Research Journal of Applied Sciences, Engineering and Technology 8(13): 1531-1543, 2014
DOI:10.19026/rjaset.8.1131
ISSN: 2040-7459; e-ISSN: 2040-7467
© 2014 Maxwell Scientific Publication Corp.
Submitted: June 03, 2014 Accepted: July 13, 2014 Published: October 05, 2014

**Research Article**

# Optimizing Expressibility and Performance of Kleene Operators on Binary Tree-based Directed Graph in Complex Event Processing

Babak Behravesh, Siti Mariyam Shamsuddin, Alex Hiang Tze Sim and Hassan Chizari
Faculty of Computing, Universiti Teknologi Malaysia, Malaysia

**Abstract:** This study presents a novel method to develop kleene operators on a Binary Tree-Based Directed Graph (BTDG) to improve expressibility and performance of our developed complex event processing system. Complex Event Processing (CEP) systems are widely employed to notify opportunities and threats, which many of these situations are discovered through detection of multiple occurrences of the similar set of events. Many of CEP systems proposed kleene operators to detect these situations. The high expressibility of our implemented CEP system achieved by developing algorithms on kleene operator. These algorithms are designed on a BTDG as a unified basis to arrange input data and joining this input according to a given query. The performance is improved through directing events instantly to join with other events, dynamically switching to an optimal plan and late applying combination for kleene with specified size. The experiments shows kleene operator on BTDG brings rather high degree of expressiblity and performance in compare to its counterparts.

**Keywords:** Binary tree-based directed graph, complex event processing, event matching, kleene operator

## INTRODUCTION

In event processing systems, events are received from several peripherals; the events are investigated to discover match cases against user-defined queries and the results are either notification messages or automated actions. The user-defined queries determine the relationship between events with respect to logical and temporal constraints and the queries containing kleene closure, in particular, search for multiple occurrences of events. A large body of applications in many areas employs CEP systems to monitor iterative occurrences of events in supply chain management systems to count number of pallets that read by RFID antenna (Gyllstrom *et al*., 2007), in health care services to track health-care workers for hygiene compliance (Wang *et al*., 2010) and in stock market analysis to monitor multiple changes in stock price (Mei and Madden, 2009; Peer *et al*., 2013).

Several CEP systems developed in academia and enterprise used either finite state machine or tree structure as a foundation to perform event matching. In FSM (Wu *et al*., 2006; Agrawal *et al*., 2008; Cugola and Margara, 2012) and Tree-based (Mei and Madden, 2009) systems event processing systems kleene is defined for three possible cases; kleene *, denotes that the event can have no occurrence or can occur more than 0. kleene +, implies event can occur once or more times; kleene num, detects situations that event occurs in a specified times. In FSM systems a great deal of discussions dedicated to kleene operator and ignoring some events to cover this operator (Agrawal *et al*., 2008; Muthusamy *et al*., 2010; Demers *et al*., 2007). These systems use Non-deterministic Finite Automaton (NFA) as a basis to distinguish the relevant from irrelevant events with respect to the query. They added an option in their query definition that determines the level of relaxation for event selection (Agrawal *et al*., 2008). One the other hand, in tree-based systems, namely ZSTREAM (Mei and Madden, 2009), they use a ternary of buffers to confine number of composite events. In Mei and Madden (2009), they argued on shortcoming in NFA-based models to support concurrency.

According to some of previous work, improving expressibility of query languages to support wide range of queries is considered as an important issue (Diao *et al*., 2008; Cugola and Margara, 2012). In SASE (Wu *et al*., 2006), a CEP system developed to support sequence and negation operator as well as some strategies to set level of relaxation on relevant events; however, SASE does not support kleene operator. In SASE+ (Agrawal *et al*., 2008), they addressed the need to improve expressibility of SASE by developing an algorithm to support kleene operator. They developed match buffer as a shared buffer for multiple processes. However, NFA-based systems have the following shortcomings:

- These systems are limited to perform matching from left to right (Cugola and Margara, 2012), or right to left (Diao *et al*., 2008; Agrawal *et al*.,

**Corresponding Author:** Babak Behravesh, Faculty of Computing, Universiti Teknologi Malaysia, Malaysia

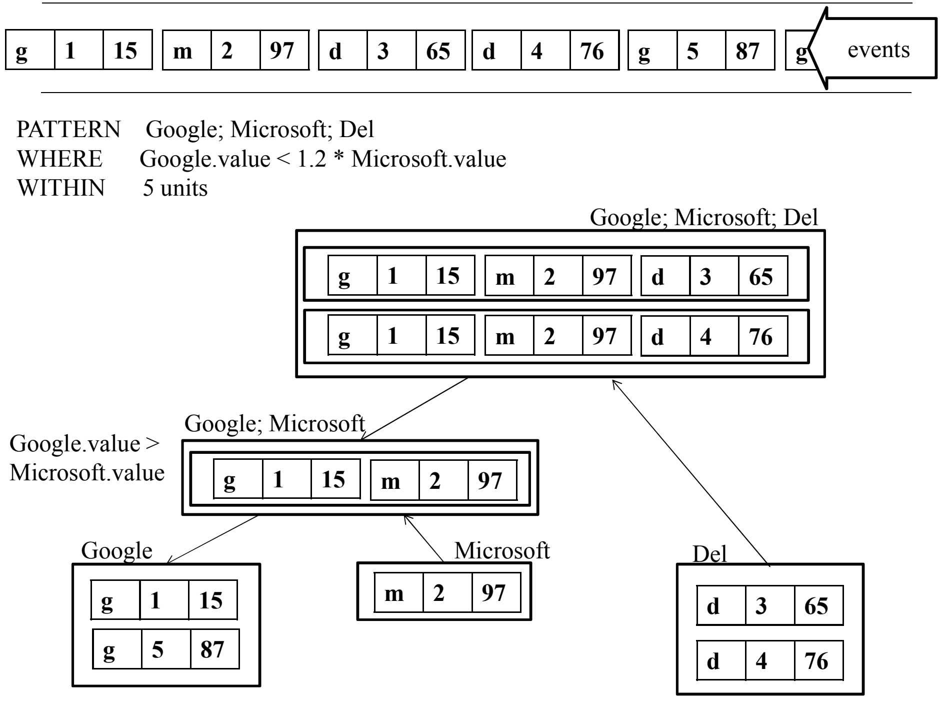

g 1 15 m 2 97 d 3 65 d 4 76 g 5 87 g
events
PATTERN Google; Microsoft; Del
WHERE Google.value < 1.2 * Microsoft.value
WITHIN 5 units
Google; Microsoft; Del
Google; Microsoft
Google.value > Microsoft.value
Google
Microsoft
Del

Fig. 1: Binaty tree-based graph in FAEM

2008), which makes these systems limited to reorder join operators.

- We argue and show in this study that their proposed shared buffer, cannot guaranty the validity of results.

In contrast to NFA-based models, ZSTREAM (Mei and Madden, 2009) manage events in tree-based structure. In supporting kleene, they developed a ternary including three event classes that are joined through an edge into the parent buffer which keeps the result of applying kleene operator. In their system, the kleene event class is surrounded by two other non-kleene event classes. ZSTREAM uses right and left buffers to erase irrelevant events from kleene buffer which is only considered in middle buffer. Their technique reduces memory space required for applying composition and reduces processing time.

However, kleene operator in ZSTREAM imposes some limitations as:

- Connecting three buffers as a ternary, with a kleene buffer in the middle, between left and right buffers, negatively affects the expresibility of the kleene operator because user may define some queries that violates this limitation.
- The sequence operators to join in right and left side of kleene buffer are compulsory while other operators may occur instead.
- Some of events out of the ternary may discard the events in range of ternary; this imposes an extra burden in creating and then removing invalid events. For the first time, we show these shortcomings in expressibility section.

