6-2011

High-Performance Composite Event Monitoring System Supporting Large Numbers of Queries and Sources

Sang Jeong LEE
KAIST

Youngki LEE
Singapore Management University, YOUNGKILEE@smu.edu.sg

Byoungjip KIM
KAIST

K. Selcuk CANDAN
Arizona State University

Yunseok RHEE
Hankuk University of Foreign Studies

See next page for additional authors

DOI: https://doi.org/10.1145/2002259.2002280

High-Performance Composite Event Monitoring System Supporting Large Numbers of Queries and Sources

SangJeong Lee, Youngki Lee, Byoungjip Kim, K. Selçuk Candan*, Yunseok Rhee§, Junehwa Song

Korea Advanced Institute of Science and Technology
Computer Science Department
{peterlee, youngki, bjkim, junesong}@nclab.kaist.ac.kr

*Arizona State University
School of Computing, Informatics, and Decision Science Engineering
candan@asu.edu

§Hankuk University of Foreign Studies
School of Electronics and Information Engineering
rheeys@hufs.ac.kr

ABSTRACT

This paper presents a novel data structure, called *Event-centric Composable Queue (ECQ)*, a basic building block of a new scalable composite event monitoring (CEM) framework, *SCEMon*. In particular, we focus on the scalability issues when large numbers of CEM queries and event sources exist in upcoming CEM environments. To address these challenges effectively, we take an *event-centric sharing approach* rather than dealing with queries and sources separately. ECQ is a shared queue, which stores incoming event instances of a primitive event class. ECQs are designed to facilitate efficient shared evaluations of multiple queries over very large volumes of event streams from numerous event sources. ECQs are composable and form a single shared network within which multiple queries are simultaneously evaluated. In this paper, we present efficient shared processing techniques operating on top of the proposed shared ECQ network. The performance evaluation shows that the proposed approach achieves a high level of scalability compared to conventional separate processing approaches in large-scale CEM environments.

Categories and Subject Descriptors

H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval – *Information filtering*; H.2.4 [**Database Management**]: Systems – *Query processing*

General Terms: Algorithms, Performance, Design.

Keywords: Composite Event Monitoring, Scalable Processing, Event Streams.

1. INTRODUCTION

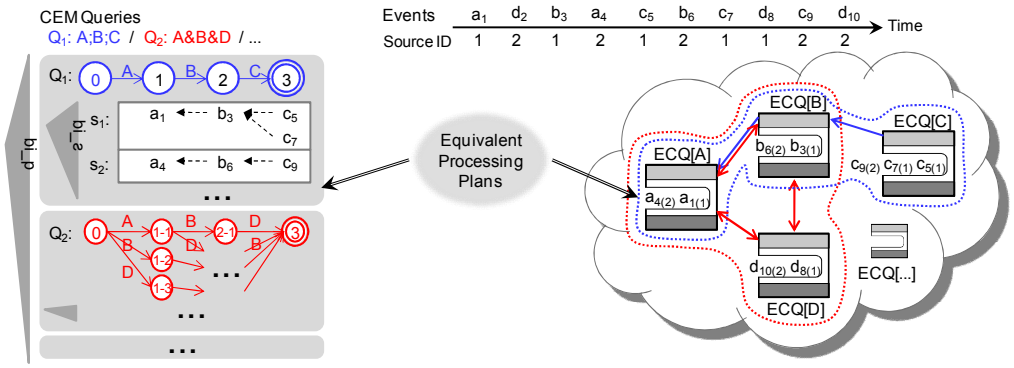
Efficient monitoring of composite events over large volumes of event streams is critical in many application domains, including product management [1], network monitoring [2], stock market analysis [3], and traffic monitoring [4]. In many applications, a multitude of composite event monitoring (CEM) queries are registered and all of them are simultaneously monitored by the system over the same event streams. Previous research in composite event detection, however, has focused on optimizing the monitoring of *individual* queries [1][3][5]. We note that optimizing system resources "*separately*" for each query has inherent limitations when the system needs to deal with large numbers of simultaneous queries and event sources. Thus, we propose a novel scalable CEM framework that efficiently evaluates in a "*shared*" manner large numbers of CEM queries against input streams from numerous event sources.

Challenges. CEM frameworks are often confronted with the scalability challenges that arise from the presence of very large numbers of (a) simultaneous CEM queries and (b) event sources. For example, to identify effective advertising targets, a credit card company may want to identify card holders following certain purchasing event patterns of many diverse scenarios such as couple dating, sporting events, shopping sprees, travel, etc. Each of these cases would be represented as a multitude of CEM queries registered in the system and they all would be tracked simultaneously over the stream of credit card transaction events. In a metropolitan city, there often exist thousands of purchasing patterns of interest as well as millions of credit card holders.

A straightforward approach to process simultaneous CEM queries is to evaluate these individual queries separately [1][3][5]. Figure 1-(a) describes the approach using multiple CEM queries. In this setup, given a set of CEM queries, as many query processing plans need to be created and evaluated. Moreover, incoming event instances need to be delivered to the relevant plans and possibly stored in each plan for later query evaluation. It is obvious that such an approach would be extremely wasteful: Although there are common events engaged in multiple processing plans, their storage and computation cannot be shared effectively across different plans. Processing times would then increase with the number of queries and input rates. Moreover, the approach would require considerable storage space to hold incoming instances and intermediate states for each plan; this would make high-performance in-memory processing of large-scale CEMs difficult.

Recently, the researcher community started considering multiple event sources. Yet, effective approaches dealing with very large numbers of sources are elusive. Wu et al. showed that in non-deterministic finite automata (NFA) based CEMs, unnecessary state transitions can frequently occur when there are different event sources [1]. To tackle this challenge, they partition stacks of event instances for separate processing of individual sources. Note that there exist separate stacks partitioned for different sources in the processing plan of query Q_1 in Figure 1-(a). However, when the number of sources is very large, this implies that a large

DEBS'11, July 11–15, 2011, New York, New York, USA.



(a) Separate processing scheme (b) Event-centric sharing approach

Figure 1. "Separate processing" scheme vs. proposed "event-centric sharing" approach

number of separate stacks need to be created, resulting in severe storage and management overheads.

Proposed Solution. In this paper, we take an *event-centric sharing* approach to address the inefficiencies due to duplicated data structures and separate processing of conventional CEMs when supporting large numbers of queries and sources. Our approach is based on the idea that a primitive event, specified commonly in multiple queries, can be shared for efficient processing and storage. Moreover, all incoming instances of a given event class, regardless of their sources, can be stored and handled together within a shared storage, e.g. a queue. Based on the composition patterns of all registered CEM queries, these shared queues can form a single *shared network* in which processing and storage for each primitive event class are inherently shared by all relevant queries. For instance, Figure 1-(b) illustrates the key idea of the proposed approach; the queues are shared by two event sources as well as two CEM queries. The comprehensive discussion on Figure 1 will be given in Section 3.

Our Contributions. Based on the observations of shared processing opportunities, we develop a new scalable composite event monitoring (CEM) framework, *SCEMon*. SCEMon, in its core, is an automata-based architecture; but it consists of data structures and algorithms that are designed to maximize event and sub-pattern sharing across multiple queries as well as sources. The contributions of this paper can be summarized as follows:

- We explore the scalability problem arising in large-scale CEM environments. We investigate the performance of conventional separate processing schemes and explore inherent limitations when dealing with large numbers of queries and sources.
- We then propose a novel data structure, *event-centric composable queue (ECQ)*, that enables efficient shared processing of large numbers of simultaneous queries and event sources. An ECQ is a shared queue storing incoming event instances of a primitive event class. For each primitive event class, only a single ECQ is allocated and is shared by multiple CEM queries. ECQs are composed flexibly within a single shared network to support the diverse composition semantics of the queries. Each ECQ is also shared by all event sources. This design substantially reduces processing and storage overhead necessary to manipulate intermediate results for each query and event source separately.
- On top of this sharable data structure, we develop a suit of efficient shared processing techniques including *event instance sharing*, *sub-pattern sharing*, and *partial matching block (PMB) reduction*. These techniques are brought together in SCEMon which localizes each instance manipulation on a corresponding ECQ and a few *adjacent* ECQs, and evaluates CEM queries incrementally with each subsequent event instance. SCEMon supports various types of composite event patterns such as sequence, conjunction and disjunction.
- We experimentally demonstrate that relying on the novel data structures and shared processing techniques, the scalability issues can be tackled effectively. The performance results of our extensive evaluation show the competitive performance of SCEMon against conventional CEM approaches.

The rest of the paper is organized as follows: Section 2 introduces related work. Section 3 discusses the proposed approach of SCEMon in comparison to conventional approaches. Section 4 presents the data structure of ECQ and Section 5 describes the shared processing techniques using ECQs. Section 6 gives the performance cost analysis. Section 7 discusses the experimental results for performance, and finally Section 8 concludes the paper.

