VISIRI - Distributed Complex Event Processing System for Handling Large Number of Queries

Malinda Kumarasinghe, Geeth Tharanga, Lasitha Weerasinghe, Ujitha Wickramarathna, Surangika Ranathunga

VISIRI - Distributed complex event processing system for handling large number of queries

Malinda Kumarasinghe, Geeth Tharanga, Lasitha Weerasinghe, Ujitha Wickramarathna*, and Surangika Ranathunga

Department of Computer Science and Engineering, Faculty of Engineering, University of Moratuwa, Katubedda, Sri Lanka

Abstract. Complex event processing (CEP) systems are used to process event data from multiple sources to infer events corresponding to more complicated situations. Traditional CEP systems with central processing engines have failed to cater to the requirement of processing large number of events generated from a large number of geographically distributed sources. Distributed CEP systems have been identified as the best alternative for this. However, designing an optimal distributed CEP system is a non-trivial task, and many factors have to be considered when designing the same. This paper presents the VISIRI distributed CEP system, which focuses on the problem of optimally processing a large number of different type of event streams using a large number of CEP queries in a distributed manner. The CEP query distribution algorithm in VISIRI is able to distribute a large number of queries among a set of CEP nodes in such a way that the event duplication in the network is minimized while not compromising the overall throughput of the system.

Keywords: Complex Event Processing,Distributed systems,Algorithms

1 INTRODUCTION

Complex event processing (CEP) systems are used to process event data from multiple sources to infer events corresponding to more complicated situations. Traditional CEP systems with central processing engines have failed to cater to the requirement of processing large number of events generated from a large number of geographically distributed sources [2, 4]. Distributed CEP systems have been identified as the best alternative for this.

The middleware of complex event processing systems can be internally built around several distributed complex event processors. These processors cooperate in processing and routing events sent from event sources, and finally in delivering results to the required event sinks [1]. A processor makes use of a CEP engine internally to process the incoming events. This is done using CEP queries. Query is a mechanism for extracting events that satisfy a rule or a pattern, from incoming events. A query may contain operators such as filter, window, join, pattern and sequence.

* Authors names are in alphabetical order.

However, simply having a middleware is not sufficient to have a distributed CEP system. Crucial decisions have to be taken on how to distribute the load among the complex event processing nodes in the system. There are two approaches in distributing the processing load among nodes : operator distribution and query distribution [1, 4]. Operator distribution refers to the approach of dividing a query into a distinct sequence of steps to handle complex queries. Each step is allocated to a node in the system. Query distribution allocates a set of queries among the nodes of the distributed system. There are several application scenarios in distributing complex event processing as well: (1) handling large number of CEP queries and event streams, (2) handling event streams that have high event frequencies and/or large events, and (3) handling complex resource intensive queries.

VISIRI is a distributed complex event processing system for handling large number of queries and large number of different types of event streams. It presents a lightweight middleware for a distributed CEP system, and the communication among the nodes in the system is achieved via Hazelcast [3].

Its approach on distributing the processing load is by query distribution. Distributing queries in an optimal manner is a NP-Hard problem [8]. When distributing a large number of queries among a set of CEP nodes, many concerns have to be addressed. These include the throughput of the overall system, network latency in transmitting events from event sources to CEP nodes, balancing resource utilization at processing nodes, and reducing event duplication when transmitting events from sources to nodes. Existing research on query distribution focused on optimizing a subset of these concerns. For example, the COSMOS project [8] focused on reducing network latency and reducing communication between the nodes. The SCTXPF system [4] focused only on reducing the event duplication network traffic within the system.

The most important aspect of the VISIRI system is its query distribution algorithm. It focuses on reducing the event duplication network traffic within the system while making sure that the processing load is evenly distributed among the nodes. Load on a node is calculated by considering the cost of the CEP queries. The cost of queries is calculated by the cost model, which is based on the empirical studies done by Mendes et al. [5] and Schilling et al. [6]. With this query distribution algorithm, VISIRI balances the resource utilization of the CEP nodes, so that no node gets overloaded (given that there is no sudden change in the incoming event streams) and adversely affects the throughput of the overall system.

