Real-time Processing of Rule-based Complex Event Queries for Tactical Moving Objects

Yihuai Liang[1], Jiwan Lee[1], Bonghee Hong[1] and Woochan Kim[2]
[1]*Department of Computer Science and Engineering, Pusan National University, Pusan, Republic of Korea*
[2]*Agency for Defense Development, Republic of Korea*

Keywords: Continuous Processing, Complex Event Processing, CQ Index, Dynamical Rule, Tactical Moving Object.

Abstract: Target data for tactical moving objects are streaming data collected in real time via radar, sonar, and other sensors. A system of continuous complex event query with dynamic rule definitions and high performance is needed to process that target data. We develop a continuous complex event query system with rule-based layered architecture. A continuous processing flow is decomposed into four modules hierarchically, which are event filtering, event capture, Continuous Queries (CQ) and Complex Event Processing (CEP). Each module has its responsibility but works together for a completed continuous processing flow. This paper shows that it is possible to dynamically insert, update, delete and search rule specifications of each layered modules through the decomposition of the whole system. Many rules are registered in the system for processing input event data continuously in real time. To improve the performance of getting matching CQs for each incoming event, CQ index is developed. Finally, experimentations and performance evaluations are carried out.

1 INTRODUCTION

Different from traditional database management system, data stream management system (DSMS) processes the input stream data then produces the output results continuously in real time. A data stream is a sequence of tuples that are generated continuously and incrementally over time with no end. Many applications process high volumes of streaming data.

If in the following situations, it should be considered to use DSMS: (1) Too large amounts of interesting data to store in hard disks. For example, data from sensor networks with a massive number of measurement points. (2) Require real-time analysis and feedbacks. In a DSMS, the processing model is push-based or data-driven. It evaluates persistent queries on transient, append-only data and outputs results automatically if incoming data meet query conditions. There is a trade-off between latency and accuracy, because of processing single-pass stream data in main memory.

Our target data from tactical moving objects are stream data collected in real time via radar, sonar, and other sensors. The data have the following characteristics: (1) Temporal generating and dynamic changing, (2) Detected data only contains information on simple events. The data might be duplicated, missing, outlier and so on. (3) Large volume from a massive number of measurement points. The target data should be processed to detect emergencies within 1 second. Our motivation is to analyze real-time situations and make alert decisions of tactical moving objects. We develop a DSMS with the ability of complex event query over the target data. By using the system, we query complex events and detect potential threats in real time.

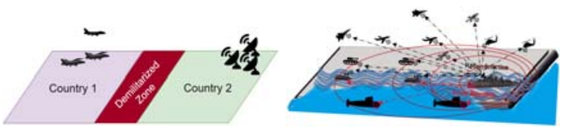

Figure 1: Scenarios of fighter invasion and detection by radars and sonars.

From inputting source data to performing a complex event query, the system goes through a series of processing procedures, including data filtering, data adapting, event tracking, meaning refinement, continuous queries, and complex event queries. The system could contain hundreds of given rule specifications and runs based on them. It should

Liang, Y., Lee, J., Hong, B. and Kim, W.
Real-time Processing of Rule-based Complex Event Queries for Tactical Moving Objects.
DOI: 10.5220/0007735700670074
In *Proceedings of the 4th International Conference on Complexity, Future Information Systems and Risk (COMPLEXIS 2019)*, pages 67-74
ISBN: 978-989-758-366-7

support adding, updating and deleting rule specifications dynamically without restarting the whole system. Several rules might work together for a complete processing flow. So how to make it easy and flexible to add, update or delete a rule without affecting other rules? Finding a flexible and dynamic way to define, organize and manage the rule specifications is one of the motivations in this paper.

To find out matching CQs for an arriving event, a naïve approach is to check conditions of each CQ one by one. It is simple but time-consuming. In our system, we build a CQ index to solve this issue.

In conclusion, our contributions in this paper are:

- To propose a layered rule-based architecture for complex event queries.
- To define rule specifications based on the decomposition of the layered architecture. In order to make it flexible and dynamic to define, add, update and delete a rule specification.
- To develop a CQ index by using R*-tree to solve the performance issue of CQ matching.