Figure 1 shows the arrangement of buffers and cursors in FAEM, the system that we previously developed (Behravesh *et al*., 2014). Given the stream of events and a user-defined query, FAEM generates a tree-based directed graph. Event classes as appeared in *PATTERN* section of the query forms single class events and their composition creates multi-class events. The predicates Google.value <1.2 * Microsoft.value is determined based on Where clause and assigned to the lowest class as possible. Each event contains ID, timestamp and value. Buffers are hashed by the ID based on pattern section of the query. In event stream, events are placed in their respective buffers and after satisfying evaluation; their composition is placed in their parent buffer. The constraints are logical and temporal, the logical constraints are define in WHERE clause and join operator here is ";" implying sequence operator. Temporal constraints are either sliding time-window constraint which is determined by WITHIN, or order constraint that here defines it is expected to find Microsoft events following Google events. The aim of this study to improve expressibility and performance of pattern matching in CEP systems through optimization on various kleene operators. In order to achieve this goal, following objective are satisfied:

- Designing kleene operators' algorithms on binary tree-based directed graph to improve expressibility of applying kleene on concurrent events.

Moreover, the directed graph helps the system to the perform matching as soon as receiving an event.

- Developing an optimized kleene num algorithm to improve performance of event matching through not to generating excessive intermediate results. In addition, the algorithm avoids unbiased removal of valid composite events.

However, the shortcoming of these systems is described in details in results section.

## METHODOLOGY

One of the inputs of a CEP system is a user-defined query which finds its match cases over stream of input events. The queries in below are related to stock market that are define by an expert user. In dealing with endless stream of stock market events, these queries are some of the sample queries to discover their match cases:

```
Query 1:
PATTERN Google; Microsoft+
WHERE Google.value >100$
AND Google.value <Microsoft
WITHIN 5 UNIT
Query 2:
PATTERN Google; Microsoft [3] and Del
WHERE Google.value>1.12 * Mircosoft.value
AND Google.value >1.2 * Del.value
WITHIN 10 SEC
Query 3:
PATTERN Google; Microsoft*; Del; Intel
WHERE Intel.value>Mircosoft.value
WITHIN 200 UNIT
```

Query 1 searches for as many as Google events within the range of time window, 5 events, each of these events worth more than 100$ that these are followed by a Microsoft event which worth more than all of these Google events. The symbol+implies that 1 or more Google events. Query 2, in a floating time window of 10 sec, discovers Google events that followed by three Microsoft events which their prices have at least 12% less values than Google these are followed by a Del event which its price is at least 20% less than Google's. Query 3, searches input stream for match cases when Google event is followed by as many as Microsoft events, 1 Del event and 1 Intel event within 200 events, so that the value of Intel stock event is more than Microsoft's. The symbol * implies 0 or more Microsoft events.

**Kleene operator in FAEM:** In this study, kleene operator for all sorts of kleene (*/+/ num) follows an identical structure based on BTDG. However, for kleene num, an extra step is applied to generate all sets in the final step. Likewise other buffers, events in kleene buffer collect all of their respective events in its buffer with respect to requirements of time-window and WHERE clause. The only difference is as they join with other operators, a group of events, more than one instance of kleene events are selected from kleene buffer.

**Kleene star and plus:** When a new event is received, the system pushes the event into its related buffer. If its pair buffer based on query is found a kleene, grouping is applied on kleene events with respect to temporal and logical constraints and composite event is pushed into their parent buffer; similarly for each newly received kleene event, grouping is applied based on constraints and the event instances in its adjacent buffer which is a non-kleene buffer. When the left buffer is a kleene buffer and associative operator is between right and left buffer, the algorithm below generates composite events in MBuf.

**Algorithm 1:** Kleene on Left
**Input:** LBuf, RBuf, MBuf are left, right and result event buffers, respectively;
qry is query
**Output:** MBuf result buffer ◁ left buffer is kleene and right buffer is a non-kleene

```
1: initialize LGroupTemp
2: for Rr = RBuf [end]; Rr = RBuf [init]; ++Rr do
3:     if Rr.start ts<Now-time window then
4:         remove RBuf [init] to Rr; continue;
5:     for Lr = LBuf [0]; Lr = LBuf [end]; ++Lr do
6:         if Lr.start ts<Now-time window then
7:             remove LBuf [0] to Lr; continue;
8:         if Have Constraints Satis_ed (Lr, Rr) &&
           Lr.end ts<Rr.start ts then
9:             insert Lr into LGroupTemp
10:    if (LGroupTemp is empty) AND
       (LBuf.KleeneType == '*') then
11:        insert Rr into MBuf
12:    else if (LGroupTemp is not empty) then
13:        insert (LGroupTemp+Rr) into MBuf; clear
               LGroupTemp
14: RBuf [initial] = RBuf [end]; ◁ setting initial location for next round
```

Provided a pair of engaged-in-join event classes, if the left event class is a kleene then algorithm 1 is applied. The input are LBuf and RBuf which are the left and right buffers, representing the left and right engaged-in-join event classes. MBuf is the parent buffer which keeps results of join between events in LBuf and MBuf. qry is the user-defined continuous query. The output is set of composite events stored in MBuf that keeps the result of applying a binary operator between events in LBuf and RBuf. LGroupTemp is a temporary composite event, which is populated incrementally and inserted in MBuf upon completion. In lines (2-4), the algorithm checks all events in right buffer from the end to the beginning and checks if the events are in range of time-window. The out-of range events are subjected to deletion. Similarly, in lines (5-7) events in left buffer are checked. In each round, if the left and right events satisfy logical and time order constraints, only the left event is inserted into LGroupTemp, lines (8-9). So, at the end of the inner loop, LGroupTemp is populated by all events in LBuf which are matched to Rr. In line (10-13), the function checks if the LGroupTemp is empty, since only kleene start is allowed to have a composite event entered when there is no event match in kleene section. Otherwise, Rr is added at the end of LGroupTemp and then LGroupTemp is inserted into MBuf. LGroupTemp is cleared for the next round. In order to not generate duplicate composite events we need to start from the last position of a non-kleene buffer (line 15).

Algorithm 2, applies kleene on pair of buffers where the left and right buffers represent a non-kleene and a kleene event classes, respectively. The input and output parameters of algorithm 2 is the same as algorithm 1. The two variables defined in lines 1 and 2 are boolean and are set to false by default. The first variable determines if the specified event in kleene section is matched with at least one event in the left buffer. The second variable determines if the MBuf contains an event starting by a specific event in left buffer. For all events in right buffer the algorithm checks if its start time is within the range of time window and the out-of-range events are removed from the buffer (lines 3-5). The validity in range of time-window is checked for events in left buffer (lines 7-9), which is a nested loop in the right buffer loop. For all events in left buffer, the algorithm checks logical constraints and order constraint, if at least one of the events in left buffer, Lr is satis_ed (10-11). The third nested loop checks for in range-of-time window event, Mr, in parent buffer, MBuf, that starts with Lr and inserts Rr at the end of Mr (12-17). At the end of searching the MBuf (lines 18-19), if Lr is not found at the beginning of any events in MBuf, the composition of Lr+Rr is inserted into MBuf.