The rest of the paper is organized as follows. Section 2 discusses literature related to query distribution in distributed CEP systems, and empirical studies on the cost of CEP queries. Section 3 presents the design and implementation of the VISIRI distributed CEP system, along with detailed descriptions of its query distribution algorithm and the cost model. Section 4 presents evaluation results and finally section 5 concludes the paper.

2 Related Work

SCTXPF [4] and COSMOS [8] are examples of distributed CEP systems based on query distribution.

The query distribution algorithm in the SCTXPF system is optimized for large number of complex event processing queries and for very high events rates. It parallelizes the event processors (EPs) and then allocates certain number of CEP queries to each of them. In this algorithm, queries with common sets of attributes are assigned to the same node. This minimizes the number of EPs that need the same event streams and thus minimizes the number of multicasts. Therefore the system is able to reduce the event duplication network traffic within it. However, this algorithm assumes that all the queries require the same processing power, which is not the case most of the times. Even if two queries consume the same set of attributes, their computational intensiveness (i.e. the amount of computer resources it requires) could have very large differences. This is because the computational intensiveness of a query heavily depends on its operators. However, when the set of queries are simply divided among the nodes by the SCTXPF algorithm, each node has the same number of queries, and some nodes that have lot of high cost queries could easily get overloaded.

In contrast to the SCTXPF algorithm, VISIRI query distribution algorithm considers the cost of each query when distributing the queries among the processing nodes. This is the main difference between the two algorithms.

The COSMOS distributed CEP system employs a heuristic based graph mapping algorithm to distribute the load among CEP nodes. In this approach, processors are represented by vertices and communication latency is represented by the weight of the edges. When distributing queries, this network latency is considered. Furthermore this algorithm filters out the events from the initial nodes so that network traffic of the internal system is reduced. This model suits a problem where a particular set of queries are interested in the output of another set of queries. However, this algorithm also does not consider the cost of individual queries when distributing them among CEP nodes.

Schilling et al. [6] have studied about the cost of executing different query operators in their empirical study. According to their study, filter rules have the lowest latency and logical rules have much higher latency. Temporal windows have the highest latency. Therefore such temporal windows should be given the highest weight when considering the latency and computational intensiveness. When considering the cost versus size of the history of the temporal windows, this study suggests an exponential growth in cost. Cost versus the number of attributes has no such growth and it is almost the same for any number of attributes. However, when the number of queries in a CEP node increases, the cost increases linearly.

Furthermore Marcelo et al. [5] have studied about different engine types and how they perform on different query types. They have identified that sliding windows and jumping windows have huge differences in performance, where sliding windows take much computational power.

3 VISIRI Distributed CEP System

Figure 1 shows the VISIRI high level architecture. It consists of event sources, dispatchers, CEP nodes and event sinks. Here, event sources generate the low-level events, and the complex events identified by the CEP nodes are received by the event sources.

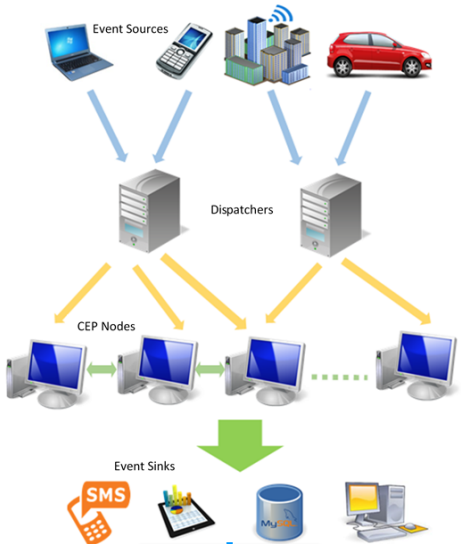

Fig. 1. VISIRI high level architecture

VISIRI CEP system assumes nearly homogeneous CEP nodes. User can freely select one CEP node and deploy the queries. After deploying the queries, that particular CEP node plays the role as the main node, which executes the query distribution algorithm (discussed later in this section). This query distribution algorithm distributes the queries among all the active CEP nodes in the system. Then the allocated queries are automatically deployed in the CEP nodes and the dispatcher is notified about the query allocation.