The rest of this paper is organized as follows. Section 2 presents an event processing flow. Section 3 presents rule definitions. Section 4 presents the CQ index. Section 5 presents experiment and section 6 shows results of performance evaluation. Section 7 presents related work. Finally, we conclude the paper in section 8.

2 EVENT PROCESSING FLOW

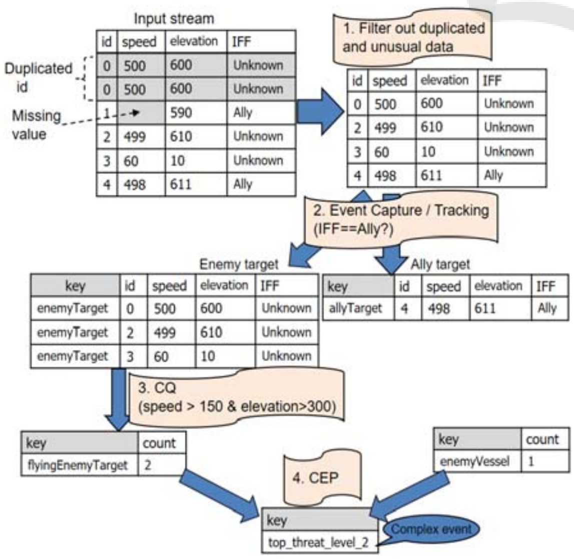

Figure 2: An example of a complex event processing flow.

Before the explanation of event processing flow, let us take an **example** shown in Figure 2. First, we filter out duplicated and unusual input stream data. Second, we track each data and assign a new meaning to them. Here, if a data whose field *IFF* equals to "Ally", it belongs to "allyTarget", else belongs to "enemyTarget". Next, we perform CQ to count the data of "enemyTarget" based on condition "*speed>150 & elevation > 300*". Finally, we perform CEP to get a complex event "top_threat_level_2".

The data from input to output goes through a continuous processing flow. We decompose the flow into four sub-modules to make the flow clear and simplify rule definitions. A rule processing flow goes through four steps as follows:

Step 1: Filter out duplicated and unusual incoming data.

Step 2: Capture and track events to assign new meanings to them.

Step 3: Continuous query events using operators based on query conditions and window. The input data is from step 2.

Step 4: Perform complex event queries over the input simple events from step 2 or step 3. Its results can trigger pre-defined response actions.

In Figure 3, we show the architecture and event processing flow of our system. There are four layers/modules, and each has its own registered rules and output results. They connect together by their output results to form a complete flow of continuous complex event queries.

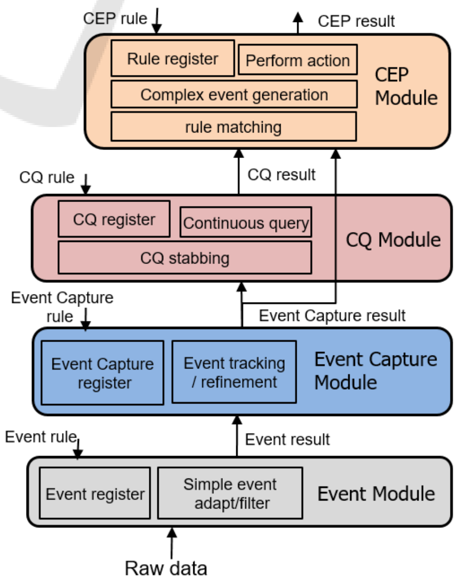

Figure 3: Layered architecture on continuous complex event queries.

Each module has its responsibilities. *Event Module* is responsible for filtering incoming raw data and adapting them to simple events. *Event Capture Module* is responsible for tracking and refining the simple events from the *Event Module*. For example, as for simple event that meets the condition *speed>150km/s & elevation>100m*, we could assign it new meaning: it is an event of a flying target. *CQ Module* is responsible for continuous queries, consisting of sliding/tumble windows, stateful/stateless operators, and query expressions and so on. *CEP module* is responsible for complex event processing. It derives complex events from multiple simple events. The module takes the output results of the *Event Capture Module* and *CQ Module* as input data, then performs rule matching and responses actions.