2. RELATED WORK

Event monitoring systems have evolved and been expanded for diverse application domains, online transaction logs [8], built-in-sensor reporting in a building [9], RFID readings in a market [1] and stock trading [3]. The current approaches can be roughly classified into *automata*-based complex event management systems, such as SASE [1] and Cayuga [5], *Petri Net*-based systems like SAMOS [6], *event tree*-based systems, such as Sentinel [7] and ZStream [3], and *event graph*-based systems including InfoFilter [10].

There have been continuous research efforts to improve the performance of CEMs. SASE [1][11] extends non-deterministic finite automata (NFA) to deal with multiple event sources. Cayuga [5], also NFA-based, focuses on efficient predicate evaluation using indices along with automata transition. Recognizing that NFA-based CEMs are limited to sequential patterns due to the explicit state transitions of NFAs, ZStream [3] takes an event tree-based approach to support rich composition semantics such as concurrent events or negated events that should not occur. It provides the cost model for different composition patterns and the optimization technique to search for an optimal evaluation plan. Akdere et al. also develop the event graph-based CEM across distributed event nodes [2]. They generate multi-step event acquisition and processing plans that minimize event

transmission costs. However, conventional CEMs have difficulties in dealing with large numbers of simultaneous queries and event sources together. Most of them treat multiple queries and sources separately; their main contributions are not to develop shared processing techniques, but to optimize individual processing plans per query and source. Such separate CEM processing may potentially limit the scalability required for massive processing.

Previous works on multi-query optimization, e.g., predicate indexing [5], sub-graph merging [10], and sub-event sharing [2] can be considered as efforts to address the problem. However, it is not straightforward to make the data structures of existing CEMs be shared effectively, since they are still founded on NFAs or event trees. A state in an NFA represents not only the current event class but also the history of state transitions with past event instances. Thus, the state can hardly be shared unless the state transitions to the state from the beginning are identical between different NFAs. Achieving performance benefits through sharing would be moderate due to the rare chance of sharing. Since an intermediate node in an event tree also designates partial compositions, it can be rarely shared among multiple queries.

3. COMPOSITE EVENT MONITORING

The CEM semantics and language we adopt in this paper are analogous to those used in other CEM systems [1][3][11]. Based on the basic CEM notation, we present a common approach of CEM processing and discuss the potential challenges in large-scale CEM environments. Then, we introduce our event-centric sharing approach dealing with such scalability challenges.

3.1 CEM Notation

We define **primitive events** as atomic occurrences of interest. More precisely, we represent an incoming event instance as a tuple <*src_id*, *event_class*, *start_ts*, *end_ts*, *attrs[]*>, where *src_id* refers to the identifier of the event source, *event_class* refers to the class that the instance belongs to, *start_ts* and *end_ts* refer to the start and end timestamps of the event instance respectively, and *attrs[]* refers to the list of attribute values.

On receiving primitive events, **composite events** are detected from a collection of primitive and/or other composite events. CEM queries associate primitive or composite events together to form new composite events. The most frequently used composition type **sequence (A;B)** finds the instances of event B following the instances of event A within a specific time window. **Conjunction (A&B)**, i.e., concurrent events, denotes that event A and event B occur within a specified time window in any orders. **Disjunction (A|B)** means that either event A or event B occurs. This is simply a union of the two event classes and, in its most generic definition, no time constraints on the events are included.

The formal semantics of CEM queries with different patterns is given in Table 1. The PATTERN clause specifies the type of composition patterns such as sequence, conjunction, and disjunction. The WITH clause presents a list of the event classes that should occur to form the composite event. The WHERE clause imposes predicates on event attributes while the WITHIN clause describes the time window for the events.

Upon an input event instance, each CEM query can generate different results depending on a *selection mode*. It can generate at most one composite event instance which represents the most recent composition of participating events. This can be considered as a *recent* selection mode in active database among several different composition modes [8]. An *all* selection mode is also used frequently that generates all composite event instances satisfying the monitoring conditions. SCEMon can support the two modes; different output generation of SCEMon depending on different selection modes is discussed later in Section 5.1.

Table 1. Formal semantics of CEM queries

Given event instance stream, e_strm = (e_1, e_2, ..., e_i, ...) – infinite series
Upon the arrival of e_i= <src_id, event_class, start_ts, end_ts, attrs[]>,
each query generates composite event instances, c's, satisfying the below conditions:

Pattern	Query Language	Monitoring Semantics
Sequence	**qry_seq:** PATTERN Sequence WITH $E_1, E_2, ..., E_n$ WHERE [symbol] WITHIN t_cond	c = <src_id, qry_id=qry_seq.id, start_ts, end_ts, (e_{M1}, e_{M2}, ..., e_{Mn})>, where • c.src_id = e_i.src_id = e_{Mj}.src_id for all $1 \le j \le n$, • e_{Mj} is an instance of the event class E_j for all $1 \le j \le n$, • e_{Mj-1}.end_ts ≤ e_{Mj}.start_ts for all $2 \le j \le n$, • c.start_ts = e_{M1}.start_ts, c.end_ts = e_{Mn}.end_ts, • (c.end_ts – c.start_ts) ≤ qry_seq.t_cond, • $e_{Mn} = e_i$, and • ∀ e_k ∋ e_k.src_id = e_i.src_id and c.start_ts < e_k.start_ts ≤ e_k.end_ts < c.end_ts.
Conjunction	**qry_cnj:** PATTERN Conjunction WITH $E_1, E_2, ..., E_n$ WHERE [symbol] WITHIN t_cond	c = <src_id, qry_id=qry_cnj.id, start_ts, end_ts, (e_{M1}, e_{M2}, ..., e_{Mn})>, where • c.src_id = e_i.src_id = e_{Mj}.src_id for all $1 \le j \le n$, • e_{Mj} is an instance of the event class E_j for all $1 \le j \le n$, • c.start_ts = min({e_{Mj}.start_ts}) for all $1 \le j \le n$, c.end_ts = e_i.end_ts = max({e_{Mj}.end_ts}) for all $1 \le j \le n$, • (c.end_ts – c.start_ts) ≤ qry_cnj.t_cond, and • ∀ e_k ∋ e_k.src_id = e_i.src_id and c.start_ts < e_k.start_ts ≤ e_k.end_ts < c.end_ts.
Disjunction	**qry_dsj:** PATTERN Disjunction WITH $E_1, E_2, ..., E_n$	c = <src_id, qry_id=qry_dsj.id, start_ts, end_ts, (e_i)>, where • c.src_id = e_i.src_id, • e_i is an instance of the event class E_j for any $1 \le j \le n$, • c.start_ts = e_i.start_ts, and c.end_ts = e_i.end_ts

CEM queries can be used to specify diverse purchasing event patterns for credit card companies. CEM query 1 below presents an example of a sequential pattern.

CEM query 1. Sequential pattern

PATTERN	*Sequence*
WITH	*CNMA_A, RSTR_B, BAR_C*
WHERE	*[symbol]* *$20 < CNMA_A.payment < $50* AND *$80 < RSTR_B.payment < $120* AND *BAR_C.payment < $50*
WITHIN	*5 hours*

This query is intended to represent a particular purchasing event pattern potentially related to "dating", i.e., two seats purchased at a theater (CNMA_A), meals for two at a restaurant (RSTR_B), and some (but not too much) drinking at a bar (BAR_C). Note that [symbol] means the condition of matching source ids among incoming event instances.

CEM query 2. Conjunction pattern

PATTERN	*Conjunction*
WITH	*BRND_A, BRND_B, BRND_C*
WHERE	*[symbol]* *BRND_A.payment + BRND_B.payment + BRND_C.payment < $200*
WITHIN	*1.5 hours*

CEM query 2 may represent a shopping pattern in an outlet mall. Since the shopping order does not matter here, the query uses the conjunction type. As discussed in [12], such queries can be handy for shop managers who would like to send coupons or advertisements to attract the customers who have not bought brand goods sufficiently.

These types of queries open the opportunity for credit card companies to advanced mobile advertising and business promotions based on credit card holders' purchasing patterns. A large number of CEM queries can be created in various ways over a given set of available purchasing event classes, and issued by

third-party advertising agencies or business owners to target their own potential customers in mobile computing environments.

3.2 Query/Source-Separate Processing

A common approach to CEM processing involves developing separate processing plans for individual queries and dealing with individual sources separately in each plan; this is illustrated in Figure 1-(a) with multiple CEM queries, i.e., Q_1, Q_2, etc. Such an approach mostly takes full advantage of indices built over many queries and sources. Upon an input event instance, it would identify queries of interest (which involve an event class of the instance in their patterns) by using the query index. For each relevant query processing plan, it would *search* for the data structure designated to an event source of the instance, *evaluate* relevant composition transitions and *store* the instance and intermediate evaluation results into the data structure if necessary. Additionally, it would *delete* any obsolete stored instances from query plans for efficient memory management.

For example, upon an input event instance b_3 from source #1, the approach would identify Q_1 and Q_2 using the 'q_id' index and invoke the processing plans of Q_1 and Q_2 respectively. Each plan would be evaluated with b_3 and store it into the instance stacks responsible for the corresponding source respectively. Note that b_3 is stored twice in the stacks of s_1 in Q_1 and Q_2 plans. Later, it would be deleted from each plan if it is determined to be no more necessary for further processing.