**Algorithm 2:** Kleene on Right
**Input:** LBuf, RBuf, MBuf are left, right and result event buffers, respectively
qry is query
**Output:** MBuf result buffer ◃ right buffer is kleene and left buffer is a non-kleene

```
1: initialize is_R_satisfied = false ◃ if a right event is satisfied
2: initialize is_in_MBuf = false ◃ if event is in MBuf or is new
3: for Rr = RBuf [0]; Rr = RBuf [end]; ++ Rr do
4:     if Rr.start ts<Now-time window then
5:         remove Rr; continue
6:     is_R_satisfied = false; is in MBuf = false;
7:     for Lr = LBuf [0]; Lr = LBuf [end]; ++ Lr do
8:         if Lr.start ts<Now-time window then
9:             remove Lr; continue
10:        if Have_Constraints_Satisfied (Lr, Rr) &&
           Lr.end ts<Rr.start ts then
11:            is_in_MBuf = false; is_R_satisfied = true
12:            for Mr = MBuf [0]; Mr = MBuf [end]; +
               +Mr do
13:                if Mr.start ts<Now-time window then
14:                    remove Mr; continue
15:                if Mr starts with Lr then
16:                    Mr = Mr+Rr
17:                    is_in_MBuf = true;
18:            if is_in_MBuf == false then
19:                insert (Lr+Rr) into MBuf
20:        if (is_R_satisfied == false) AND
           (RBuf).KleeneType == '*' then
21:            insert Lr into MBuf
```

If the operator is a kleene*, only satisfaction of non-kleene event is enough for insertion into the parent buffer and Lr is inserted into MBuf (lines 20- 21). It is obvious that the time complexity of Algorithm 1 is better than the algorithm 2. So, in plan adaptation algorithm that we previously addressed in FAEM, Algorithm 1 is more beneficial than algorithm 2, regardless of other factors affecting the optimal plan.

**Kleene on concurrent join:** Another benefit of applying binary tree-based directed graph is to perform matching on kleene events when the join operator is either conjunction or disjunction. Conjunction supports concurrency between pair pf event classes. According to the definition of conjunction (Mei and Madden, 2009), events from *A* or *B* classes only needs to be in the range of time-window without any constraint on their order of occurrence. In FAEM, we developed this definition on BTDG to support conjunction not only between two non-kleene event classes, but between a kleene and a non-kleene event class. Figure 2 shows how (*A*+& *B*) can be represented by (*A*+;*B*) ∪ (*B*;*A*+).

For every new event in the buffer A, grouping in buffer A is performed with respect to each of B events in algorithm 2 and for a new event instance of B grouping is performed on buffer A Algorithm 1. Similarly, (*A** &*B*) and (*A* [*num*]&*B*) are represented by (*A**;*B*) ∪ (*B*;*A**) and (*A* [*num*]; *B*) ∪ (*B*;*A* [*num*]), respectively.

Given Query 1, Fig. 3 shows how matching performs on receiving events. The system receives m1 and stores it in buffer *M*; the system then checks for a match case in *G* buffer. When it checks content of buffer *G* through previous cursor, it finds no event in *G*. Thus, no matching is performed. Another event instance, m2, in *M* is received from input stream, but buffer *G* is still empty, so no evaluation is performed. A new event instance, *g*3, is received in buffer *G* arrives

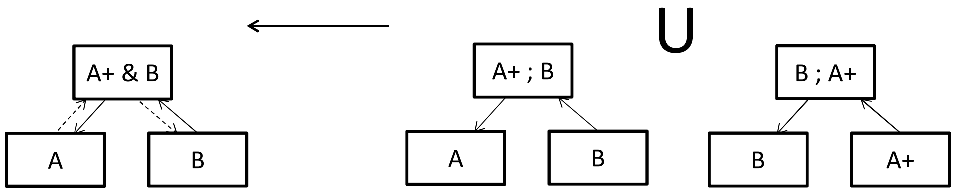


Fig. 2: Conjunction kleene in FAEM

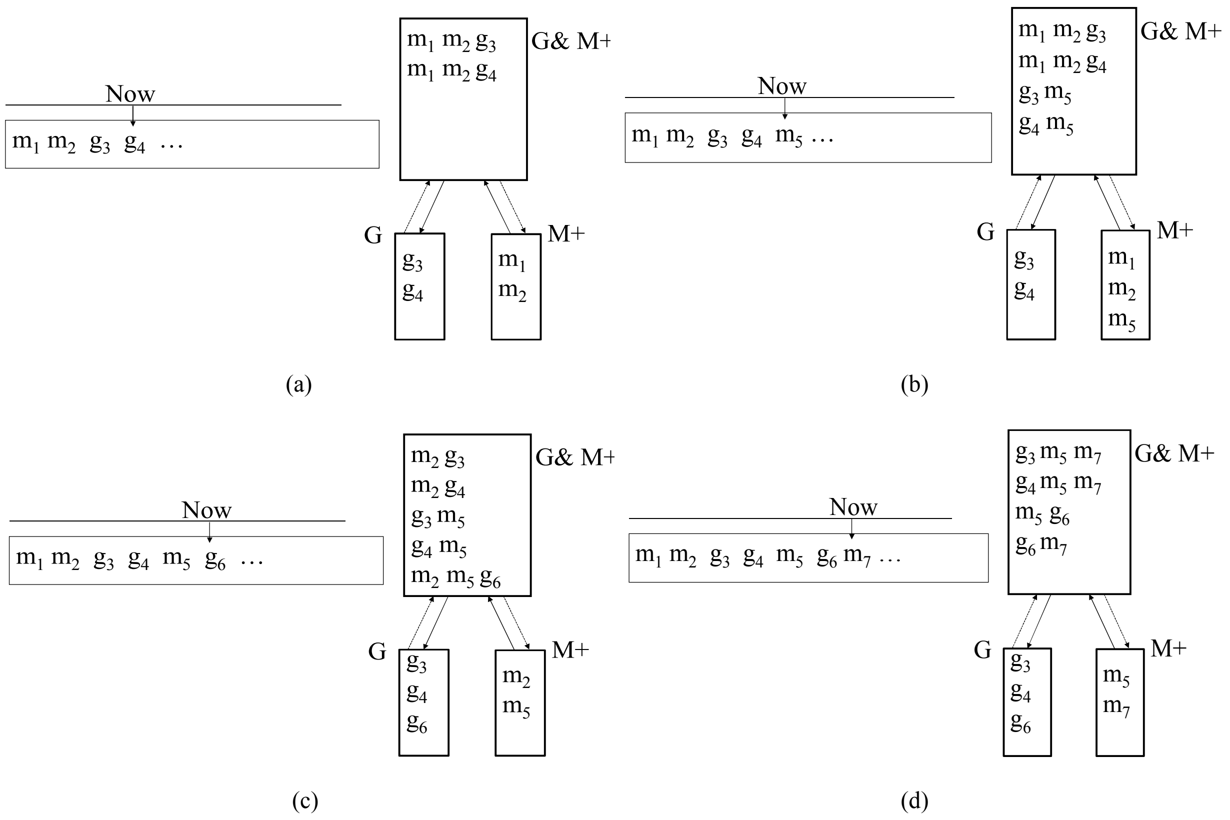


Fig. 3: Throughput (a) number of match cases, (b) memory consumption (c) and intermediate results, (d) on kleene plus in FAEM

and checks the buffer *M* through next cursor. As *g*3 is received the algorithm checks buffer *M* from beginning to find those *M* instances matched to *g*3, as a group and at the end *g*3 is added to the end of group generating *m*1 *m*2 *g*3. The composite event is inserted into the parent buffer. The same scenario happens when *g*4 is received and makes composite events with *m*1 and *m*2 with respect to constraints. When event *m*5 is received, algorithm performs grouping on *M* buffer with respect to all events in buffer *G*. In buffer *M*, the algorithm creates a group by combining *M* events instances as many matched with *g*3. The first event pushes in temporary composite event is *g*3 followed by *m*5. Here, in buffer *M*, *m*5 is the only event match with *g*3; so, *g*3*m*5 is inserted into the parent buffer. Similarly, *g*4*m*5 is inserted into the parent buffer.