3 RULE-BASED STREAM DATA PROCESSING

In the previous section, we talk about layered architecture. The system is decomposed into four modules. Each module has its own registered rules. Users tell the system what to do through inputting rules for each module. In this section, we talk about how to define rule specifications.

3.1 Rule Format of Event Filtering

The rule of this module is to filter out duplicated and unusual data. We define the rule format of the *Event Module* as follows. ***IF*** clause defines the unduplicated fields and not unusual fields for input data. It is the filter condition of this rule. ***FROM*** clause defines the DDS topic name. ***THEN*** clause defines the name of output results that satisfy the filter conditions.

IF <not duplicate(target field)> AND <not unusual(target field)*>*
FROM <DDS Topic>
THEN <target object>

Example: filter out the data if it has duplicated *id* or an unusual value of *speed* field (Figure 4).

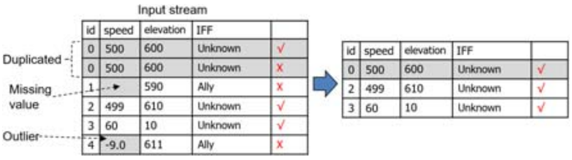

Figure 4: Example of event filtering.

IF not duplicate(id) AND not unusual(speed)
FROM moving_object
THEN target_object

3.2 Rule Format of Event Capture

Event Capture Module is responsible for tracking and refining the simple events from the *Event Module*. We define the rule format of the *Event Capture Module* as follows. ***FROM*** clause defines the input target object from the *Event Module*. If the object satisfies the conditions in ***IF*** clause, it will be assigned a new meaning.

IF <condition>*
FROM <target object>
THEN <target object with new meaning>

Example: tracking enemy target objects if data meets the condition "*IFF='unknown' OR IFF='enemy'*". (IFF, Identification Friend or Foe) (Figure 5)

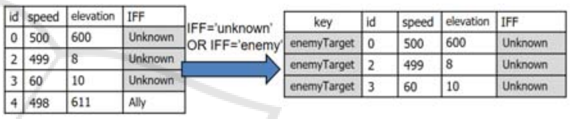

Figure 5: An example of event capture.

IF IFF='unknown' OR IFF='enemy'
FROM moving_object
THEN enemy_object

3.3 Rule Format of CQ

CQ Module is responsible for continuous queries. A CQ rule consists of query conditions, input data, a window, object field names for projection and operators. A window (sliding window or tumble window) buffers data for supporting aggregate operations, defined in the ***WINDOW*** clause.

IF <condition>*
FROM <target object from Event Capture Module >
WINDOW <length, trigger>
THEN <field name for projection, operator*>*

Example: query the count of flying event belonging to an enemy in the last 1 second and output the results every 1 second. (Figure 6)

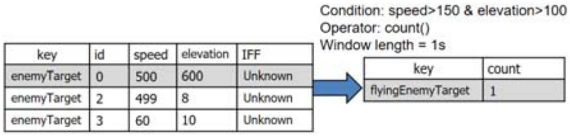

Figure 6: An example of CQ.

IF speed>150km/s AND elevation>100m
FROM enemy_object
WINDOW length=1000ms, trigger=1000ms
THEN count

3.4 Rule Format of CEP

We define the rule of CEP to indicate how to denote a complex event. To derive a complex event from multiple simple events, it is necessary to analyze the relationships among different types of simple events. The format of CEP rule is almost the same as CQ rule. However, the input target data defined in the ***FROM*** clause is from the *Event Capture Module* or *CQ Module*. We define it as follows:

IF <condition>*
FROM <objects from Event Capture or CQ Module>
WINDOW <length, trigger>
THEN <complex event>

Example: derive complex event *top_threat_level_2* if simple events *approachingAirplane* and *approachingMissile* both exist in the last 2 seconds. Perform the query every 2 seconds (Figure 7).