In large-scale CEM environments, there may exist numerous CEM queries of interest for each input event instance since a large number of simultaneous queries are specified using a set of event classes. In such cases, the same *search*, *evaluate/store*, and *delete* operations may need to be invoked repeatedly many times and this may result in very huge processing overheads. Specifically, the processing cost is significantly influenced by the numbers of queries and sources. First, the processing overhead caused by search, evaluate/store, and delete operations are multiplied by the number of the evaluated query plans which would substantially increase with larger numbers of simultaneous queries. Second, when there are a large number of event sources, the costs of the individual operations can be significantly raised due to the severe management overhead of numerous separate data structures assigned for individual sources. Even with an index built on source ids, searching for the data structure of a specific event source mostly takes up O(log *Ns*) time[1], where *Ns* is the number of sources. Thus, "separate processing" schemes can hardly cope with large-scale CEM environments.

3.3 Event-centric Sharing Approach

In this paper, our goal is to develop an efficient shared processing approach that deals with such large numbers of simultaneous CEM queries and event sources. SCEMon takes advantage of a novel data structure, called ECQ, which manages all incoming instances of a primitive event class together regardless of queries and sources. SCEMon identifies a set of essential primitive event classes for all queries, constructs a single network of corresponding ECQs respectively taking each class in charge, and evaluates all the queries simultaneously in conjunction with the constructed network. Figure 1-(b) illustrates the proposed event-centric sharing approach of SCEMon; it visualizes a shared ECQ network where four ECQs are composed into two *virtual* processing plans which are equivalent to the first two plans in Figure 1-(a). Note that, in this example, upon arrival of b_3 or b_6, Q_1 and Q_2 can be evaluated together by ECQ[B]. In essence, ECQ enables multiple sources to easily share the processing for their respective instances, and further enables multiple queries to aggressively share their common processing.

Our event-centric sharing approach is especially advantageous when there are many popular event classes of common interest specified in registered CEM queries. Let us consider the mobile advertising application discussed earlier and note that modern cities have many hot spots such as popular shopping complexes and multiplex cinemas. A large portion of CEM queries will involve such hot places, and primitive events happening in the places may trigger the evaluation of large numbers of simultaneous CEM queries. SCEMon is expected to be highly effective in such a scenario.

In addition, the proposed approach is highly beneficial in monitoring long term patterns, where the processing tends to rapidly increase the volume of intermediate evaluation results. For example, human activity patterns of interest often involve long-term processing for several hours or even days. Our sharing approach can substantially reduce the amount of the intermediate results, and thus makes the long-term processing more effective in terms of storage consumption as well as computation.

4. EVENT-CENTRIC COMPOSABLE QUEUE (ECQ)

As the basis of SCEMon, this section presents the data structure of ECQ, and constructs the shared network of ECQs developed for the efficient shared processing of SCEMon.

ECQ maintains three data structures *shared instance queue (SIQ)*, *composition link table (CLT)*, and *partial matching block (PMB)*. Figure 2 illustrates the state of a specific ECQ, denoted as ECQ_i, that deals with the *k*-th incoming event instance, e_k.

Shared Instance Queue (SIQ) manages the recent event instances for all event sources with regard to all event classes. It stores the event instances in the order of their arrivals. This single instance queue in an ECQ is shared by all relevant event sources. SIQ uses a hash table with *source_id* as its key to facilitate accesses to the recent instance e_k.

Composition Link Table (CLT) enables the construction of an integrated ECQ network that supports the shared processing of CEM queries. For ECQ_i, the corresponding CLT contains a set of composition links, one for each CEM query that ECQ_i participates in. Each link, denoted as CLink(ECQ_i, Q_j), represents the association of ECQ_i with the other ECQs specified in the *j*-th CEM query, Q_j.

CLink(ECQ_i, Q_j) is formally described as a 6-tuple *(query_id, type, t_cond, {ptr_ECQ}, flag, attr_cond)*, where *query_id, type, t_cond,* and *attr_cond* are the identifier, type, time constraint and attribute condition of Q_j, respectively[2].

{ptr_ECQ} and *flag* play critical roles in network construction:

[1] Due to the memory limit, the hash lookup with O(1) search time can hardly be used in practical main-memory systems. Memory-efficient tree-based hash tables could be used instead.

[2] We regard the query_id of Q_j as *j* for the convenience of explanation.

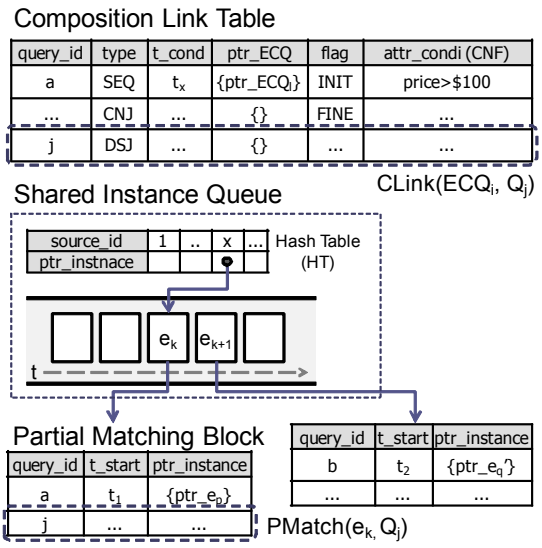


Figure 2. Data structure of ECQ

- *{ptr_ECQ}* contains the pointers to the other ECQs. The pointers facilitate tracing of the related ECQs in the network.
- *flag* marks the position of the ECQ_i in the query; it can be INIT to indicate the first ECQ starting the composition, FINE to indicate the last ECQ finishing the composition.

Figure 1-(b) shows the composition link examples using the blue and red arrows for Q_1 and Q_2, respectively. For the sequence query Q_1, the CLink of ECQ[A], i.e., CLink(ECQ[A], Q_1), is (1, SEQ, t_1, {}, INIT, null), while that of ECQ[C] is (1, SEQ, t_1, {ptr_ECQ[B]}, FINE, null). For the conjunction query Q_2, CLink(ECQ[A], Q_2) is (2, CNJ, t_2, {ptr_ECQ[B], ptr_ECQ[D]}, INIT|FINE, null). Note that CLink(ECQ[A], Q_2) points to the other two ECQs and it is also marked as INIT and FINE since any ECQ in Q_2 can start and finish the conjunction composition.

Partial Matching Block (PMB) supports incremental evaluation of CEM queries. As shown in Figure 2, a block is allocated to each event instance e_k to store the current states of partial matching in which e_k participates. The block has a set of partial matching entries, one for each composition query. The block allows the incremental extension of partial matching until the matching becomes completed. PMatch(e_k, Q_j), if it exists, represents that the partial matching of the query Q_j has been successfully extended by the instance e_k at ECQ_i. PMatch(e_k, Q_j) is formally specified as a tuple *(query_id, t_start, {ptr_instance})*;

- *query_id* is the identifier of Q_j,
- *t_start* is the start time of the partial matching, and
- *{ptr_instance}* is a set of pointers to the precedent instances, stored in other ECQs, leading to the current partial matching.

For example, an input event instance b_3 in Figure 1-(b) would have two PMatch entries for Q_1 and Q_2. PMatch(b_3, Q_1) is (1, 1, {$a_{1(1)}$}) since the partial matching is initiated at time 1, i.e., the start time of $a_{1(1)}$, and the precedent instance is $a_{1(1)}$. On the other hand, PMatch(b_3, Q_2) is (2, 3, {}) since b_3 initiates a new partial matching of conjunction and no precedents are required.

When the matching is complete, the pointers are followed iteratively to obtain all the participating event instances. Intuitively, the CLT of the ECQ_i for a primitive event class shows the schematic compositions in which ECQ_i participates, while the PMB shows the status of current partial matching in which a specific instance e_k of the primitive event class participates.

```
Input: N-ECQ and Q_j
Output: N-ECQ
1.  foreach event class specified in Q_j do
2.    if ECQ of the class does not exist in N-ECQ then
3.      create a new ECQ for the class and insert it into N-ECQ
4.  foreach ECQ_i corresponding to each event class specified in Q_j do
5.    create CLink(ECQ_i, Q_j) such that query_id ← Q_j.query_id, type ← Q_j.type,
                                  and t_cond ← Q_j.t_cond
6.    if CLink(ECQ_i, Q_j).type = SEQ then
7.      set INIT or FINE to CLink(ECQ_i, Q_j).flag w.r.t. position of ECQ_i in sequence
8.      add the pointer of the previous ECQ_p into CLink(ECQ_i, Q_j).ptr_ECQ
9.    if CLink(ECQ_i, Q_j).type = CNJ then
10.     add the pointers of all the other ECQs into CLink(ECQ_i, Q_j).ptr_ECQ
11. return N-ECQ
```

Figure 3. Algorithm for inserting a CEM query to SCEMon: N-ECQ denotes the shared network of ECQs

Given a set of CEM queries, SCEMon constructs a single network of ECQs. In the network, the ECQs of each query are networked with each other via composition links, or CLink's.