After receiving *g*6, the right buffer is checked to find match cases. In *M* buffer, *m*1 is found invalid as it cannot satisfy temporal constraint defined in time-window and is removed from *M* buffer. *m*2 and *m*5 are matched with *g*6 and their composition is inserted into *G&M*+ buffer. As *m*7 is received, the algorithm checks *G* buffer to find match cases. *m*2 is removed because it is out of time window. *m*7 is found match to *g*3 and algorithm checks composite events in *G&M*+ to find any event starting with *g*3 and adds *m*7 to the end of the composite event making *g*3*m*5*m*7. Similarly *g*4*m*5*m*7 is generated. In comparison between *m*7 and the last event in *G* buffer, *g*6 is found match to *m*7 but there is no event in *G&M*+ starting with *g*6; so, *g*6*m*7 is inserted into *G&M*+.

For a pattern section of a query as (*A*+; *B*) and (*A*; *B*+), the algorithm 1 and 2 can be applied respectively. For the conjunction between A and B when either is a kleene event, union of algorithm 1 and 2 generates results.

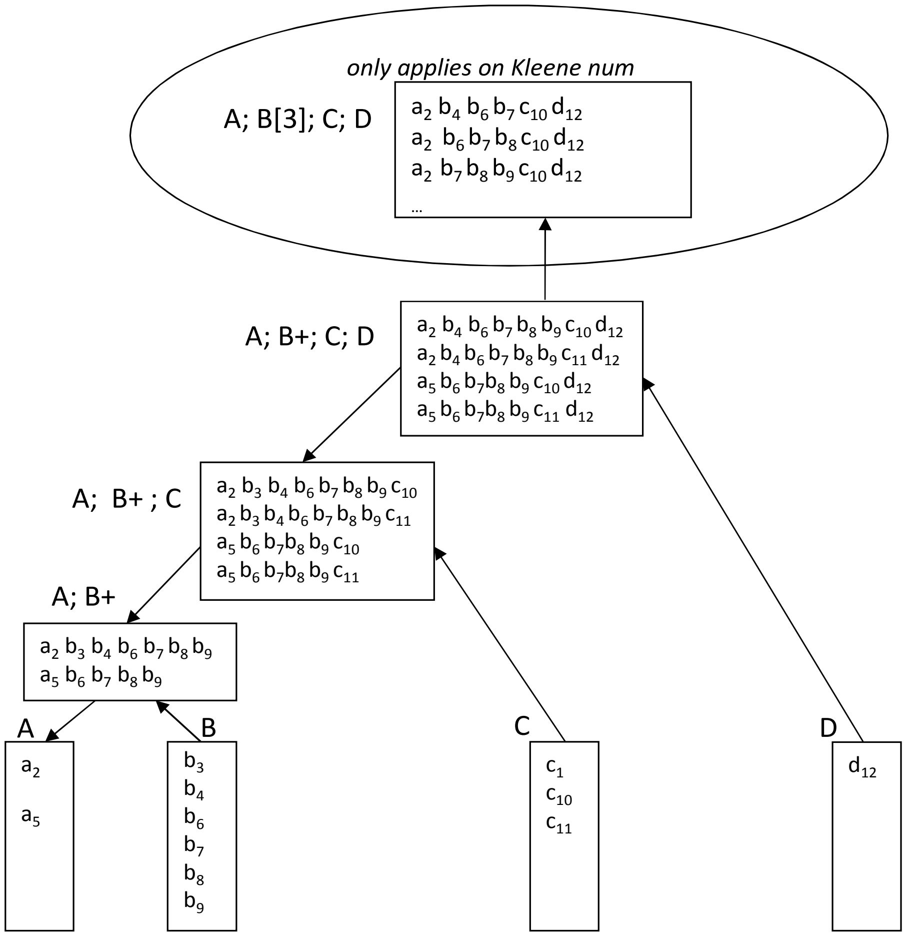

only applies on Kleene num
A; B[3]; C; D
$a_2\ b_4\ b_6\ b_7\ c_{10}\ d_{12}$
$a_2\ b_6\ b_7\ b_8\ c_{10}\ d_{12}$
$a_2\ b_7\ b_8\ b_9\ c_{10}\ d_{12}$
…
A; B+; C; D
$a_2\ b_4\ b_6\ b_7\ b_8\ b_9\ c_{10}\ d_{12}$
$a_2\ b_4\ b_6\ b_7\ b_8\ b_9\ c_{11}\ d_{12}$
$a_5\ b_6\ b_7\ b_8\ b_9\ c_{10}\ d_{12}$
$a_5\ b_6\ b_7\ b_8\ b_9\ c_{11}\ d_{12}$
A; B+ ; C
$a_2\ b_3\ b_4\ b_6\ b_7\ b_8\ b_9\ c_{10}$
$a_2\ b_3\ b_4\ b_6\ b_7\ b_8\ b_9\ c_{11}$
$a_5\ b_6\ b_7\ b_8\ b_9\ c_{10}$
$a_5\ b_6\ b_7\ b_8\ b_9\ c_{11}$
A; B+
$a_2\ b_3\ b_4\ b_6\ b_7\ b_8\ b_9$
$a_5\ b_6\ b_7\ b_8\ b_9$
A
$a_2$
$a_5$
B
$b_3$
$b_4$
$b_6$
$b_7$
$b_8$
$b_9$
C
$c_1$
$c_{10}$
$c_{11}$
D
$d_{12}$

Fig. 4: Kleene num in FAEM

**Kleene num:** Likewise, kleene+ and kleene*, in kleene num the algorithm groups as many events in kleene buffer as match with temporal and logical constraints. In higher levels before applying join operator, the number of kleene events in a composite event is checked if it fulfills the num as well as temporal and logical constraints. The checking in higher levels is because some of kleene events may be discarded from composite events if they do not satisfy constraints in next evaluations. Removing these event instances from kleene section is because number of valid events in kleene is less than num for some of composite events. In final stage some of composite events may have more kleene event instances so various composite events are generated so that with kleene length equal to num.

Given Query 3, Fig. 4 shows matching in buffers where Google, Microsoft, Del and Intel are represented by *A*, *B*, *C*, *D*. In this figure, we start the description after receiving $d_{12}$ because earlier steps are straight forward; however, later steps refer to these steps. When $d_{12}$ is received, it invalidates for instance one of the kleene events, namely *B*. In FAEM, only that event, $b_3$ is subjected to removal from the respective composite event.

The kleene operator in FAEM is applied after applying each operator in higher levels of hierarchy to check if number of kleene items is met. If number of items are less than kleene number the composite event is removed; otherwise, the proposed combination method in below is applied to generate all possible composite events when the process arrives to the final state.

**Proposition:** The events are arrived to the system based on their timestamps, or out of order events are not considered. In order to present steps clearly, we first show the function of the system through an example, then a formal definition following the algorithm for FAEM combination are presented. Given set of events placed in their related buffer based on hashing on their IDs and a continuous query as below, we apply FAEM combination.