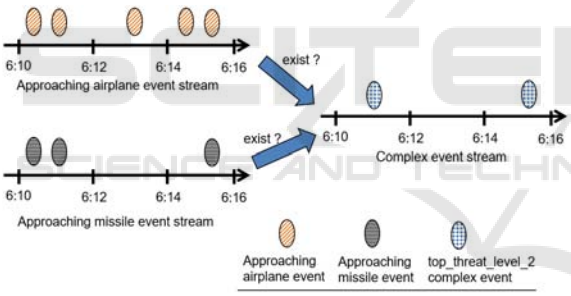

Figure 7: An example of a complex event query.

IF exist(approachingAirplane)
AND exist(approachingMissile)
FROM approachingAirplane, approachingMissile
WINDOW length=2000ms, trigger=2000ms
THEN top_threat_level_2

4 CQ INDEX

For an incoming event, we need to find out its matching CQs based on the CQ conditions. Because only the matching CQs should process the event. We call the procedure as CQ stabbing (Figure 8).

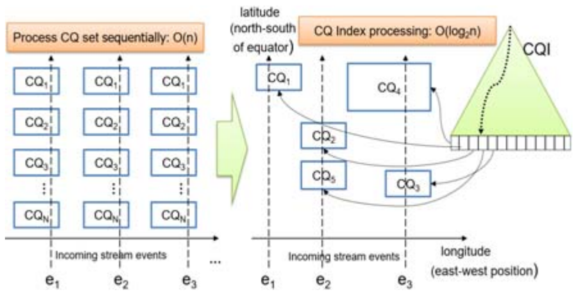

Figure 8: CQ stabbing for incoming events.

The time complexity is O(n) if we make CQ stabbing by checking conditions of each CQ one by one. So we are thinking whether there is a way to get matching CQs directly based on the field values of the event and condition values in each CQ. To achieve that, we use R*-tree as CQ index.

There are three steps to build and use a CQ index (Figure 9):

- Extract values provided by expressions of CQ. Use the values to build or update the R*-tree index.
- For an incoming event, use its values to search in the index, and get candidate CQs.
- Not all candidate CQs from step 2 match the event. So next, check their conditions one by one to find out the final matching CQs.

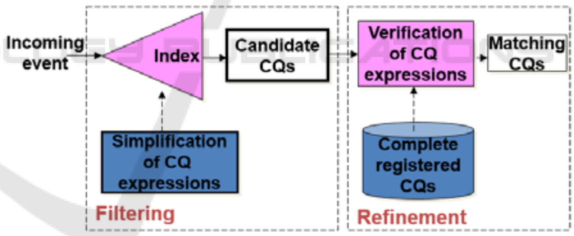

Figure 9: Filtering and refinement strategy of CQ index.

R*-tree could be a multi-dimensions index. Two examples are shown in Figure 10. In the first sub-figure, the index only stores values of two fields in two dimensions, *speed*, and *elevation*. For example, a CQ whose condition is "*30 < speed < 50 & 10 < elevation < 30*" can store in the index. One rectangle indicates one CQ, while one point in the sub-figure indicates one incoming event. Therefore, for an arriving event, to get the candidate CQs, it only needs to find out all rectangles that contain the point. It is almost the same if the index is in three dimensions, which is shown in sub-figure 2 as an example.

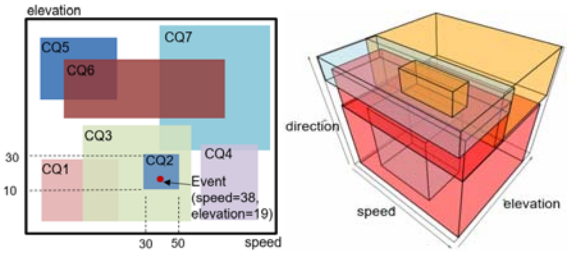

Figure 10: Examples for R*-tree index with (1) two dimensions, (2) three dimensions.