The algorithm for the network construction is presented in Figure 3. It is constructed by inserting a CEM query into the network as follows: For a new query, a new ECQ is instantiated for each primitive event specified in the WITH clause (Lines 1-3 in the figure). Some ECQs might not be created if they have already been defined in already registered queries. For the new query, the comprising ECQs are associated with each other by adding a CLink entry in their CLT (Line 5). For the sequence type, ECQs are linked sequentially; each CLink(ECQ_i, Q_j) points to the ECQ of the precedent activity, and the first and final ECQs are marked accordingly in the flag field (Lines 6-8). For the conjunction type, ECQs are linked and marked accordingly (Line 9-10).

Deleting a CEM query from SCEMon is straightforward. For each event class participating in the query, we remove the corresponding CLink entry in the corresponding ECQs. If the CLT becomes empty, the ECQ is deleted since it does not participate in any CEM queries.

5. SHARED PROCESSING TECHNIQUES

This section presents the shared processing algorithm running on top of the ECQ network. Then, the performance benefit for the proposed algorithm is discussed. We further develop advanced techniques available to improve the processing efficiency.

5.1 Shared Processing Algorithm with Instance Sharing

Upon arrival, each new event instance e_k is dispatched to its corresponding ECQ, say ECQ_i. The evaluation process inside the ECQ_i consists of two major phases: *test* and *insert*. The *test* phase evaluates whether e_k could lead to a partial or complete matching for some CEM queries. The *insert* phase updates the data structures of ECQ_i if a new composition happens.

5.1.1 Test Phase

ECQ_i identifies the set of active queries associated with it in the CLT. For each CLink entry of the CLT, it may *probe* the other neighboring ECQs specified in {ptr_ECQ} of the entry for testing the extension of partial matching. The probing is based on the

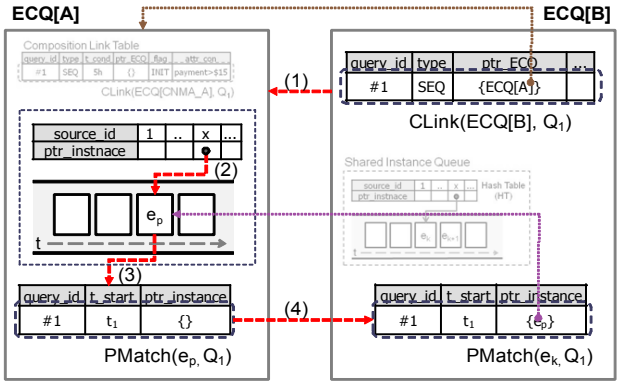


Figure 4. Processing flow of Probe function

source identifier of the incoming instance, i.e., e_k.source_id, and performed by looking at the target SIQ through the hash table. Probing is implemented as a single **Probe** function. Figure 4 illustrates the processing flow of the function using the two ECQs of the CEM query 1. Upon arrival of e_k in ECQ[B], the function looks up the recent event instance, e_p, of the same source with the incoming instance, e_k, in the target ECQ[A], i.e., e_p.source_id = e_k.source_id (Step (1) and (2) in the figure). It then finds from the PMB of e_p the existing partial matching entry for the query, PMatch(e_p, Q_1) (Step (3)). With PMatch(e_p, Q_1), the function tests if e_k can successfully extend the existing partial matching of Q_1. In detail, it is tested if the starting time of the partial composition, PMatch(e_p, Q_j).t_start, satisfies the time constraints of Q_j, i.e., Q_j.t_cond. If so, it returns the partial matching entry to designate the extension of the partial matching (Step (4)).

Using this Probe function, the test phase handles each composition pattern differently:

Sequences. The test phase deals with three different cases with respect to the position of ECQ_i in a sequence; *start*, *middle* and *end*. The pseudo code for the algorithm is presented in Figure 5. It first deals with the "*start*" case in which ECQ_i is marked as INIT in CLink(ECQ_i, Q_j). At ECQ_i, incoming e_k starts a new partial matching of Q_j; PMatch(e_k, Q_j) is created and the start time is set to the start time of e_k (Lines 1-3 in the figure). For the "*middle*" and "*end*" cases, the test phase probes the precedent ECQ, i.e., ECQ_p, in the sequence. If the Probe function confirms the extension of the partial matching, it creates a new entry PMatch(e_k, Q_j) for e_k (Lines 4-7). Especially for the "*end*" case that ECQ_i is marked as FINE in the CLink, if PMatch(e_k, Q_j) has been created already, the test phase completes the matching of Q_j with e_k and generates composite event instances as output by following the pointers in {ptr_instance} of PMatch entries (Lines 8-10). Finally, it returns the created PMatch(e_k, Q_j) to inform the insert phase of the status update (Line 11).

The output generation is different to support different selection modes, i.e., *recent* and *all*. For the recent selection mode, it follows the precedent instance pointers specified in {ptr_instance} of PMatch entries recursively to the initial matching ECQ and output a series of those instances in the form of a composite event instance. To support the all mode, every previous instance of the same source, stored in the same ECQ of the pointed instances, is harvested to compose output results as long as they satisfy the time conditions.

Input: CLink(ECQ_i, Q_j) and an instance e_k
Output: PMatch(e_k, Q_j) or null

1. **if** CLink(ECQ_i, Q_j).flag = INIT **then**
2. create PMatch(e_k, Q_j) such that t_start ← e_k.start_ts and ptr_instance ← null
3. **return** PMatch(e_k, Q_j)
4. ECQ_p ← CLink(ECQ_i, Q_j).ptr_ECQ[0] // only one pointer for sequence
5. PMatch(e_p, Q_j) ← ECQ_p.***Probe***(e_k, Q_j)
6. **if** PMatch(e_p, Q_j) != null **then**
7. create PMatch(e_k, Q_j) such that t_start ← PMatch(e_p, Q_j).t_start and ptr_instance ← {e_p}
8. **if** CLink(ECQ_i, Q_j).flag = FINE **then**
9. follow the pointers in PMatch(e_k, Q_j).ptr_instance iteratively and retrieve participating instances
10. construct and output the matching result (e_k.source_id, Q_j.query_id, PMatch(e_k, Q_j).t_start, e_k.end_ts, ({..., e_p, e_k}))
11. **return** PMatch(e_k, Q_j)
12. **return** null

Figure 5. Algorithm for sequence in test phase

Conjunctions. Conjunction is similar to sequence except that all the incoming event instances can initiate a new matching and thus we need to probe all the other connected ECQs via the Probe function. As such, the proposed ECQ-based processing algorithm can easily support conjunction queries of simultaneous queries, compared to conventional automata-based CEMs.

Disjunctions. The evaluation of disjunction is straightforward: it simply generates union of its inputs. It does not need to look up other ECQs nor check time constraints. Thus, SCEMon does not create any PMatch entries, resulting in no insertion of instances.

5.1.2 Insert Phase

The insert phase updates ECQ_i, reflecting the dynamic composition status affected by the event instance e_k. That is, e_k as well as the PMatch entries, newly created in the test phase, are stored into ECQ_i. (1) First, the entries are added into the PMB of e_k. (2) Then, e_k is inserted into the SIQ. (3) Finally, the insertion updates the hash entry HT(e_k.source_id) so that it points to the newly added e_k in the queue. Note that, for an incoming e_k, insertion takes place *at most once* even for the case where e_k extends multiple partial matching.

For efficient memory management, SCEMon needs to delete any obsolete stored instances from ECQs. All the ECQs in the shared network are accessed periodically to remove old instances from each SIQ and update the corresponding hash table accordingly.

5.1.3 Correctness of Incremental Query Processing

Receiving event instances, SCEMon changes the state of the shared ECQ network. It computes the matching results for multiple queries incrementally, as if multiple query processing plans are evaluated independently. Let $e_k \in E_i$ denote an instance e_k of event class E_i (for all e_k, there is a single E_i such that $e_k \in E_i$). Also, let $E_i \in Q_j$ denote that E_i is a member event class of the CEM query Q_j. Then the following theorem holds:

Theorem 1. Correctness of Incremental Processing for Sequence:
For a sequence query Q_j with event classes E_1, E_2, ..., and E_n with time condition t_cond, if there exists PMatch(e_n, Q_j) such that $e_n \in E_n$, then there exists (e_1, e_2, ..., e_{n-1}) such that

- $e_1 \in E_1$, $e_2 \in E_2$, ..., $e_{n-1} \in E_{n-1}$,
- e_i.end_ts $\preceq e_{i+1}$.start_ts, and
  e_n.end_ts - e_1.start_ts $\preceq Q_j$.t_cond for $1 \preceq i \preceq n-1$.

The complete matching result is given by (e_1, e_2, ..., e_{n-1}, e_n).

Proof.

By definition, if there exists PMatch(e_k, Q_j) such that $e_k \in E_1$, then PMatch(e_k, Q_j).t_start = e_k.start_ts.