Dispatcher creates the forwarding table according to query allocation. Forwarding table is a map of event stream ID to the list of CEP node IPs. Job of the dispatcher is to forward the relevant event streams only to the relevant CEP nodes. Thus in contrast to directly sending events from sources to CEP nodes, employing a dispatcher reduces network traffic. Event sources send event streams to the dispatcher and using its forwarding table dispatcher forwards these event streams to relevant CEP nodes where event processing happens according to the deployed queries in CEP nodes. After processing the event streams, the resulted event streams from the CEP nodes are sent to the event sink.

3.1 Low level architecture

Figure 2 shows how the low-level architectural components are integrated within the system. Arrows show the flow of the event streams among different components.

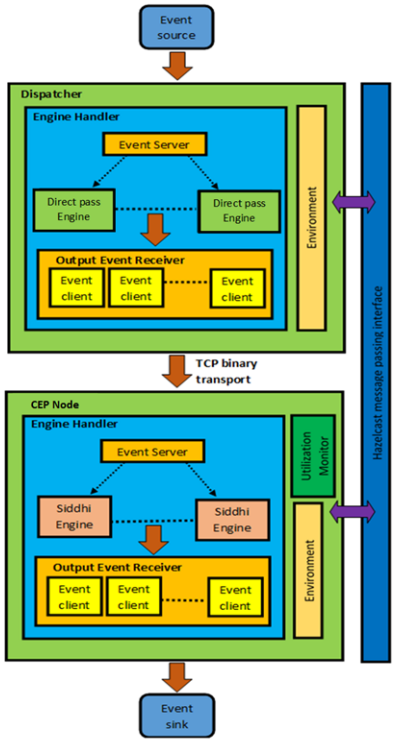

Fig. 2. VISIRI low level architecture

- Siddhi CEP engine [7] - In a CEP node, the light-weight Siddhi CEP engine is used as the processing engine. As shown in Figure 2, there is a Siddhi engine per one query in one CEP node. Siddhi engine processes the input event streams and results the output event stream. This architecture allows our system to extend to make dynamic adjustments on query distribution in runtime.
- Event client/Server - Uses TCP binary communication protocol to transport event streams.
- Environment - Each node in the system(processing node/dispatcher node) includes a its own Environment component and all data sharing tasks and message passing tasks between nodes are achieved via this component. It uses hazelcast as an intermediate interface for this communication.

3.2 Query distribution algorithm

The VISIRI query distribution algorithm considers the SCTXPF algorithm as its starting point. As described earlier, the SCTXPF algorithm aims at minimizing the number of event processors that need the same event streams and also reducing the difference between the numbers of queries deployed in the event processing nodes. The major improvement of the VISIRI algorithm over the SCTXPF algorithm is that it considers the cost of individual queries to make sure that queries with higher costs are not deployed in the same node. This prevents one node getting overloaded while some other nodes of the system are under-utilized.

VISIRI algorithm takes following inputs when distributing queries[1].

- Set of queries to be distributed
- Set of processing nodes and dispatchers
- Queries currently allocated for each node

The algorithm considers the following important factors:

- Costs of the queries (depending on the complexity of query operators)
- Number of existing queries in each node
- Number of common event types required for the query

A suitable cost model calculator is required to measure the complexity and costs of the queries. These costs may also depend on the underlying implementation of the complex event processing engine. However, this aspect is not considered in the VISIRI cost model. The cost model is discussed in the next section.

Algorithm 1 gives the pseudo code of our query distribution algorithm.

Line 3 finds the minimum number of queries currently assigned to a single node.

Lines 5-9 remove all nodes that have queries above a certain threshold. This is to balance the overhead of having large number of queries in the same node.

Lines 11-18 refer to the procedure to find the minimum total cost of a node. Here the total cost of a node is calculated by taking the sum of the costs of the queries deployed in that node.

In lines 20-24, all nodes having costs more than a certain threshold value are removed from the candidate list to balance the cost distribution among the nodes.

Lines 26-38 finds the nodes having maximum number of input streams in common with the given query. For example, if the query has input streams s1, s2 and s3, and queries already deployed in a Node A have s2, s3 and s4 as input streams, then node A and the query has 2 common input streams (s2 and s3). Here the input streams of a node are the union of all input streams of deployed queries. In this code segment, nodes with maximum number of common input streams are selected so that the number of new events that need to be sent over the network as inputs is minimized. This reduces event duplication and preserves network bandwidth. Here we assume all event types arrive in same frequency.