The data stored in an R*-tree are all points or all regions. The data type of each dimension in an R*-tree is the same. However, some predicates in a CQ condition includes different data types. For example, the condition "*IFF='enemy' AND 30<speed<100 AND elevation < 10*" includes string and integer two data types. In such a case, the shapes in the index are not rectangles or cuboids. It is not suitable to add the CQ condition to the index directly. We take a strategy to solve this problem without modifying the implementation of R*-tree, which is described in the next section.

5 IMPLEMENTATION FOR SUPPORTING REAL-TIME PROCESSING

We have already implemented a prototype system in C++ program language, running in a single computer with Windows OS. We use a flexible structure to organize processing flow among continuous processing (CP) units. A CP could be a CEP, CQ, event capture processing or event filtering. Two CPs connect through a queue (Figure 11). The whole graph is a directed acyclic graph (DAG).

How to maintain the data in each queue shown in Figure 11 on the right? Our strategy is consuming the data in the queues as soon as possible. First, we maintain a *"CP-queue"* to store those CPs whose input streams are not empty. Second, we create a round-robin scheduler running in a new thread. The scheduler does two things: one is popping a CP from the head of the *CP-queue*, and two is consuming data from its connecting input queue (called *input-queue*) until empty or consuming more than 100 events. If the *input-queue* is not empty after popping out more than 100 events, push back the CP to the tail of the *CP-queue*, and turn to process another CP. Let us take an example. In Figure 11, we assume all CPs are stateless. Initially, only the inputs of CP1 and CP2 are not empty. So push them to the *CP-queue*. The round-robin scheduler pops out CP1, processes it and stores results in the *input-queue* that connects to CP3, causing the input of CP3 becoming not empty. So push back CP3 to the tail of *CP-queue*. Next, pop out CP2 and do the same procedure until the *CP-queue* becomes empty.

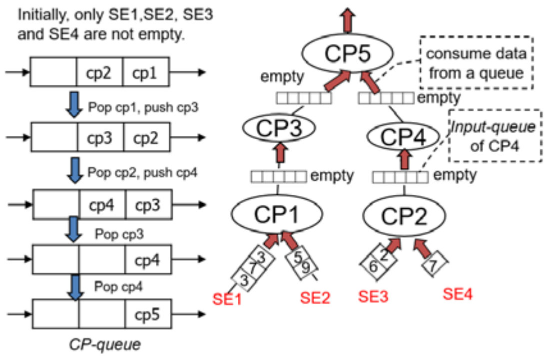

Figure 11: Implementation of a continuous processing flow.

A CP could be stateless or stateful. If an operation whose calculation result is affected by history processed data, it is stateful, such as aggregate operations *Sum* and *Count*. For aggregate operators, we use a sliding window or tumble window, which is organized as a data structure queue.

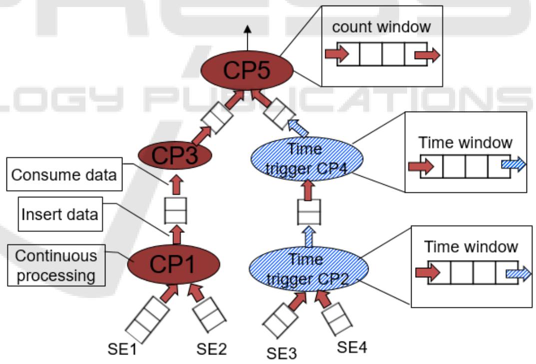

Figure 12: Multi-thread for maintaining data in different kinds of queues.

Now the question is how to maintain the queue inside a stateful operation? For a clear explanation, we focus on the queue of a sliding window or tumble window. A window is called *time window* if its length is based on time, while a window is called *count window* if its length is based on event count. In Figure 12, the arrow that is before a queue indicates inserting data to the queue, while the arrow after a queue indicates consuming data from the queue. The components in red color are stateless or contain a *count window*. They are processed by the thread of round-robin scheduler, which is mentioned above in this section. The components in blue color are stateful and contain time window. They are processed by a new thread, which is responsible for time trigger, called time trigger scheduler. The time trigger scheduler schedules the re-processing time for CPs, processes it when the time up for each and store results to the output queues.