By Lemma 1, if there exists PMatch(e_n, Q_j) such that $e_n \in E_n$, there exists PMatch(e_{n-1}, Q_j) such that $e_{n-1} \in E_{n-1}$, e_{n-1}.end_ts ≤ e_n.start_ts, (e_n.end_ts - PMatch(e_{n-1}, Q_j).t_start) ≤ Q_j.t_cond. …(1)

By induction, there exist PMatch(e_{n-2}, Q_j), ..., PMatch(e_2, Q_j), and PMatch(e_1, Q_j) such that $e_{n-2} \in E_{n-2}$, ..., $e_2 \in E_2$, $e_1 \in E_1$, and e_{n-2}.end_ts ≤ e_{n-1}.start_ts, ..., e_1.end_ts ≤ e_2.start_ts.

By definition, PMatch(e_{n-1}, Q_j).t_start = PMatch(e_{n-2}, Q_j).t_start = ... = PMatch(e_1, Q_j).t_start = e_1.start_ts.

Thus, (e_n.end_ts - e_1.start_ts) ≤ Q_j.t_cond. (By equation (1))

∴ Corresponding (e_1, e_2, ..., e_{n-1}) exists.

End of Proof

Lemma 1.

For any $E_k \in Q_j$ where $k > 1$, if there exists PMatch(e_k, Q_j) such that $e_k \in E_k$, then there exists PMatch(e_{k-1}, Q_j) such that

- $e_{k-1} \in E_{k-1}$,
- $e_{k-1}.end\_ts \leq e_k.start\_ts$, *and*
- $(e_k.end\_ts - PMatch(e_{k-1}, Q_j).t\_start) \leq Q_j.t\_cond$.

Proof.

There exists PMatch(e_k, Q_j) such that $e_k \in E_k$ for k > 1, if and only if there exists e_{k-1} such that

$e_{k-1} \in E_{k-1}$ (∵ $E_{k-1} \in$ CLink(ECQ_k, Q_j).ptr_ECQ),

e_{k-1}.end_ts ≤ e_k.start_ts (∵ e_{k-1} was already stored), and

(e_k.end_ts - PMatch(e_{k-1}, Q_j).t_start) ≤ Q_j.t_cond.

∴ Corresponding PMatch(e_{k-1}, Q_j) exists.

End of Proof

For conjunction, the algorithm probes all participating ECQs to check where the matching is complete; for disjunction it probes none. Therefore, for these patterns, correctness follows trivially.

5.1.4 Shared Processing with Instance Sharing

The processing algorithm presented in this section is designed to inherently share incoming event instances by multiple CEM queries on top of the shared ECQ network. We note that the performance improvement of SCEMon is brought mainly at the insert phase; the algorithm stores a small number of instances in ECQs. The cost of the delete operation necessary to clean memory with obsolete instances is also reduced accordingly. Thus, the costs of insert and delete operations become constant regardless of the number of queries, resulting in significant performance improvement. Next, we discuss how to further optimize the processing of SCEMon by leveraging additional sharing and redundancy reduction opportunities.

5.2 Sub-Pattern Sharing

In the basic SCEMon discussed so far, individual queries are handled separately during the test phase since CLT maintains them separately. Thus, when different queries share a partial pattern, this pattern is tested multiple times (See Figure 1 for the example of a partial pattern, i.e., **A;B** and **A&B**, in our two example queries; here it is necessary to probe ECQ[A] twice for an input instance of ECQ[B]).

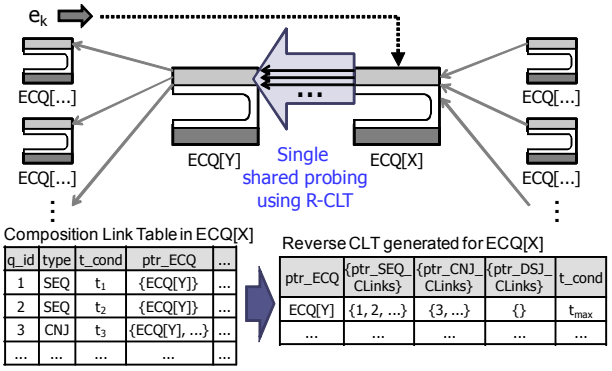


Figure 6. Sub-pattern sharing using R-CLT

Specifically, for a *binary* pattern formed by two adjoining ECQs, its former ECQ needs to be probed by the later ECQ multiple times. Thus, this redundant work can be eliminated substantially by sharing a probing result among different queries. The probing result can also be shared between the queries of different composition patterns, e.g., between sequence and conjunction queries. In addition, successive application of such binary pattern sharing can effectively support any longer sub-patterns. Note that such sub-pattern sharing opportunities, especially sharing of any binary patterns rather than longer sub-patterns, exist plentifully in large-scale CEM environments, for instance, monitoring population patterns in hot spots of a city.

Figure 6 illustrates the general case of the sub-pattern between two adjoining ECQs, i.e., ECQ[X] and ECQ[Y]. For a number of CEM queries, ECQ[Y] is specified to be probed by ECQ[X] multiple times in *any* positions of *any* composition patterns. Note that it is not identical to the case of prefix sharing techniques such as 'prefix-caching of automata states' in [16] and 'pre-fix state merging' in [17]. Unlike in previous work, instead of trying to reuse intermediate results matching entire common prefixes, we reuse partial processing results through probing as shown in the figure. Since common prefixes require complete matches from the beginning of the pattern, they are less frequently available than the partial results we are probing. Thus the proposed mechanism results in further savings by taking advantage of extended sub-pattern sharing opportunities.

5.2.1 Reverse CLT (R-CLT) and R-CLinks

To share a probing result among all relevant queries, we extend the CLT structure to include the information on the many diverse queries to share the probing result. For this purpose, we introduce a new table, called *Reverse CLT (R-CLT)*. **Reverse CLT (R-CLT)** is generated from CLT in each ECQ and contains a set of *Reverse CLinks (R-CLinks)*, which is formally defined as 5–tuple *(ptr_ECQ, {ptr_SEQ_CLinks}, {ptr_CNJ_CLinks}, {ptr_DSJ_CLinks}, t_cond)*, where

- *ptr_ECQ* is a pointer to an ECQ that needs to be probed;
- the three sets, *{ptr_SEQ_CLinks}*, *{ptr_CNJ_CLinks}*, and *{ptr_DSJ_CLinks}*, contain the pointers of CLinks including the probed ECQ in the CLT, which correspond to sequence, conjunction, and disjunction queries respectively; and
- *t_cond* is the maximum value of the time conditions among the CLinks in {ptr_SEQ_CLinks} and {ptr_CNJ_CLinks}.

R-CLT is derived easily from CLT. (a) For each distinct ptr_ECQ specified in CLinks, an R-CLink entry is created. (b) Then, the corresponding CLinks are inserted into {ptr_SEQ_CLinks},

{ptr_CNJ_CLinks}, or {ptr_DSJ_CLinks}. (c) Finally, those CLinks which do not have any pointers to other ECQs, (i.e., CLinks of disjunction or CLinks flagged as INIT in sequence), are gathered into a special R-CLink whose ptr_ECQ is null.

When sub-pattern sharing is enabled, SCEMon enumerates R-CLT instead of CLT in the test phase. For each R-CLink in R-CLT, it probes the corresponding ECQ *only once* with the largest time condition. If the Probe function returns a precedent instance that satisfies the time condition, SCEMon evaluates the extension of partial matching for each CLink specified in the sets, {ptr_SEQ_CLinks} and {ptr_CNJ_CLinks}. If it returns null, then no further evaluation is needed. For those R-CLink's whose ptr_ECQ is null, it is not necessary to probe another ECQ; thus, SCEMon simply proceeds to the insert phase.

5.2.2 Shared Processing with Sub-Pattern Sharing

The cost of the Probe function is mainly caused by searching for a precedent instance of a specified event source stored in SIQ, especially with the presence of a large number of event sources. Thus, the reuse of precedent event instances once returned by the Probe function plays a critical role in sharing benefits. This sharing is highly beneficial when there exist a large number of the CEM queries that monitor some binary patterns of common interest frequently. The cost of the probe operation during the test phase can be reduced significantly as the degree of sub-pattern sharing among CEM queries. Even for the worst case, the cost is bounded by the number of primitive event classes (or the number of ECQs), not by the number of CEM queries.

5.3 PMB Reduction

SCEMon also shares among multiple queries the partial matching information of individual event instances. This is especially useful when those queries are triggered initially by common event instances. For those cases, the PMB size can be reduced by helping share the PMatch entries across queries.

Since SCEMon investigates each PMatch entry to evaluate the possible extension of partial matching of a corresponding query, the PMB size corresponding to an event instance can influence the processing cost. As the size of PMB increases, SCEMon needs to spend more time for accessing necessary PMatch entries.

Let *Ps* denote the selection probability, meaning how many subsequent instances extend the partial matching in sequence queries. If *Ps* is small enough, PMatch entries of initial matching for the sequence queries are dominant in PMB. We note that the PMatch entries of initial matching contain redundant information for the corresponding sequence queries: all the recorded PMatch.t_start's corresponding to the same start timestamp. Moreover, the PMatch.{ptr_instance} values are null. Thus, we can drop these entries from PMB and substitute the start timestamp of the event instance for the corresponding initial matching time. By doing this, we can reduce the size of PMB approximately as a factor of *Ps*. Consequently, we can reduce the evaluation cost of partial matching extensions during the test phase significantly for small values of *Ps*.