[1] The algorithm iteratively distributes queries, with one iteration per query.

Algorithm 1 Query distribution algorithm

```
Require: Query q, Node[] nodes
candidates = nodes;
//find minimum queries
min-queries = min(nodes[0].queryCount,nodes[1].queryCount,...)
//filter nodes with too many queries
for node in candidates do
    if node.queryCount >minQueries + QueryVariability then
        candidates.remove(node)
    end if
end for
//find minimum total cost
minCost = infinity
for node in candidates do
    cost = sum(node.queries[0].cost,node.queries[1].cost, ...)
    node.cost = cost
    if minCost >cost then
        minCost = cost
    end if
end for
//filter nodes with too much cost
for node in candidates do
    if node.cost >minCost + CostVariability then
        candidates.remove(node)
    end if
end for
//find maximum common event types
qInputs = q.inputStreams
maxCommonNodes =[]
maxCommonInputs = 0
for node in candidates do
    node.allInputs = union(node.queries[0].inputStreams,node.queries[1].inputStreams.)
    commons = count(intersect(qInputs,node.allInputs))
    if maxCommonInputs == commons then
        maxCommonNodes.add(node)
    else if maxCommonInputs >commons then
        maxCommonNodes.clear()
        maxCommonNodes.add(node)
        maxCommonInputs = commons
    end if
end for
candidates = maxCommonNodes
//select one randomly from the candidates
target = random.select(candidates)
return  target
```

3.3 Cost model

In order to calculate the cost of a given query, we have developed a cost model that gives a numeric value for a query based on the empirical studies done by Schilling et al. [6] and Marcelo et al. [5].

The cost model first identifies the queries that have filtering parts and assigns them cost values depending on the number of filtering attributes. Furthermore, a cost value is assigned to the number of attributes in the input stream definitions and the output stream definitions. This is because the literature [6] suggests that when the number of attributes in the event streams for a particular query increases the resource requirement for that query increases. Apart from that, the number of input streams and the output streams count is also added to the cost value. These values are expected to give an indication of the impact of handling large number of event streams in a query.

The cost model gives a much higher priority to queries with windows. Depending on the window length, an exponential cost is added to the query so that windows with higher length will get a higher number. Our cost model can support window queries of time or length with the expiration mode of sliding or batch.

Finally the logarithmic value of the total cost value is obtained so that the cost value can be restricted to a more meaningful range. Our cost model still does not give exact accurate values for pattern queries and join queries, but a simple numerical value depending on the aforementioned factors is given to them. In order to obtain a more accurate number, a good performance analysis has to be done on those types of queries.

Table 1 shows how the cost value changes for three different sample queries with different time windows[1].

	Query	Cost Value
1	from cseEventStream[price==foo.price and foo.try <5 in foo] select symbol, avg(price) as avgPrice	3.1986731175506815
2	from car [Id >=10]#window.length(10000) select brand,Id insert into filterCar;	13.815633550400394
3	from StockExchangeStream[symbol == IBM]#window.time(1 year) select max(price) as maxPrice, avg(price) as avgPrice, min(price) as minPrice insert into IBMStockQuote for all-events	31.664045840884167

Table 1. Sample queries and their costs generated by the cost model

[1] Queries are expressed in Siddhi event processing language.

4 Evaluation

4.1 Event and query model

Since our target is to handle large number of queries, we used randomly generated events and randomly generated queries to evaluate our system. Our system is capable of configuring the number of input event streams and the number of output stream definitions. For evaluation purposes, we used 1000 event stream definitions and 500 output stream definitions. When generating the random queries we use those input stream definitions and produce results to the output stream definitions.

In our query model we initially generated a set of simple queries with around two maximum filtering conditions and with only the length batch windows. But later we increased the complexities of the queries to get the proper advantage from our cost estimation model.

In our random query generator, several types of queries such as filter queries, window queries and windows with filter queries can be created. A query may contain either one filtering condition or two filtering conditions. In the window queries scenario, the random query generator gives either length windows or length batch windows and it outputs the aggregated result like maximum, sum or average value within the window. The window length is also given by a random value.