In our system, the output results of CP upstream have high possibility to be used by multiple CPs downstream. There is an example shown in Figure 13. If there are many CPs consuming input data from the same queue, it will have a performance problem to check conditions of each CP downstream one by one. We notice that the case shown in Figure 13 is the same as the one shown in Figure 8. For this case, we use CQ index to solve the problem, which is to find out matching CPs directly by using an index, rather than checking query conditions one by one.

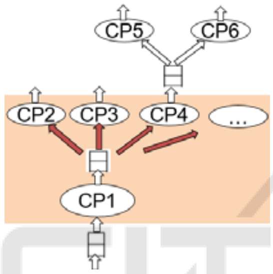

Figure 13: The case to use CQ index.

In section 4, we mentioned that a CQ condition includes different data types cannot be added to an R*-tree directly. Our strategy to solve this problem is to transform the data type of each predicate to be the same. Also, we transform equation predicates to interval predicates. For example, an equation predicate "*id=3*" can be expressed as "$3 \leq id \leq 3$". Here is a complete example to transform predicates "*IFF='enemy' AND 30<speed<100 AND elevation < 10*" to fit the R*-tree. Firstly, we uniform their data type to be *Integer* by using "*std::hash<std::string>*" in C++ to calculate the hash value of string "*enemy*", assuming its hash value equals to number 1389. Secondly, we transform all equation predicates to interval predicates. Thirdly, make "*elevation < 10*" to be "*MIN < elevation < 10*" (*MIN* denotes the minimum integer number). So finally we get the result that is "*1389≤IFF≤1389 AND 30<speed<100 AND MIN < elevation < 10*", which can be added to an R*-tree.

An incoming event tuple can be expressed as {"*id=3, IFF=enemy, speed=50, elevation=9*"} for example. It indicates a point in the R*-tree. We can query all regions in the R*-tree that contain the point by using the function "*void intersectsWithQuery(const IShape& query, IVisitor& v)*" provided by open-source **libspatialindex** (libspatialindex, 2019) project.

Our prototype system provides GUI for users to register rule specifications. Users input a rule specification and click the button "*add*" to finish the registration (Figure 14). Our system will create a graph of processing flow based on input and output stream names of each rule specification. Users can display all or search the graph by an output stream name (Figure 15).

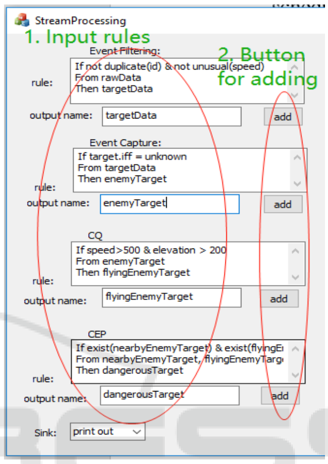

Figure 14: GUI for adding rule specifications.

Our system can insert, search, display, update and delete rule specifications dynamically. In Figure 15, we show the GUI components for these functionalities. By inputting the name of an output stream, users can search or delete a specified rule specification. The system starts to work after clicking the button "start processing". Figure 16 shows the way to display processing results.

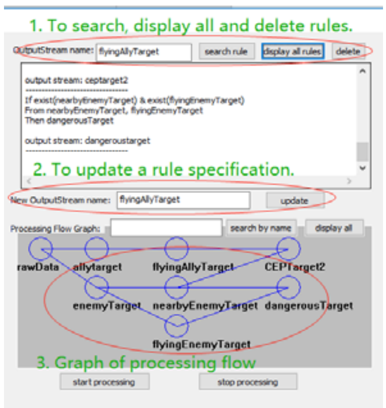

Figure 15: GUI to search, display, delete and update rule specifications, and to display processing flow graph.