**Combination algorithm:** Kleene num generates some or all of composite events as result of applying a partial combination (Mei and Madden, 2009) or a complete combination algorithm. During process kleene num is considered as kleene plus. FAEM applies combination algorithm on top of full match cases. FAEM considers maximum number of matched events. The combination algorithm in FAEM functions in three steps: The algorithm checks the previous results of combination on kleene events, $e_0$ to $e_{i-1}$, to use for generating new combinations for $e_i$. Generating all combinations using conventional combination algorithm for n-2 kleene events, $e_1$ .. $e_{n-2}$, out n events. Then, adding the $e_{n-1}$ and $e_n$ to the end of event of each composite event. Here, $a_2b_4b_6b_7b_8b_9c_{10}d_{12}$ is the first events in down buffer with 5 kleene events, $b_4b_6b_7b_8b_9$:

- To make combination with length of 3 as defined in pattern section of the query, *A*; *B* (Cugola and Margara, 2012); *C*; *D*, according to FAEM Combination algorithm, the algorithm selects the first three events in kleene, $b_4b_6b_7$. Then, the events related to other classes are placed before and after kleene section, $a_2$ and $c_{10}d_{12}$ to generate the first results of combination. So, it generates $a_2b_4b_6b_7c_{10}d_{12}$ and adds this composite event to the final buffer. Time complexity is *O* (*1*).
- Then, the algorithm makes a copy of newly generated composite event and replaces the last event in kleene section, $b_7$, with next kleene event which has not been considered yet, $b_8$, which produces $a_2b_4b_6b_8c_{10}d_{12}$ and adds this composite event to the final buffer.

**Algorithm 3:** Kleene Num
**Input:** *n*, *r*, r events from n valid events as in combination, finalBuf
**Output:** result buffer finalBuf

```
1: ce = deque from downBuf
2: cePrefix = copy prefix section of ce, not including
   the kleene section
3: cePostfix = copy postfix section of ce, not including
   the kleene section
4: ceKleene = copy Kleene section of ce
5: if Size of ceKleene<r then
6:     break;
7: else if size of ceKleene = r then
8:     ceNewItem = cePrefix+ceKleene+cePostfix
9:     insert ceNewItem into FirstSetBuff ◁ should be
       sorted
10: else if size of ceKleene>r then
11:     for i = 1; i< = r; i++do
12:         ceTemp ceTemp+ceKleene [i]
13:     ceNewItem = cePrefix+ceTemp+cePostfix
14:     insert ceNewItem into FirstSetBuf
15:     i = r
16:     while i<n do
17:         i = i+1
18:         j = sizeof FirstSetBuff;
19:         while j> = 1 do
20:             ceTemp = kleene section of
                FirstSetBuff (j)
21:             if last event of ceTemp = ceKleene [i-1]
                then
22:                 ceTemp = replace the last event of
                    ceTemp with ceKleene [i]
23:                 ceNewItem = cePrefix+ceTemp+
                    cePostfix
24:                 insert ceNewItem into FirstSetBuff
25:             else
26:                 break;
27:             j- -;
28:         ceTempBuffer = Combination (n-2, r-2,
            ceTemp)
29:         for all composite event ceTB in
            ceTempBuffer do
30:             ceTemp = ceTB+ceTemp [i-1]+
                ceTemp [i]
31:             ceNewItem = cePrefix+ceTemp+
                cePostfix
32:             insert ceNewItem into SecondSetBuff
33:   return FinalBuffer
```

- Then the algorithm takes the previously final event in kleene, $b_7$ and concatenates it to current final event in kleene, $b_8$; keep them as two last events of new composite events for later use. Then, place it as two last final events in kleene section of new events. The beginning events are obtained from combination algorithm where it is applied on *r-2* first events in kleene, $b_4b_6$, apart from $b_7$ and $b_8$ which have been already considered. For all composite events as result of combination of *r-2* first events in kleene, the combination result is attached to $b_7b_8$ and adds the other events $a_2$ and $c_{10}d_{12}$ to make composite events. The new composite events are also added to the final buffer.

Finally MergeJoinSort algorithm is applied twice to sort composite events in buffer in ascending order based on their time stamps:

- At first MergeJoinSort algorithm is applied to sort composite events added in step 2 and 3 produces a sorted set of newly added composite events in final buffer.
- Given the sorted composite events in *i* section which only covers the newly added events and old composite events previously placed in final buffer, MergeSortJoin algorithm applied to sort all composite events in final buffer.

The process is continued with a new event in kleene section, $b_9$ and stops when there is no more kleene event to check. The algorithm 3 presents kleene num in FAEM.

For composite events with non-kleene events if constraint is not satisfied, the system ignores the composite event and proceeds to the next composite event. In order to avoid false negatives for each composite events including kleene events, if the constraint is not satisfied, the system ignores only unmatched kleene event instances and checks if number of kleene events, in kleene events group, still meets the constraints; then, composite event is generated for matching. For example, for kleene num, number of kleene events is expected to be greater equal to num and in case of finding less than num matched the composite event is ignored and proceeds to the next composite event.

## IMPLEMENTATION

In implementation of FAEM, buffers are instantiated at run-time with respect to the pattern section of a given query. FAEM considers one buffer for each event class, specifically: single event class buffers which are in the lowest level of hierarchy in a tree-based architecture and composite event class buffers which are instantiated to preserve composite events as results of applying join operators. Buffer is a collection of events of the same type. The events in each buffer determined based on hashing on its ID and preserves the respective events as long as they are within range of a time window. Single and Composite event buffers are connected to each other in a hierarchical tree-based directed graph. During implementation, a composite event class consists of a vector of simple events arranged by their timestamp; while a single event class consists of event ID, timestamp and values attributes where the latter is a vector of value. Each buffer instance is in join with another buffer through next and previous cursors considering that only the next cursor exists between a pair of buffers with two-way operators in between. For single event class predicates, FAEM places predicates in the buffer of respective simple event buffer; while for multi-class predicates, the predicate is placed in the internal node containing involved event classes in predicates at the lowest possible level.

The dataset that we used in this study is Yahoo! KDD cup dataset related to music tracks downloaded by users. The dataset represents a sampled snapshot of the Yahoo! Music community's preferences for various musical items with different entities including tracks, albums, artists and genres. The number of user ratings is varied from tenths to thousands of ratings (events) a day. FAEM reads from a file containing more than 1,000,000 user ratings at maximum speed and allocates a time-stamp to each event at their respective arrival time.

## RESULTS

In order to evaluate the expressibility and performance of FAEM in supporting kleene operator, the tests performed as follows:

- On expressibility, we compare FAEM with NFA-based systems (Agrawal *et al*., 2008; Cugola and Margara, 2012; Mei and Madden, 2009) as tree-based system to show the limitations of these systems to support range of user-defined queries.
- On performance, we first show the dynamic behavior of tree-based system to switch to efficient plan while NFA-based plans does not support. Then, we compare performance of kleene num in FAEM with ZSTREAM (Cugola and Margara, 2012) which both are tree-based systems.

FAEM is implemented in C++ using STL: vector to support buffers. We implemented the tree-based approach as described in (Mei and Madden, 2009) (in C++ using STL: List to maintain buffers). In our experiments STL: vector performs faster than STL: List when data is small. Conversely, for large amount of data STL: List performs better than STL: Vector.

We performed all experiments on a dual core CPU 2.1 GHz. Intel Pentium 4 and 2 GB RAM DDR2. The system reads data from a pre-recorded data file and data pulled into the system with the maximum rate system accept, where the rate is calculated as the number of events in a second. The time required to read data from files, in addition to the time needed to deliver output is not considered. We run some experiments on Yahoo! KDD Cup2011. In some of the experiments we report peak memory. The average of 40 runs performed on all experiments.

**Expressibility:** At first we present the expressibilty of ZSTREAM on queries presented on methodology section. ZSTREAM presented algorithm to support kleene operator only lets the kleene event class appears between two non-kleene event classes joined as a ternary with sequence operators between them. Thus, ZSTREAM is not able to run Query 1 because the length of pattern is less than 3 and does not support Query 2 because the join between 2nd and 3rd event classes is a conjunction, while ZSTREAM only supports sequence operator to join event classes.