Below given are some of the sample Siddhi queries that were generated by the random query generator.

```
from stream2
  [attr3 < 45.18 and attr1 > 71.6 and attr5 > 63.37
   and attr4 > 35.71 and attr1 > 83.35 and attr2 > 89.95
   and attr3 < 50.0 and attr2 < 15.83]
   select attr1 insert into stream46

from stream2
  [attr4 < 94.05 and attr1 > 83.05 and attr5 > 46.27
   and attr2 < 34.01 and attr2 < 32.74 and attr3 > 59.25
   and attr5 > 94.62 and attr1 > 4.06]#window.lengthBatch(104)
   select  max(attr1) as attr1, max(attr2) as attr2
   insert into stream3

from stream2 # window.lengthBatch(210)
   select  max(attr1) as attr1, max(attr2) as attr2
   insert into stream14
```

The queries were generated with the same seed value for the random generator therefore the same set of queries is obtained all the time for the same number of queries, input definitions and output definitions. Having exact queries and events was important have a fair comparison when evaluating different algorithms and over different configurations. Event sources were configured to generate events in maximum rate possible.

4.2 Query Distribution Algorithm Comparison

To evaluate our algorithm we compared it with the SCTXPF algorithm and a random query distribution algorithm. When comparing the algorithms, mainly two factors were considered: total query execution cost variance and event duplication. As the execution cost variance we measured how much variance the processing nodes have when the queries are distributed with respect to the estimated cost values. Cost threshold value (highest total cost of the queries deployed in a CEP node) for our algorithm was kept at 400 while keeping query count threshold (highest number of queries deployed in a CEP node) at 80 for both our and SCTXPF algorithm. Those values were selected to get maximum performance from the machines we used for the performance evaluation later in the evaluation process.

Multicasting of events from dispatcher to CEP nodes is a critical factor for network overhead of the system. Our algorithm focuses on minimizing the number of CEP nodes that needs same type of events by placing similar type of queries in a single node. Figure 3, Figure 4 and Figure 5 show how the event duplication changes in the system for the three different algorithms for 1000, 5000 and 10000 queries.

Fig. 3. Event stream duplication for 1000 queries

As can be seen in the figures, our algorithm and SCTXPF algorithm have given similar results for the event duplication as expected. Random algorithm shows clear difference in event duplication, which suggests that randomly distributing queries leads to network traffic increase within the system. Furthermore when the number of queries increases, the event duplication almost remains same for our algorithm and SCTXPF algorithm but in the random query distribution algorithm it increases drastically.

Fig. 4. Event stream duplication for 5000 queries

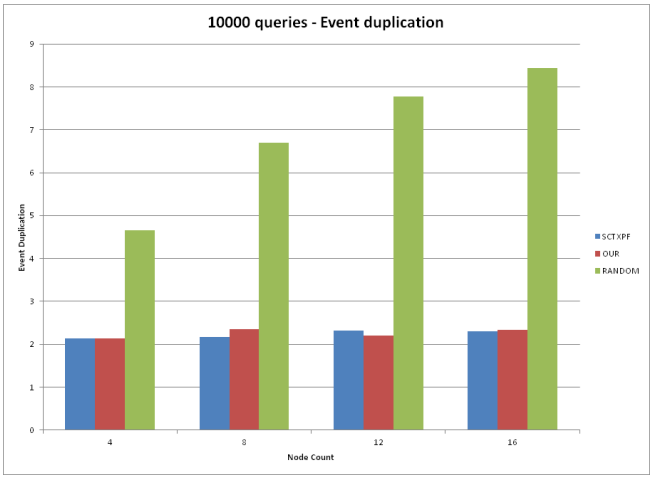

Fig. 5. Event stream duplication for 10000 queries

When the cost variance among the processing nodes is considered, our algorithm was able to gain a considerable advantage over the other two algorithms when the number of queries increases. Figure 6, Figure 7 and Figure 8 show how the cost variance behaves when the number of nodes increases for 1000, 5000 and 10000 queries.