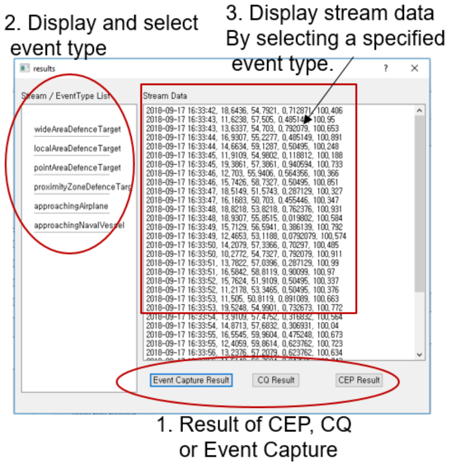

Figure 16: GUI for display stream data.

6 PERFORMANCE

To evaluate the system performance compared with Esper (EsperTech, 2019), we set up 50 rule specifications and calculate processing time with different numbers of input data (Figure 17). To evaluate the effect of using CQ index, we set up 50 CEP rules and calculate the processing time with and without using CQ index (Figure 18) and calculate the time after processing 10,000 events with and without CQ index (Figure 19).

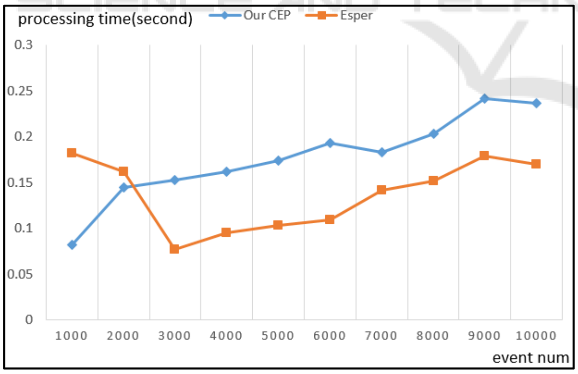

Figure 17: Set up 50 CEP rules, calculate processing time.

We generate event tuples as input stream data with randomly assigned values. Attributes of a tuple are *id*, *time*, *speed*, *elevation*, *IFF*, *longitude*, *latitude* and so on. The evaluation result shown in Figure 17 indicates that the performance of our system is slower than Esper but not different too much. The evaluation results in Figure 18 and 19 indicate that the system has a higher performance by applying CQ index.

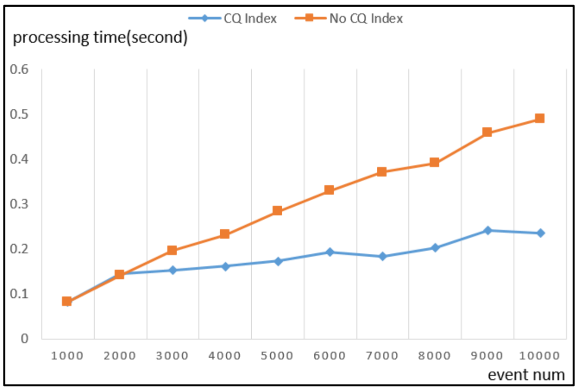

Figure 18: Processing time for 50 CEP rules.

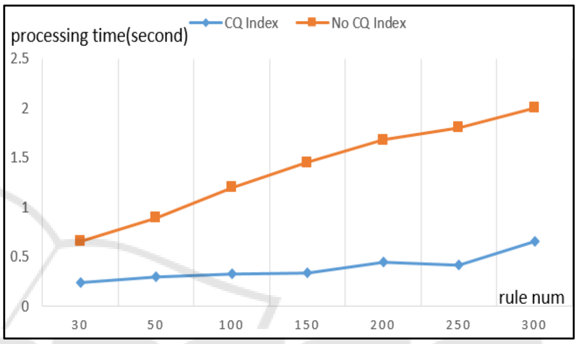

Figure 19: Processing time for 10000 events.

