Seventh International Workshop on Data Management on New Hardware, pp. 27–32. ACM (2011)
22. Hirzel, M., Andrade, H., Gedik, B., Kumar, V., Losa, G., Nasgaard, M., Soule, R., Wu, K.: Spl stream processing language specification. IBM Research, Yorktown Heights, NY, USA, Technical report RC24 897 (2009)

23. Stonebraker, M., Çetintemel, U., Zdonik, S.: The 8 requirements of real-time stream processing. ACM SIGMOD Rec. **34**(4), 42–47 (2005)
24. Golab, L., Özsu, M.T.: Issues in data stream management. ACM SIGMOD Rec. **32**(2), 5–14 (2003)
25. Wang, Y., Cao, K., Zhang, X.: Complex event processing over distributed probabilistic event streams. Comput. Math. Appl. **66**(10), 1808–1821 (2013)
26. Mani, M.: Efficient event stream processing: handling ambiguous events and patterns with negation. In: Xu, J., Yu, G., Zhou, S., Unland, R. (eds.) DASFAA Workshops 2011. LNCS, vol. 6637, pp. 415–426. Springer, Heidelberg (2011)
27. Kawashima, H., Kitagawa, H., Li, X.: Complex event processing over uncertain data streams. In: 2010 International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), pp. 521–526. IEEE (2010)
28. Jiang, Q., Chakravarthy, S.: Scheduling strategies for a data stream management system. Computer Science & Engineering, BNCOD, pp. 16–30 (2004)
29. Sharaf, M.A., Labrinidis, A., Chrysanthis, P.K.: Scheduling continuous queries in data stream management systems. Proc. VLDB Endow. **1**(2), 1526–1527 (2008)
30. Wu, J., Tan, K.-L., Zhou, Y.: QoS-oriented multi-query scheduling over data streams. In: Zhou, X., Yokota, H., Deng, K., Liu, Q. (eds.) DASFAA 2009. LNCS, vol. 5463, pp. 215–229. Springer, Heidelberg (2009)
31. Babcock, B., Babu, S., Datar, M., Motwani, R., Widom, J.: Models and issues in data stream systems. In: Proceedings of the Twenty-First ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, pp. 1–16. ACM (2002)
32. Babcock, B., Babu, S., Datar, M., Motwani, R., Thomas, D.: Operator scheduling in data stream systems. VLDB J. Int. J. Very Large Data Bases **13**(4), 333–353 (2004)
33. Babcock, B., Babu, S., Motwani, R., Datar, M.: Chain: operator scheduling for memory minimization in data stream systems. In: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, pp. 253–264. ACM (2003)