According to these results we can conclude that when the number of queries increases, our algorithm is able to deploy the queries among the processing nodes with minimum cost variance. Therefore all the processing nodes will receive

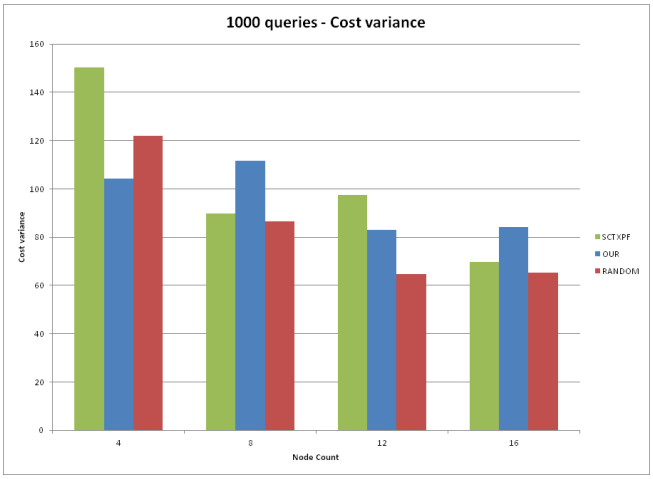

Fig. 6. Total cost variance for 1000 queries

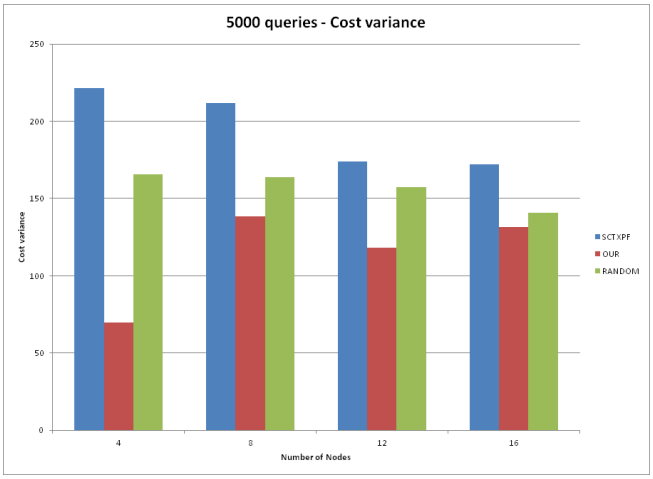

Fig. 7. Total cost variance for 5000 queries

queries with relatively equal estimated processing cost according to our cost model.

4.3 Performance evaluation

For this performance evaluation we used the same event and query model that we used for the algorithm evaluation described above in the section 4.1.

System configurations: For this evaluation we have used a computer lab that has Core i3 machines of 3.2 GHz. The operating system was Ubuntu 12.04 32 bit

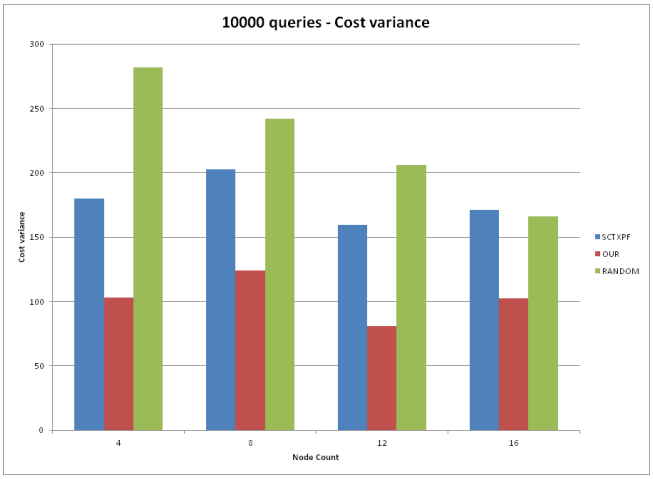

Fig. 8. Total cost variance for 10000 queries

version. Each machine had 2GB RAM and the machines were connected using a 100Mbps Ethernet connection.

Results: For the performance analysis we have sent 15 sets of 1,000,000 events through the system and evaluated the throughput by averaging the total time taken for processing those sets of events.

Initially we evaluated the system for 5000 simple queries, which had around two filtering conditions and smaller length batch windows. Figure 9 shows results for this set of queries. For this set of queries, our algorithm and the SCTXPF algorithm performed in a similar manner. This is because for those simple queries our algorithm was not able to get a clear advantage from the cost model we have generated. However, both our algorithm and the SCTXPF algorithm perform better than the random query distribution algorithm.