ZSTREAM is expressible to support Query 3; However, some of the useful intermediate results are removed during matching. Figure 5 illustrates event composition in ZSTREAM. The superscript and subscript of each event shows its value and timestamp, respectively. A composite event is generated as result of applying on kleene operator, $g^{12}_{2}$; $m^{16}_{3}$; $m^{12}_{6}$; $m^{13}_{7}$; $d^{18}_{8}$. This composite event in joining with a $i^{11}_{9}$ in buffer *I*, instead of removing the unmatched single events from kleene section of the composite event, removes the entire composite event, as their proposed method presented no mechanism to deal with this situations.

On the other hand, in NFA-based systems namely, SASE+ (Agrawal *et al*., 2008) uses a breadth first

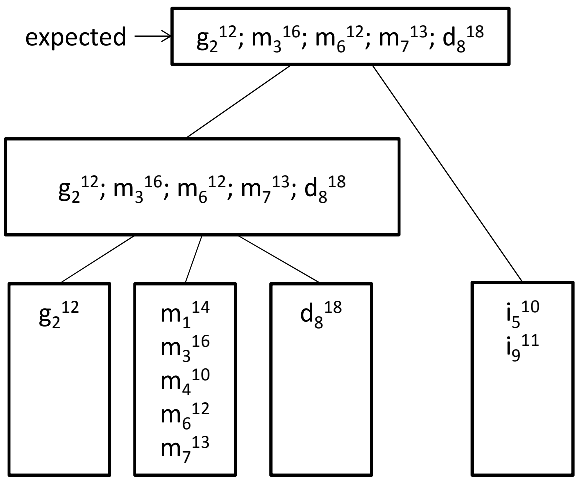

expected
$g_2^{12}; m_3^{16}; m_6^{12}; m_7^{13}; d_8^{18}$
$g_2^{12}; m_3^{16}; m_6^{12}; m_7^{13}; d_8^{18}$
$g_2^{12}$
$m_1^{14}$ $m_3^{16}$ $m_4^{10}$ $m_6^{12}$ $m_7^{13}$
$d_8^{18}$
$i_5^{10}$ $i_9^{11}$

Fig. 5: Shortcoming of kleene+ and * in ZSTREAM

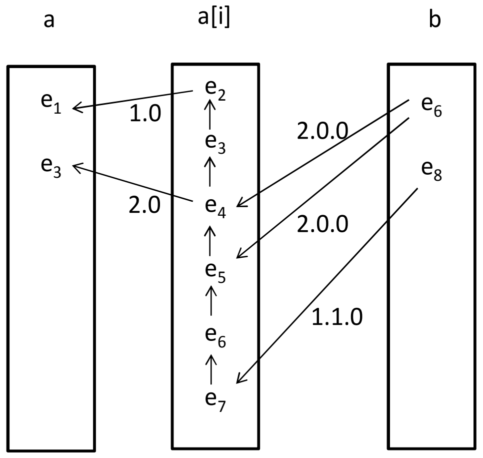


Fig. 6: Shortcoming of SASE in sharing buffer

algorithm to search for all the items in breadth to find match cases. Then search is continued in previous breadths in backward manner. They used a shared buffer to avoid assigning numerous stacks and store events in them. To keep track of each run they assign a version number to connect events. However, we show that in some scenarios, using their share buffer technique would be troublesome. According to their model, events related to different runs can be placed in their respective buffers and pointing to each other as shown in Fig. 6.

When a new event is received a version number id assigned to its cursor as a reference to track events and shares the path with other processes. The problem arises when an instance of buffer *B* violates one or more events in *a* [*i*] which are in common with the previous runs. For example, if $e_8$ in *B* invalidates any of *i* (Gyllstrom *et al*., 2007) instances, namely $e_3$ which is in common with 1.0.0 version of $e_6$ in *B*, the path in *a* [*i*] is not straight forward as addressed in Agrawal *et al.* (2008) that "If two runs, despite their distinct history, have the same computation state at present, they will select the same set of events until completion". We even thought about an alternative solution for their system through assigning extra pointers to jump over uncommon events in *a* [*i*], but it can dramatically increase number of pointers and neutralize the benefit of shared buffer. In Cugola and Margara (2010, 2012) they improved kleene operator on finite state machine, however these systems are unable to support two of key event operators namely, conjunction and disjunction.

In contrast to these systems, FAEM avoids aforementioned problems through applying the method presented in kleene num section. FAEM is implemented based on BTDG and supports conjunction between pair of event classes when one of the event classes is either kleene*, +or num. In performance section, we present the experiments on kleene and conjunction in FAEM. For the patterns with length of one, FAEM inserts a dummy buffer as the left child of the binary tree and the result of applying kleene is placed in their parent buffer.

Table 1: Events population in 1 million records of Yahoo! music dataset

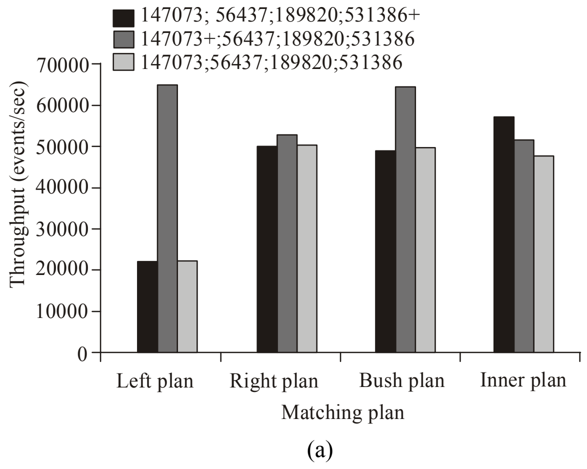
| Track ID | t147073 | t56437 | t189820 | t531386 |
|---|---|---|---|---|
| Population in 1 million records | 1661 | 1350 | 1325 | 1268 |

**Performance:** The performance in CEP systems is measured as maximum of rate of input which can be processed in a second. We have not considered the time to read from input and to display output. The performance is affected by various factors including, order of joins, population of events on kleene operator, Selectivity of Events, Time window on throughput and memory consumption, position of kleene on throughput and allocated memory that we present in details. In addition, the process is performed in-memory, so memory allocation is an important measure that we show here.

However, if in the 3rd query, we change kleene+ into kleene num, ZSTREAM has two shortcomings affecting the performance:

- It creates many events during applying kleene on a ternary Google; Microsoft (Mei and Madden, 2009); Dell which later in join with Intel some of these events may found invalid and needs to be removed. Due to this extra burden on kleene in ZSTREAM, throughput of the system is affected negatively.
- The limitation of ternary on kleene, limits ZSTREAM is unable to switch to an efficient matching plan.

The population of events in each buffer affects on performance of matching. In KDD Cup 2011 dataset we selected the most frequent events to define queries on them with the population presented in Table 1 aggregated on 1 million users' ratings.