6. PERFORMANCE ANALYSIS

The processing cost of SCEMon mainly includes five components: the cost of searching for an ECQ (C_{ECQ}), the cost of probing other ECQs to search for precedent instances (C_{Probe}), the cost to evaluate the extension of partial matching ($C_{Evaluate}$), the cost to insert an instance into an ECQ (C_{Insert}), and the cost to clean obsolete instances (C_{Clean}). Therefore, the cost CP is:

$$C_P = k_1 C_{ECQ} + k_2 C_{Probe} + k_3 C_{Evaluate} + k_4 C_{Insert} + k_5 C_{Clean}$$

The weights k_1 through k_5 are infrastructure-specific.

C_{ECQ} depends on the number of all ECQs, which is equal to the total number of primitive event classes, *Npc*. As long as a small number of primitive events are shared by a large number of CEM queries, the cost can be considered tiny compared to other terms.

C_{Probe} is a function of the average number of queries that an ECQ participates in (n_q), the average number of probing for a query (n_{pr}), and the unit cost of probing (c_{pr}).

- n_q can be computed as the degree of event class sharing D_S, (*Nq Nqec*) / *Npc*, where *Nq* is the number of queries, *Nqec* is the average number of event classes used in a query, and *Npc* is the total number of primitive event classes used in SCEMon.
- n_{pr} is estimated differently for different composition types: it is approximately 1 for sequence, *Nqec* for conjunction, 0 for disjunction.
- C_{pr} involves lookups to the index in the SIQ. This cost depends on the implementation of ECQ. In SCEMon, these lookups have logarithmic complexity O(log *Ns*) by using *STL:map* [13] for the index[3], where *Ns* is the number of event sources.

Hence, the term C_{Probe} is calculated as (D_S n_{pr} log *Ns*), which is proportional to *Nq* and log *Ns*. Yet, thanks to the sub-pattern sharing discussed in Section 5.2, this cost can be bounded by (*Npc* n_{pr} log *Ns*) since n_q is at most the number of ECQs, i.e., *Npc*.

$C_{Evalute}$ is mainly dependent on the size of the PMB of a precedent instance, since it looks up the partial matching information for a query. The average size of PMB can be approximated to n_q, which is D_S. Therefore, $C_{Evaluate}$ can be computed as O(log D_S). According to Section 5.3, this cost can be reduced by the PMB reduction by a factor of *Ps*.

C_{Insert} is a function of the probability of insertions (p_{ins}) and the unit insertion cost (c_{ins}): $C_{Insert} = p_{ins}\ c_{ins}$. Since we use *STL:deque* and *STL:map* for the queue and the index in the SIQ, c_{ins} has the logarithmic complexity O(log *Ns*) for the insertions. The probability, p_{ins}, is between 0 and 1; our experiments in Section 7 showed that in practice most input event instances are inserted into some ECQ. Hence, taking a conservative approach, p_{ins} can be approximated as 1 for all composition types (except for disjunction, for which p_{ins} is approximated as 0 since we do not store event instances for disjunction). Note that C_{Insert} is not proportional to the number of CEM queries, *Nq*. This is because of the instance sharing technique presented in Section 5.1.

Finally, C_{Clean} is a sum of the cost to check all SIQs (c_{chk}) and the cost to delete obsolete instances in them (c_{del}).

- c_{chk} is estimated as the number of ECQs, i.e., *Npc*. Note that for the conventional separate processing scheme in Figure 1-(a), it is very large, i.e., (*Nq Ns*).
- c_{del} is proportional to the average number of stored instances and the unit cost of deletions. Due to the instance sharing, SCEMon stores incoming event instances at most once regardless of the number of queries. Thus, it is only proportional to *Ns*, not to *Nq*.

[3] Refer to http://www.cplusplus.com/reference/stl/ for the complexity of STL containers.

The storage cost of SCEMon, C_S, can be estimated as follows: Let the rate of input event instances incoming to SCEMon is ($Ns\ r_e$), where r_e is the average rate of event generation from a source. The length, T_W, of the time window for storing events is determined based on the time constraints of the registered CEM queries. Let c_s is the size of the memory consumed by each instance, then C_S can be calculated as ($p_{ins}\ Ns\ r_e\ T_W\ c_s$). As we discussed above, p_{ins} is 0 for disjunction and approximately 1 for all the other types with a large number of the queries. Note that, thanks to the ECQ sharing, C_S is proportional to Ns, but not to Nq.

7. PERFORMANCE EVALUATION

In this section, we evaluate the performance improvement of SCEMon over conventional separate processing (SP) schemes in large-scale CEM environments. The experiments were run on Intel Core 2 Quad Yorkfield Q9550 CPU (2.83GHz) and 8 GB RAM. The machine was running Debian Linux 2.6.18 64-bit.

Workloads. For the evaluation, we generate synthetic input event instances and CEM queries. Input event instances are generated randomly in the form of a tuple (*source_id*, *event_class*, *start_ts*, *end_ts*), as described in Section 3.1. CEM queries are generated based on the sequential query templates given below.

CEM query template

PATTERN	*Sequence*
WITH	*A, B, C*
WHERE	*[symbol]*
WITHIN	*[200-240] mins*

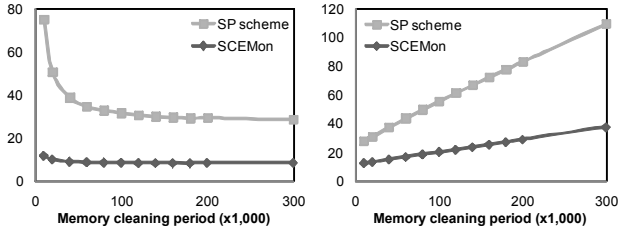
The primitive event classes in the WITH clause, i.e., A, B and C, are randomly selected among a given set of primitive event classes. The time condition in the WITHIN clause is also randomly specified in the value range. In this work, we focus on the efficient processing of composition patterns so that any additional predicate conditions in the WHERE clause are not included.

As default setting, we use 1K sequential queries and 1K event sources assuming 50 primitive event classes. Upon the start of experimental runs, queries are registered into the system. Also, generated event instances are loaded into the main memory and pulled into the system at the maximum rate it could accept.

Evaluation metrics. We measured the performance in terms of processing and storage costs. First, for the processing cost, we use the *unit processing time* as an evaluation metric; it is defined as $t_{elapsed} / N_{total\_event}$, where N_{total_event} is the total number of the input event instances, and $t_{elapsed}$ is the total elapsed processing time, not including time to deliver the output. Second, for the storage cost, we count the average number of event instances stored in ECQs; note that the amount of stored instances indicates the storage consumption in CEM processing. The number of the stored instances is counted before and after each memory cleaning, and the average is computed over 15 cleanings after warm-up.

Comparing techniques. For comparison, we have implemented a conventional SP scheme based on the work done by Wu et al. [1]. The SP scheme illustrated in Section 3.2 is implemented in C++ using STL [13]. The SP scheme is fairly implemented by performance comparisons with publically available CEM implementations, Cayuga [14] and Esper [15] (See Section 7.6 for the detailed discussion).

To closely investigate the effectiveness of the shared processing techniques employed in this work, we use three different SCEMon implementations: (1) with event instance sharing (Section 5.1), (2) with instance sharing and sub-pattern sharing (Section 5.2), (3) with instance sharing, sub-pattern sharing and PMB reduction (Section 5.3). For fair comparison, SCEMon is also implemented in C++ using STL [13].



(a) Cleaning time (ms) (b) Memory consumption (MB)

Figure 7. Effect of memory cleaning period (N_c)

Memory cleaning period. The system performs periodic cleaning, where it deletes obsolete instances older than the maximum time window of all relevant queries. Cleaning obsolete activity instances is critical for in-memory processing of CEM queries. It has significant impact on the processing as well as storage cost. It is especially important for the SP scheme since the memory space would quickly be exhausted due to the numerous per-object stacks.

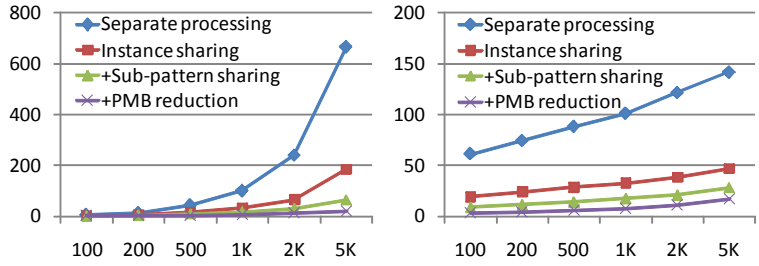
We examine the effect of the cleaning on the processing and the storage cost. Figure 7 shows the results while we perform the cleaning every Nc updates of event instances. We observe that there exists a trade-off between the cleaning time and memory usage. The cleaning time of SCEMon is not much affected by the cleaning period. The time-ordered instances stored in shared ECQs make the cleaning process highly efficient since the number of the queues to be scanned is relatively small and the set of obsolete instances is easily identified. On the other hand, for the SP scheme, a huge number of the per-source stacks should be scanned to delete obsolete event instances as well as empty stacks. The figure presents that too frequent cleaning causes unnecessary scanning with rarely effective deletions, resulting in the excessive cleaning time. The figure also shows that it is hardly beneficial to defer cleaning beyond some extent, since the cleaning time decreases marginally. Meanwhile, deferred cleaning substantially increases the memory consumption to hold more obsolete instances and unused stacks. In the experiments below, we set the base cleaning period to 100K where the cleaning time starts to be saturated while the memory consumption increases linearly.