As shown in Figure 10, the throughput changes according to the number of nodes for 2500 queries. There is a clear improvement of throughput in our algorithm for this case.

Figure 11 shows how our system behaves in the case of 5000 queries and both the algorithms show less improvement when moving from 8 nodes to 12 nodes.

Figure 12 is from the results that were taken for 10000 queries. We were not able to run the case of four nodes due to the low memory capacities of the machines. And we observed a slight decrement of throughput from nodes 8 to 12.

Results evaluation: According to the results, it can be said that our algorithm is able to deploy queries among the set of processing nodes with minimum cost variation while keeping event duplication at a low level when compared with the SCTXPF and random algorithms.

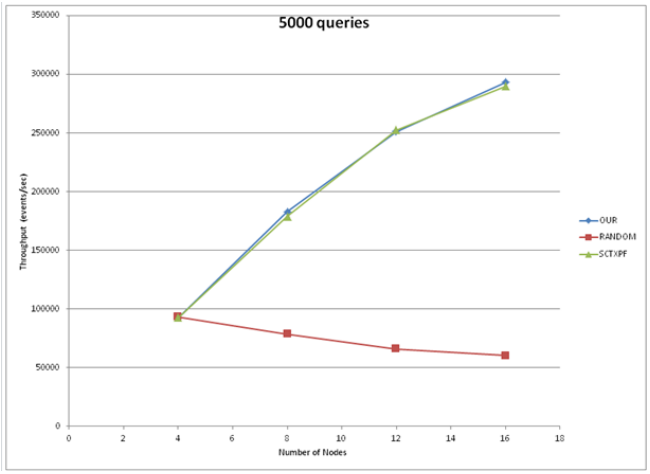

Fig. 9. Throughput for simple 5000 queries

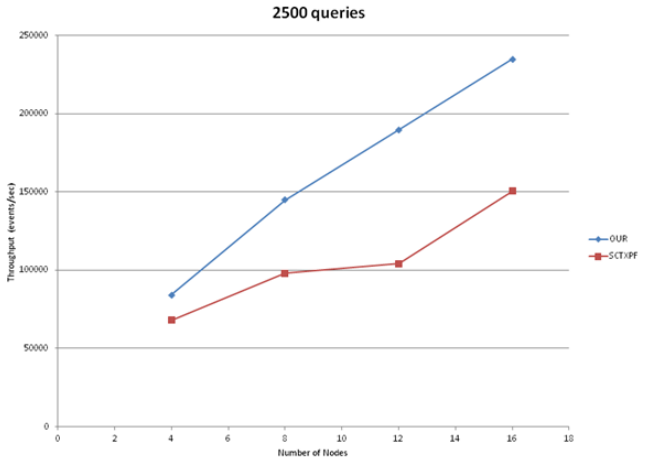

Fig. 10. Throughput for 2500 queries

Furthermore when system throughput is considered, our algorithm has a clear advantage over the SCTXPF for all three cases- 2500 queries, 5000 queries and 10000 queries. Also we can observe that in that evaluation scenario, almost in all the cases when the number of nodes increases the throughput of the system with our algorithm increases. Therefore with higher number of nodes our algorithm is able to provide much higher throughput. However bottleneck situations with respect to network bandwidth at event sinks may arise when the number of nodes increases due to large number of queries being processed and all of them