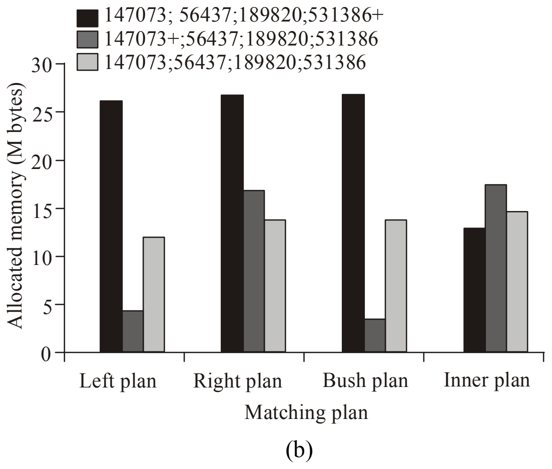


Fig. 7: Throughput and memory of kleene on various plans

**Order of joins:** The matching plans determine the order of applying join on each pair of event classes. For a Pattern of a query with the length of 4 as G; M; D; I, there are several plans can be applied as Left, Right, Bushy and Inner a ((*G*; *M*); *D*); *I*, (*G*; (*M*; (*D*; *I*))), ((*G*; *M*); (*D*; *I*)) and (*G*; ((*M*; *D*); *I*)) respectively. The Fig. 7 shows throughput and allocated memory for FAEM on 1,000,000 events on three queries as below:

- PATTERN 147073; 56437; 189820; 531386+ WITHIN 5000 unit
- PATTERN 147073+; 56437; 189820; 531386 WITHIN 5000 unit
- PATTERN 147073; 56437; 189820; 531386 WITHIN 5000 unit

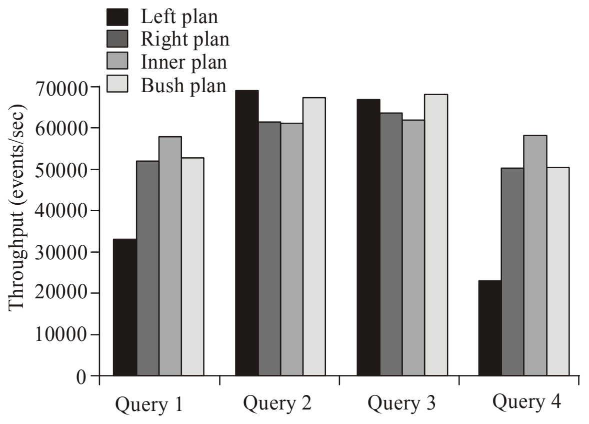
When the kleene is on the right side, the Inner plan performs better than other plans. The results of applying (56437; 189820) generates less intermediate results than any other pair of adjacent buffers. For Right and Bushy plans because kleene is appeared on the right side (189820; 531386+), the number of composite events as result of join never exceed number of events in (189820); However, when a new instance of (531386) is received and found match, the length of composite events is increased for all existing events in (189820; 531386+) buffer. Thus, Right and Bushy plans are not as efficient as Inner plan; However, these two plans perform better than Left plan because of rather high number of events in (147073) buffer that generates large number of composite events which many of these composite events are discarded as they do not satisfy time-window constraints.

When the kleene is located on the leftmost event class, the throughput of Left and Bushy plans outperforms other plans. As result of applying Left plan, the number of composite events is decreased in compare with a case where the kleene is does not exists in query. The number of composite events never exceed number of events in right side. For example: (147073; 56437) generates more composite events than (147073+; 56437) and number of composite events in (147073+; 56437) never exceed the number of events in (56437), due to grouping of events in (147073) matched with respect to every instance of (56437). The less number of intermediate results causes the less memory required for event matching, so Bushy and Left plans required less memory in compare to Right and Inner plans. The reason of achieving better performance and memory consumption for Left and Bushy plans on 2nd query is, applying join when kleene event is on the left side is less costly than the cases where kleene is on the right side of a join or there is no kleene in query because time complexity of Algorithm 1 is less than Algorithm 2. The worst memory consumption happens when the right most buffer is a kleene buffer and Bushy plan is selected because both of buffers involved in (147073; 56437) are highly populated and generates many results.

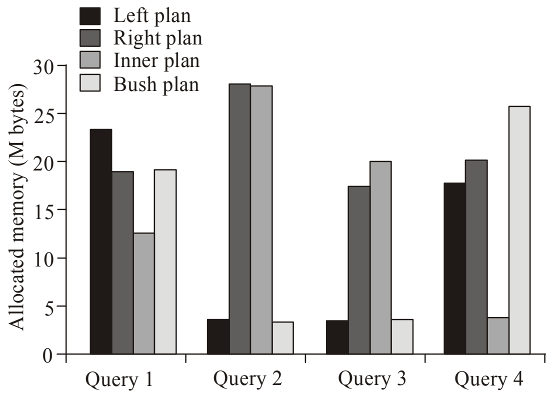
When there is no kleene in the query, Left plan performs worst of all, because many intermediate results are generated in early stages of matching which many of them may discard from further process as they do not fulfill the requirement of time window constraints. Starting from less populated event classes which are located on right and the efficiency of back-track event matching in preventing generation of excessive intermediate results, Right plan performs slightly better than Bushy and Inner plan on the 3rd query.

**Population of events in kleene:** We applied four queries on a time window with size of 5000 events. The Pattern section of these queries are as follows:

- 531386; 189820; 56437; 147073+
- +531386; 189820; 56437; 147073
- +147073; 56437; 189820; 531386
- 147073; 56437; 189820
- The size of time-window for all these queries is selected 5000 events

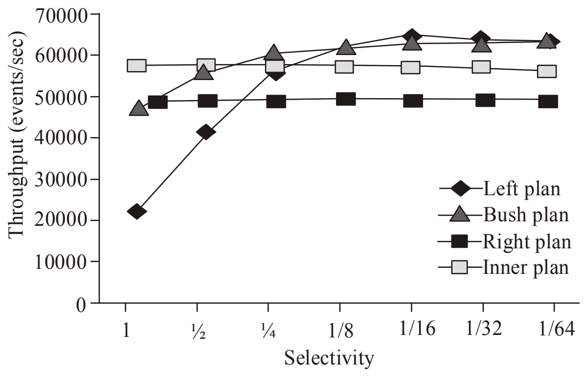


(a)

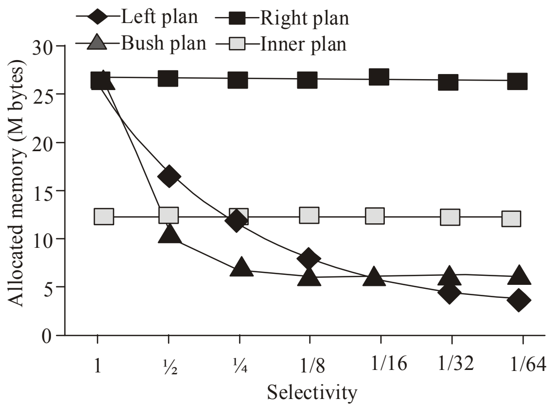


(b)

Fig. 8: Population of events on throughput and memory for various queries



(a)



(b)

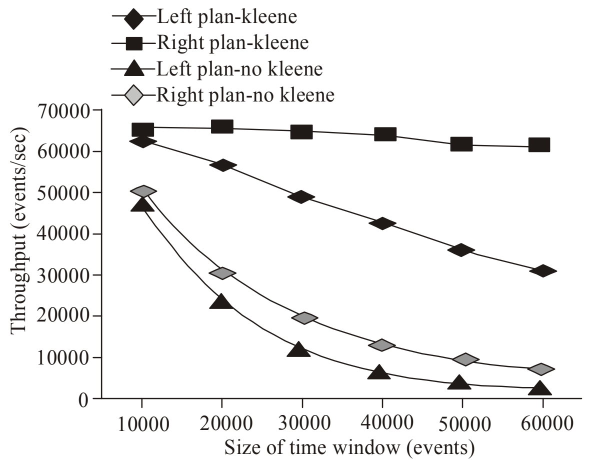
Fig. 9: Position of aggregation on throughput and memory for various queries

The Fig. 8 shows the performance and memory consumption of the system when the population of events affect on different plans.