7.1 Processing Scalability

This section shows the scalability of the SP scheme and SCEMon in large-scale CEM environments. Figure 8 presents the unit processing time in microsecond.

Scalability with the number of queries (Nq). We increase the number of queries from 100 to 5K. As shown in Figure 8-(a), the unit processing times of the SP scheme drastically increase for larger Nq's compared to SCEMon. In contrast, the suit of shared processing techniques based on ECQs enables SCEMon to significantly reduce the processing time, especially at larger Nq's With 5K simultaneous queries, for instance, it shows about 32 times better performance than the SP scheme.

Scalability with the number of sources (Ns). Figure 8-(b) shows the performance with increasing the number of sources up to 5K. The result illustrates that the processing times of the SP scheme and SCEMon increase proportionally to (log Ns); note that X-axis



(a) Increasing *Nq* (b) Increasing *Ns*

Figure 8. Unit processing times (μsec) of SP scheme vs. SCEMon

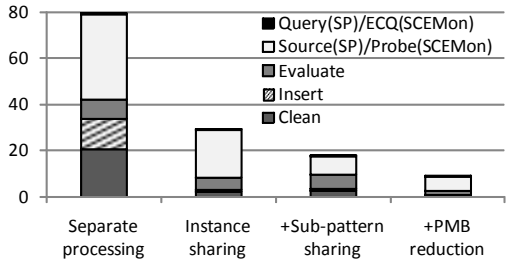
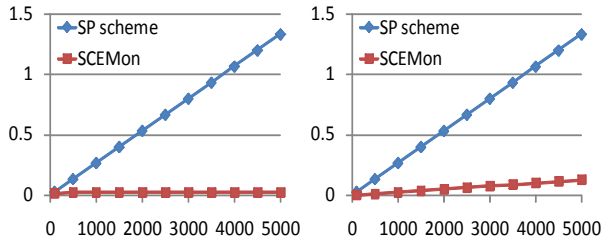
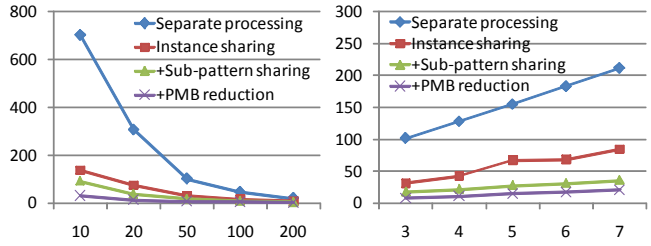


Figure 9. Performance breakdown of conventional SP scheme vs. SCEMon



(a) Increasing *Nq* (b) Increasing *Ns*

Figure 10. Number of stored event instances (M)



(a) Increasing *Npc* (b) Increasing *Nstep*

Figure 11. Unit processing times (μsec)

of the graph has a logarithmic scale. Yet, it also shows that SCEMon deals with a large number of event sources much more efficiently than the SP scheme.

7.2 Effectiveness of SCEMon Techniques

In this section, we closely investigate where the significant performance gain comes from. Figure 9 shows the breakdowns on the unit processing times of the SP scheme and SCEMon in default setting, i.e., 1K queries and 1K sources. The figure shows that almost a half of the processing time in the SP scheme is spent to search for a proper data structure corresponding to a source of an incoming instance (labeled "*Source(SP)*") due to the overhead of managing many event sources. The time for deleting obsolete event instances in separate data structures ("*Clean*") takes the second place. Yet, the time for evaluating state transitions for individual instances ("*Evaluate*") is shown to be relatively small.

In SCEMon with the instance sharing, the times for storing incoming event instances and deleting obsolete instances ("*Insert*" and "*Clean*") are reduced significantly, compared to those of the SP scheme. The performance improvement conforms to our expectations and analysis we performed while designing the shared storage of ECQs. However, the times for probing other ECQs and evaluating the extension of partial matching ("*Probe(SCEMon)*" and "*Evaluate*") are not reduced sufficiently.

For SCEMon with additional sub-pattern sharing, the time for probing other ECQs ("*Probe(SCEMon)*") is reduced. This shows that the probing operation is optimized by using the R-CLT and happens only once for each target ECQ. Moreover, we simplify the probing operation by omitting the step for checking up the initial matching time in the PMB and defer it to the evaluation step. Thus, the time for evaluating the extension of partial matching ("*Evaluate*") increases slightly. This result also conforms to the analysis in Section 6.

Finally, for SCEMon with full sharing techniques, the time for evaluating the extension of partial matching ("*Evaluate*") is reduced, since the PMB reduction technique is designed to reduce the PMB access time during the evaluation. In addition, event instances stored in ECQs are associated with smaller PMBs, resulting in the time for deleting obsolete event instances ("*Clean*") is reduced accordingly.

7.3 Storage Scalability

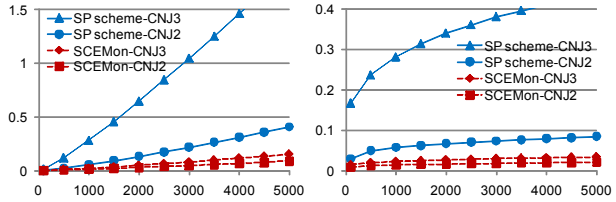
We also evaluate the storage performance of SCEMon compared with the SP scheme. Figure 10-(a) shows the average number of event instances stored in the data structure as the number of queries (*Nq*) increases. The SP scheme stores event instances in each NFA separately and does not share them at all. This leads to the redundant storage consumption proportional to the number of queries. For SCEMon, however, the numbers of stored instances are saturated for larger *Nq*'s. This is the result of instance sharing by which only a single copy of individual incoming instances is stored regardless of a multitude of simultaneous queries.

Figure 10-(b) demonstrates the remarkable storage efficiency of SCEMon over the SP scheme for large numbers of sources. In fact, the storage costs of SCEMon also increase linearly with the number of sources (*Ns*) due to the higher rates of incoming event instances for larger *Ns*'s. However, SCEMon keeps the storage costs much lower, almost 10% in the setting, compared to the SP scheme by virtue of instance sharing.

7.4 Performance Characteristics of SCEMon with Other Attributes

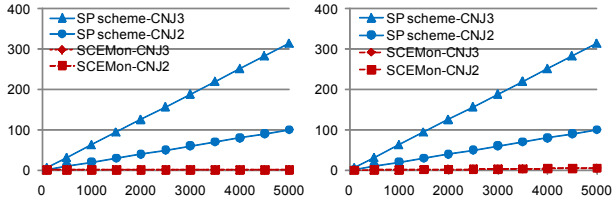
We have also performed the performance evaluation of SCEMon and the SP scheme with varying numbers of primitive event classes and sequence steps in the template sequence query.

Performance with the number of primitive event classes (*Npc*). We investigate how the performance of SCEMon varies due to the degree of event class sharing. We change the number of primitive event classes from 10 to 200 so that an identical number of simultaneous queries share them in different levels of sharing.



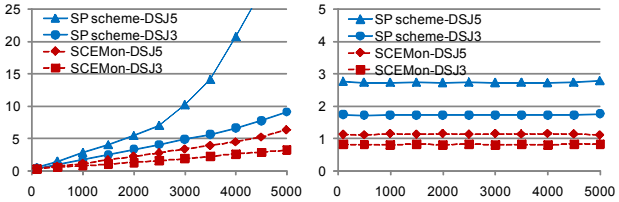
(a) Increasing *Nq* (b) Increasing *Ns*

Figure 12. Unit processing time (ms) with conjunction



(a) Increasing *Nq* (b) Increasing *Ns*

Figure 13. Number of stored instances (M) with conjunction



(a) Increasing *Nq* (b) Increasing *Ns*

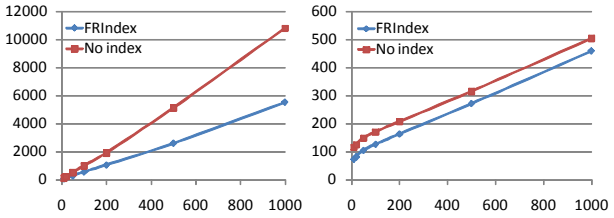
Figure 14. Unit processing time (ms) with disjunction

(See Figure 11-(a).) When *Npc* is 10, i.e., we only use 10 primitive event classes in the setting, an event class should be shared by large numbers of simultaneous queries. On the other hand, when *Npc* is 200, the degree of sharing decreases. Due to the proposed event-centric sharing approach, the performance of SCEMon becomes much better than that of the SP scheme with smaller *Npc*'s. As expected, with larger *Npc*'s, the performance gap between the SP scheme and SCEMon gets smaller, but SCEMon still performs better than the SP scheme.