7 RELATED WORK

CEP Language: Much research has been carried out on CEP language and several languages of CEP have been proposed. The paper (Sadri et al., 2004) proposed a language SQL-TS, which is an extension of SQL to express complex sequential patterns in a database. Paper (Demers et al., 2007) presents query language Cayuga based on Cayuga Algebra for naturally expressing complex event patterns. Papers (Agrawal et al., 2008), (Wu et al., 2006) present the language SASE and use NFA-based technology to implement high-performance complex event processing over streams. Also, CEDR (Barga et al., 2006) presents the language for temporal stream modeling. Those languages have common components. They support Sequencing, Kleene closure, Negation, Value predicates, Windowing, Return and so on. The languages could be implemented with high performance by using NFA-based technology. The event selection strategy is Strict or partition contiguity, Skip till next match and Skip till any match.

Optimizing CEP Performance: Paper (Mozafari et al., 2012) proposes a high-performance approach that supports CEP on XML streams. It uses XSeq language to extend XPath with natural operators over XML streams. The papers (Agrawal et al., 2008), (Wu et al., 2006) use NFA-based technology to improve the performance of pattern matching over streams. Papers (Krishnamurthy et al., 2006), (Yang et al., 2009) try to improve CEP performance by making use of sharing among similar queries. (Johnson et al., 2007) Uses out of order stream data by maintaining a small state and without complete stream reconstruction to improve the efficiency of regular expression matching on streams. Paper (Schultz et al., 2009) rewrites event patterns in a more efficient form before translating them into event automata. The work (Akdere et al., 2008) uses plan-based techniques to minimize event transmission costs and can efficiently perform CEP across distributed event sources.

8 CONCLUSIONS

In this paper, we propose a layered architecture to decompose a complex event query into four parts, corresponding four modules of the system. By doing that, we make the responsibilities of each module clearer and simply the rule definitions. Besides, it helps to insert, delete, search rules dynamically. For each module, we make rule definitions and describe their format in detail. This paper shows that it is possible to process various input rules for continuous processing dynamically in layered specifications. We use R*-tree as a multi-dimension index to speed up continuous queries.

ACKNOWLEDGEMENTS

This research is supported by the Agency for Defense Development (UD160012DD).

REFERENCES

Sadri, R., Zaniolo, C., Zarkesh, A., & Adibi, J. 2004. Expressing and optimizing sequence queries in database systems. *ACM Transactions on Database Systems (TODS)*, 29(2), 282-318.

Barga, Roger S., et al. "Consistent streaming through time: A vision for event stream processing." *arXiv preprint* cs/0612115. 2006.

Demers, Alan J., et al. 2007. "Cayuga: A General Purpose Event Monitoring System." *Cidr. Vol. 7.*

Agrawal, Jagrati, et al. 2008. "Efficient pattern matching over event streams." *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. ACM.

Wu, Eugene, Yanlei Diao, and Shariq Rizvi. 2006. "High-performance complex event processing over streams." *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. ACM.

Mozafari, et al. 2012. "High-performance complex event processing over XML streams." *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM.

Krishnamurthy, Sailesh, Chung Wu, and Michael Franklin. 2006. "On-the-fly sharing for streamed aggregation." *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. ACM.

Yang, D., Rundensteiner, E.A. and Ward, M.O. 2009. A shared execution strategy for multiple pattern mining requests over streaming data. *Proceedings of the VLDB Endowment*, 2(1), pp.874-885.

Johnson, Theodore, S. Muthukrishnan, and Irina Rozenbaum. 2007. "Monitoring regular expressions on out-of-order streams." Data Engineering, 2007. *ICDE 2007. IEEE 23rd International Conference on. IEEE.*

Schultz-Møller, Nicholas Poul, Matteo Migliavacca, and Peter Pietzuch. 2009. "Distributed complex event processing with query rewriting." *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*. ACM.

Akdere, Mert, Uğur Çetintemel, and Nesime Tatbul. 2008. "Plan-based complex event detection across distributed sources." *Proceedings of the VLDB Endowment 1.1*: 66-77.

EsperTech, Espter. 2019. http://www.espertech.com/esper/

libspatialindex, 2019. https://libspatialindex.org/index.html