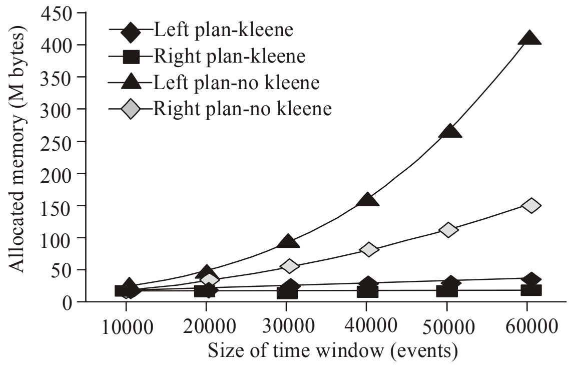
For Query 1, left plan Inner plan out performs other plans because it consumes the least memory and provides the highest throughput, while left plan consumes too much memory to apply matching and needs a rather extended time to perform the same task. It is notable that population of 147073 is more than 531386 and it is expected to have left plan prominent; however, grouping of events for 147073 in the right side of the query reduces the overall number of composite events in right side and increases the speed of matching. For the second query the grouping is applied on left most event class and performance is higher for plans containing joins between two first even classes. In Query 3, the grouping is applied on the left most event class and causes to achieve a higher performance for Left and Bushy plans. Finally for the forth query, the population of events in Left plan is much higher than others, so Left plan performs worst than other plans. It can be seen, the population of events along with the position of kleene in a query are influential to select the best plan.

**Selectivity of events:** In WHERE clause of queries, logical constraints may affect on performance regarding the filtering power of the predicates. The filtering power, selectivity, is tested on PATTERN 147073; 56437; 189820; 531386+. Figure 9 show throughputs and the amount of allocated memory for event matching in various rates of selectivity between two first event classes 147073 and 56437. The various rate of selectivity between 147073 and 56437 has a great effect on Left and Bushy plans due to these two classes are evaluated in finer levels of hierarchy. Left plan is the more affected than Bushy plan because of descending number of events from left to right. The Right and Inner are not much affected because the filtering between 147073 and 56437 is applied in final stage where not many composite events are discarded. Similarly, the various rates of selectivity between 189820 and 531386 is not much effective on Left plan but other plans are affected.

**Time window on throughput and memory consumption:** Figure 10 shows the effectiveness of time window on throughput of event matching. The performance for patterns with kleene shows better results than no-kleene because kleene collects as many

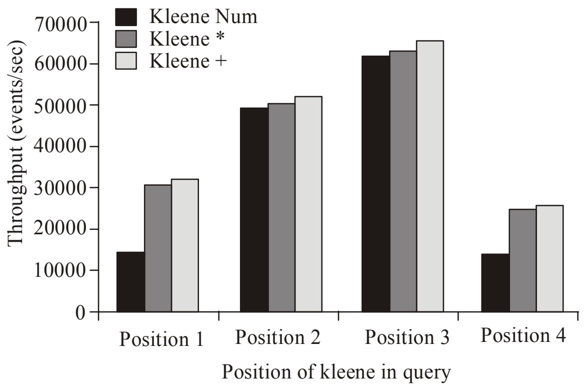


(a)

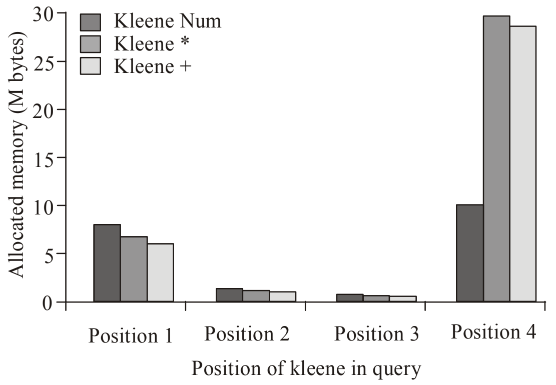


(b)

Fig. 10: Length of time-window on throughput and memory for various queries



(a)



(b)

Fig. 11: Position of kleene on throughput and allocated memory

events in a group to create a single composite event and produces much less distinctive composite events.

This saves memory space and rather high throughput for patterns with no kleene. On the other hand, Right plan performs better than Left plan as it has less number of composite events in lower levels of hierarchy. However, increasing size of time window increases the allocated memory and more events to evaluate decreases the throughput.

**Position of kleene on throughput and allocated memory:** Figure 11 shows the throughput and memory space in Right plan when the position of kleene Num (with size = 3), + and * changes from 1st to 4th position on the PATTERN 147073; 56437; 189820; 531386 WITHIN 10000 unit.

Kleene Num performs the best of all; however, the gap between kleene Num and other plan is higher when the kleene is located on first and last position. Because only 3 events, in accordance to size of kleene num are selected and needs an extra step to generate combinations, the throughput is less than all. The amount of consumed memory for the kleene Num except for the fourth position is more than others. Because in final stage of matching many events are generated that their size in kleene num are 3, while the final stage is not applied for the kleene+ and *. For the 4th position, kleene Num consumes 1/3 of memory in compare to kleene+ and * due to generating less composite events with the size of 3.

## CONCLUSION

The focus of this study is on improving expressibility and performance measures of kleene operator on our previously developed BTDG. Through addressing the shortcomings of both NFA-based and ternary tree-based systems to support kleene operator, we developed kleene operator including three algorithms for kleene+, * and num. The experiments show that our implementation supports wider range of queries than these systems. In addition, the high performance of the system is due to fast directing events into higher levels of event hierarchy through the cursors, the dynamic installation of optimal plan as well as, late applying combinations because some of the events are discarded during matching.

## ACKNOWLEDGMENT

Support for this project from UTM Big Data Centre, Universiti Teknologi Malaysia in the form of UTM Flagship Project-UTM E-Learning Big Data Analytics is gratefully acknowledged.

## REFERENCES

Agrawal, J., Y. Diao, D. Gyllstrom and N. Immerman, 2008. Efficient pattern matching over event streams. Proceeding of the ACM SIGMOD International Conference on Management of Data. Vancouver, Canada, pp: 147-160.

Behravesh, B., S. Mariyam and A.H.T. Sim, 2014. Optimizing throughput and end-to-end latency in complex event processing. Under Revision in Math Probl. Eng., pp: 1-33.

Cugola, G. and A. Margara, 2010. TESLA: A formally defined event specification language. Proceeding of the 4th ACM International Conference on Distributed Event-based Systems, pp: 50-61.

Cugola, G. and A. Margara, 2012. Processing flows of information: From data stream to complex event processing. ACM Comput. Surv., 44(3): 1-70.

Demers, A.J., J. Gehrke, B. Panda, M. Riedewald, V. Sharma and W.M White, 2007. Cayuga: A general purpose event monitoring system. Proceeding of the 3rd Biennial Conference on Innovative Data Research (CIDR, 2007), Jan. 7-10, pp: 412-422.

Diao, Y., N. Immerman and D. Gyllstrom, 2008. SASE+: An agile language for kleene closure over event streams. Technical Report, UMass, pp: 1-13.

Gyllstrom, D., Y. Diao, E. Wu, P. Stahlberg and G. Anderson, 2007. SASE: Complex event processing over streams. Proceeding of the Biennial Conference on Innovative Data Systems Research (CIDR), pp: 407-411.

Mei, Y. and S. Madden, 2009. Zstream: A cost-based query processor for adaptively detecting composite events. Proceeding of the 35th SIGMOD International Conference on Management of Data, pp: 193-206.

Muthusamy, V., H. Liu and H.A. Jacobsen, 2010. Predictive publish/subscribe matching. Proceeding of the 4th ACM International Conference on Distributed Event-based Systems, pp: 14-25.

Peer, B., P. Rajbhoj and N. Chathanur, 2013. Complex events processing: Unburdening big data complexities. Infosys Labs Briefings, 11(1): 53-64.

Wang, D., E.A. Rundensteiner, H. Wang and R.T. Ellison, 2010. Active complex event processing: Applications in real-time health care. Proceeding of 36th International Conference on Very Large Data Bases, Sept. 13-17, pp: 1545-1548.

Wu, E., Y. Diao and S. Rizvi, 2006. High-performance complex event processing over streams. Proceeding of the 2006 ACM SIGMOD International Conference on Management of Data, pp: 407-418.