Performance with the number of sequence steps (*Nstep*). The number of sequence steps also influences the degree of event-class sharing, since each query can contain more numbers of primitive event classes with larger *Nstep*'s. Figure 11-(b) demonstrates that the unit processing times of the SP scheme and SCEMon increase as *Nstep* increases, similar to the impact of increasing *Nq*. However, the impact is not as significant as that of increasing *Nq*, since a large portion of incoming event instances fails to extend partial matching and is discarded without any further evaluation.

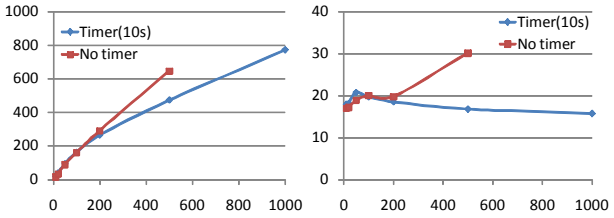
7.5 Performance Evaluation with Conjunction and Disjunction

We have also evaluated the performance of SCEMon with conjunction and disjunction queries. Figure 12 and 13 show that SCEMon outperforms the SP scheme significantly for conjunction for increasing *Nq*'s and *Ns*'s. This is because the SP scheme instantiates all the permutated sequences of the participating event classes in an NFA, which results in a huge number of NFA states loaded in the system. Our results show the sharp increase in the processing cost for the conjunction queries of three event classes, i.e., CNJ3. On the other hand, Figure 14 shows the results of disjunction. As a larger number of NFAs are instantiated (see



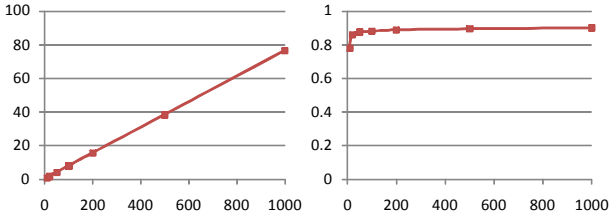
(a) Increasing *Nq* (b) Increasing *Ns*

Figure 15. Unit processing time of Cayuga: *FRIndex* is a primary optimization technique suggested in Cayuga that provides optimized access to attributes among different queries. We have built the index on the event class ids so that the effectiveness of the index gets significant with increasing *Nq*.



(a) Increasing *Nq* (b) Increasing *Ns*

Figure 16. Unit processing time of Esper: *Timer* is a kind of stopwatch. If associated pattern expressions do not turn true within the specified time period, they are stopped and permanently false. The setting of no timer with 1000 sources causes out-of-memory exception.



(a) Increasing *Nq* (b) Increasing *Ns*

Figure 17. Unit processing time of our SP scheme implementation

disjunction of five events, i.e., DSJ5, compared to DSJ3), the processing time of the SP scheme gets longer due to the lack of sharing. In Figure 14-(b), note that the processing cost of disjunction queries remains almost constant with a fixed number of queries. This is because the incoming event instances are not stored and looked up for disjunction at all.

7.6 Performance of Publically Available Implementations

To understand the performance of conventional SP schemes with large numbers of queries and sources, we have conducted the performance evaluation using publically available implementations. First, we retrieved the Cayuga implementation from [14] and ran it on a 32-bit Windows XP machine of Intel Core 2 CPU (2.13GHz) and 3.50GB RAM. The Linux configuration of the Cayuga implementation is not correctly supported so that we use Windows instead.

The evaluation shows that Cayuga is not efficient for large numbers of queries and sources. We measured the unit processing time in microsecond as we increase the number of queries (*Nq*) and the number of sources (*Ns*) from 10 to 1K respectively. (We used 10 as the default values of *Nq* and *Ns* in each experiment. It

could not support 1K queries and 1K sources at the same time.) The results shown in Figure 15 demonstrate that the unit processing time of Cayuga increases linear proportionally to *Nq* and *Ns*. We further challenged Cayuga with larger scales and observed that the implementation consumes too much memory to run in the experimental setting.

Figure 16 presents the performance evaluation of another publically available CEM implementation, *Esper*. We retrieved the implementation from [15] and performed evaluation in the same setting as Cayuga. It shows better performance than Cayuga, yet the processing times are larger than those of our SP scheme implementation (See Figure 17 that presents the processing times of our implementation in the same setting). This is because ours is much specialized to the core processing of CEM while Cayuga and Esper are involved in additional processing such as dealing with XML input. Based on these results, we note that our implementation of conventional SP schemes is reasonable for fair comparison to SCEMon.

8. CONCLUSION and DISCUSSIONS

In this paper, we focused on the scalability issues when large numbers of CEM queries and event sources exist in upcoming CEM environments. To address these challenges effectively, we take an *event-centric sharing approach* rather than conventional query/source-separate processing approaches. ECQ is a novel data structure designed to facilitate efficient evaluations of multiple queries over very large volumes of event streams from numerous event sources. ECQs are composable to build a single shared network within which multiple queries are efficiently evaluated. We developed a set of the shared processing techniques on top of the ECQ network. Our evaluation showed that our approach outperforms the conventional approaches in large-scale CEM environments.

While we did not discuss it in this paper, SCEMon can easily support negation and Kleene closure; these require slight modifications in the *Probe* function. For negation, the condition testing, i.e., if any event instance exists and satisfies the time constraints, should be inversed; for Kleene closure, the Probe function checks the number of the stored precedent instances satisfying the time and value constraints in the SIQ. In addition, the test phases for sequence and conjunction also need to be modified. All ECQs designated to generate outputs need to look up the ECQs participating for negation and Kleene closure, before it generates output results. Since these modifications are straightforward, we do not present the algorithms in this paper.

Another aspect of CEM processing not discussed in this paper is nested query monitoring. A nested query consists of different patterns, e.g., a conjunction of sequences, a sequence of conjunctions and disjunctions, etc. Supporting such nested queries has been considered as an important issue in composite event processing research, yet rarely addressed. SCEMon can have virtual primitive event classes for individual composition patterns in nested queries. We register each pattern as a CEM query and define a virtual primitive event class (VPEC) for the results of the query. SCEMon feedbacks the results redefined as the instances of the VPEC as input. In other words, SCEMon constructs the shared network of ECQs that are responsible for real primitive event classes and VPEC's. Treating nested patterns as primitive event classes, SCEMon can evaluate complicatedly nested queries effectively within the shared network of ECQs.

9. ACKNOWLEDGMENTS

This research was supported by Future-based Technology Development Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (2010-0020729).

10. REFERENCES

[1] Wu, E., Diao, Y. and Rizvi, S. 2006. High-performance complex event processing over streams. In Proc. of SIGMOD.

[2] Akdere, M., Çetintemel, U. and Tatbul, N. 2008. Plan-based Complex Event Detection across Distributed Sources. In Proc. of VLDB.

[3] Mei, Y. and Madden, S. 2009. ZStream: A Cost-based Query Processor for Adaptively Detecting Composite Events. In Proc. of SIGMOD.

[4] Yang, D., Rundensteiner , E. and Ward, M. 2009. A Shared Execution Strategy for Multiple Pattern Mining Requests over Streaming Data. In Proc. of VLDB.

[5] Demers, A., Gehrke, J., Panda, B., Riedewald, M., Sharma, V. and White, W. 2007. Cayuga: A general purpose event monitoring system. In Proc. of CIDR.

[6] Gatziu, S. and Dittrich, K. 1994. Events in an active object-oriented database, In Workshop on Rules in Database Systems.

[7] Chakravarthy, S., Krishnaprasad, V., Anwar, E. and Kim, S. 1994. Composite events for active databases: Semantics, contexts and detection. In Proc. of VLDB.

[8] Urban, S., Biswas, I. and Dietrich, S. 2006. Filtering features for a composite event definition language. In Proc. of SAINT.

[9] Hinze, A. 2003. Efficient filtering of composite events, In Proc. of BNCD.

[10] Elkhalifa, L., Adaikkalavan, R. and Chakravarthy, S. 2005. InfoFilter: A system for expressive pattern specification and detection over text streams. In Proc. of SAC.

[11] Agrawal, J., Diao, Y., Gyllstrom, D. and Immerman, N. 2008 Efficient pattern matching over event streams. In Proc. of SIGMOD.

[12] Ananthanarayanan, G., Haridasan, M., Mohomed, I., Terry, D. and Thekkath, C. 2009. StarTrack: A framework for enabling track-based applications. In Proc. of MobiSys.

[13] Standard Template Library, http://www.cplusplus.com/reference/stl/

[14] Cayuga source code, http://sourceforge.net/projects/cayuga/

[15] Esper official site, http://esper.codehaus.org

[16] Candan, K., Hsiung, W., Chen, S., Tatemura, J. and Agrawal, D. 2006. AFilter: Adaptable XML Filtering with Prefix-Caching and Suffix-Clustering. In Proc. of VLDB.

[17] Hong, M., Riedewald, M., Koch, C., Gehrke, J. and Demers, A. 2009. Rule-based multi-query optimization. In Proc. of EDBT.