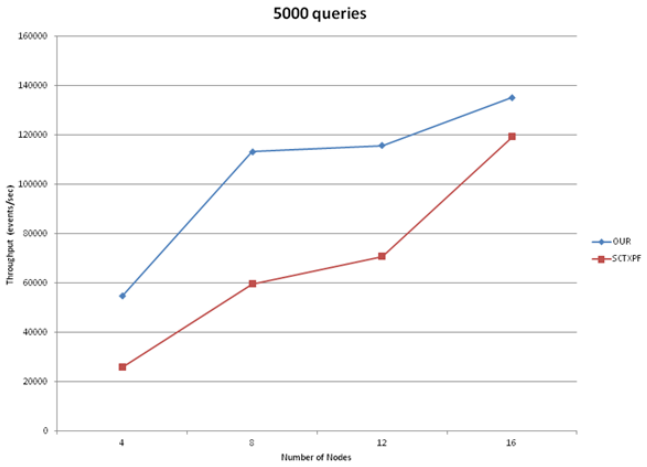

Fig. 11. Throughput for 5000 queries

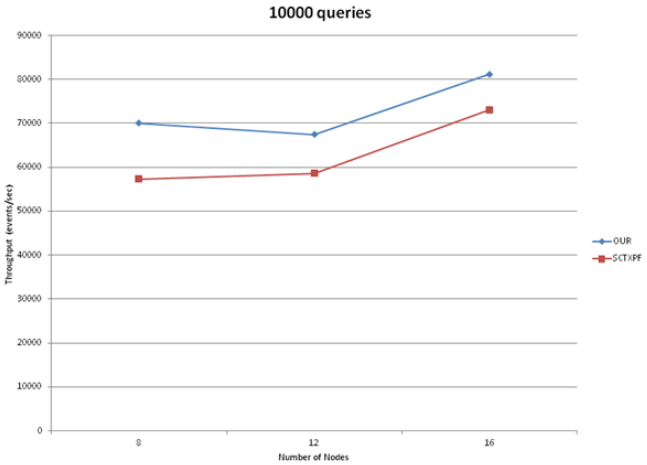

Fig. 12. Throughput for 10000 queries

producing outputs. In the case of 10000 queries when increasing nodes 8 to 12 we can observe this kind of scenario.

5 Conclusion

This paper discussed the architecture of the VISIRI distributed CEP system, which aims at handling large number of queries and large number of different types of event streams. It includes a query distribution algorithm that takes event

stream duplication and the estimated query execution cost into consideration when allocating queries among a set of processing nodes.

With that query distribution algorithm, VISIRI system is able to keep the event stream duplication below a certain level while the total cost variance among the processing nodes is kept low when compared to some other algorithms for distributing number of CEP queries. Furthermore we evaluated our system for the performance by considering the throughput as the measuring factor and our algorithm had a clear advantage over the existing algorithms.

As a future enhancement, VISIRI can be improved to support query rewriting at the dispatcher level so that unnecessary events can be filtered from the dispatcher thus reducing the internal network traffic further. Apart from that, the VISIRI system architecture can also be extended to support heterogeneous event processing engines so that different types of queries can be processed by different processing engines according to the types of queries they are best at processing. Furthermore our query distribution algorithm can also be extended to support factors such as network latency. However, we once again emphasize that coming up with an optimal query distribution algorithm that considers all these factors is a NP-hard problem.

References

1. Gianpaolo Cugola and Alessandro Margara. Deployment strategies for distributed complex event processing. *Computing*, 95(2):129–156, 2013.
2. Opher Etzion and Peter Niblett. *Event processing in action*. Manning Publications Co., 2010.
3. Hazelcast.com. Hazelcast - leading in-memory data grid IMDG, January 2015.
4. Kazuhiko Isoyama, Yuji Kobayashi, Tadashi Sato, Koji Kida, Makiko Yoshida, and Hiroki Tagato. A scalable complex event processing system and evaluations of its performance. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, pages 123–126. ACM, 2012.
5. Marcelo RN Mendes, Pedro Bizarro, and Paulo Marques. A performance study of event processing systems. In *Performance Evaluation and Benchmarking*, pages 221–236. Springer, 2009.
6. Björn Schilling, Boris Koldehofe, Udo Pletat, and Kurt Rothermel. Distributed heterogeneous event processing. DEBS, 2012.
7. Sriskandarajah Suhothayan, Kasun Gajasinghe, Isuru Loku Narangoda, Subash Chaturanga, Srinath Perera, and Vishaka Nanayakkara. Siddhi: A second look at complex event processing architectures. In *Proceedings of the 2011 ACM workshop on Gateway computing environments*, pages 43–50. ACM, 2011.
8. Yongluan Zhou, Karl Aberer, and Kian-Lee Tan. Toward massive query optimization in large-scale distributed stream systems. In *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*, pages 326–345. Springer-Verlag New York, Inc., 2008.