Active Complex Event Processing over Event Streams*

Di Wang
Worcester Polytechnic Institute
Worcester, MA, USA
wangdi@cs.wpi.edu

Elke A. Rundensteiner
Worcester Polytechnic Institute
Worcester, MA, USA
rundenst@cs.wpi.edu

Richard T. Ellison III
UMass Medical School
Worcester, MA, USA
richard.ellison@umassmemorial.org

ABSTRACT

State-of-the-art Complex Event Processing technology (CEP), while effective for pattern query execution, is limited in its capability of reacting to opportunities and risks detected by pattern queries. Especially reactions that affect the query results in turn have not been addressed in the literature. We propose to tackle these unsolved problems by embedding active rule support within the CEP engine, henceforth called Active CEP (ACEP). Active rules in ACEP allow us to specify a pattern query's dynamic condition and real-time actions. The technical challenge is to handle interactions between queries and reactions to queries in the high-volume stream execution. We hence introduce a novel stream-oriented transactional model along with a family of stream transaction scheduling algorithms that ensure the correctness of concurrent stream execution. We demonstrate the power of ACEP technology by applying it to the development of a healthcare system being deployed in UMass Medical School hospital. Through extensive performance experiments using real data streams, we show that our unique Low-Water-Mark stream transaction scheduler, customized for streaming environments, successfully achieves near-real-time system responsiveness and gives orders-of-magnitude better throughput than our alternative schedulers.

1. INTRODUCTION

Complex patterns of events often capture exceptions, threats or opportunities occurring across application space and time. Complex Event Processing (CEP) technology has thus increasingly gained popularity for efficiently detecting such event patterns in real-time [1, 8, 12, 16, 17]. However, to allow CEP technology to be an end-to-end solution, beyond monitoring the world via pattern queries, we also need to *react* to the risks and opportunities detected by pattern queries in real-time. We now illustrate this need using a healthcare application as a representative example.

*This work is supported in part by NSF grant IIS-1018443 and UMMS-WPI CCTS Collaborative grant.

Proceedings of the VLDB Endowment, Vol. 4, No. 10

Motivating Application. Healthcare-associated infections hit 1.7 million people a year in the United States, causing an estimated 99,000 deaths [18]. The HyReminder system is a hospital infection control system developed by WPI and UMass Medical School to continuously track healthcare workers (HCWs) for hygiene compliance (for example sanitizing hands and wearing masks), and to remind HCWs to perform hygiene precautions - thus preventing the spread of infections [11]. As shown in Figure 1, every HCW wears a RFID badge which has a three-color light for indicating his (hygiene) status: "safe", "warning" or "violation". Pattern queries continuously monitor each HCW's behaviors observed by sensors. The status of each HCW is continuously changed upon detecting certain event patterns. For example, a detected hygiene violation pattern will change the HCW's status to "violation" (henceforth his badge light). In order to urge a HCW to perform precautions immediately upon the detected hygiene violation, it is critical for us to support such *real-time reactions* for the pattern queries. Furthermore, the status of each HCW is also used as a condition by pattern queries, e.g., a pattern query may only monitor HCWs in "safe" status. Clearly the reactions for queries that update HCWs' status will *affect query results in turn*. It is absolutely vital to control such *concurrent* updates and accesses so as to assure the correct execution logic. Otherwise we risk for a potentially highly contagious disease to be transmitted to vulnerable patients - thus increasing patient suffering and even causing deaths.

Further in Appendix A we show that the requirements derived above are prevalent across applications ranging from algorithmic trading to fraud detection. Unfortunately, current CEP systems support "read-only" query processing. Pattern queries only contain *in-place* conditions, i.e., the query qualification is either based on the attributes of events matched in the input stream [8, 12, 16, 17] or on static information pre-loaded once yet not updated during query execu-

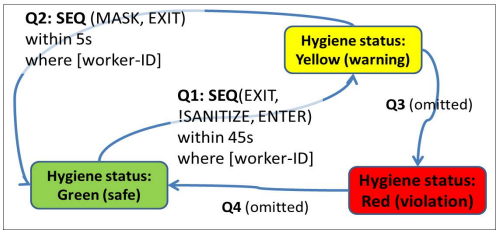

Figure 1: Representative hand hygiene logic. If the HCW is in the *start-status* node and his behavior matches the *pattern-query* annotating an outgoing edge, then change his status to the *end-status* node.

tion [1, 4, 15]. Moreover, reactions for event detections are limited to "side-effect-free" actions like sending a message or logging events that do not affect the event detection in turn [22]. Supporting concurrent update actions within the continuous execution of pattern queries over streams, especially actions that directly affect the pattern query results themselves, is an open problem.

Supporting Active Rules. We propose to tackle this unsolved problem by embedding *active rule* support within the complex event processing paradigm, henceforth called Active CEP (ACEP). Active rules in ACEP allow us to specify a pattern query's dynamic condition and real-time actions that in turn may affect the query results.

A critical technical challenge is to handle the *real-time mutual effects* between queries and reactions for queries in the high-volume stream execution. In ACEP we abstract such effects as *interactions among continuous queries and active rules*. Apparently, state-of-the-art data stream systems [4, 5, 7, 8] and CEP engines [12, 16, 17], which commonly use the *push-based* execution paradigm, have not addressed this challenge. In fact, as we demonstrate via real-world examples in Section 2, using the push-based execution for interactions among continuous queries and active rules leads to a variety of anomalies and thus erroneous results.

Introducing Stream Transactions. A common approach to deal with interactions between concurrent accesses and updates is to enforce concurrency control. However, existing concurrency control schedulers [2, 19] are based on the notion of a "database transaction" - the execution of a finite sequence of *one-time* data manipulation operations on conventional stored data sets [2]. While in our stream environments pattern queries are *continuously* executed on potentially infinite data streams. This implies that we cannot "finish" the current query as a finite-scoped operation before processing another query. In short, the concept of a transaction has not been established for stream processing. In this work, we fill this void by introducing the notion of a *transaction in the stream context*.

Furthermore, we design transactional pattern query processing to deal with interactions among continous queries and active rules. This processing is especially challenging because concurrency control poses strict time-based constraints, while our algorithms have to work for high-volume streams yet achieve *near-real-time responsiveness*.

Contributions. ***I.*** We propose the first model of integrating active rules into a stream processing system, called ACEP model, which significantly extends the state-of-the-art CEP model to meet the needs of reacting in real-time by affecting the physical or virtual world. (Section 3).

II. To characterize the interactions among continuous queries and active rules, we define the notion of correctness for stream-centric execution given concurrent accesses and updates. Based on this notion, we introduce the stream transaction model along with stream-specific ACID propositions. To the best of our knowledge, our work is the first at introducing the transaction concept into the stream processing context. (Section 4).

III. Our model empowers us to leverage classical concurrency control approaches originally designed for static databases to now solve the novel stream transaction scheduling problem. Our Strict-Two-Phase-Locking (S2PL) scheduler successfully applies a pessimistic concurrency control mechanism to the ACEP context. However, S2PL incurs a large synchronization delay due to its rigorous order preserving. Hence we provide a unique scheduler called Low-Water-Mark (LWM) which is customized to our stream context to maximize concurrent execution without compromising correctness. (Section 5).

IV. We implement the proposed techniques within a HP CHAOS CEP engine and conduct comprehensive experimental studies using real data streams. We show that LWM achieves orders-of-magnitude better throughput in high-volume workload compared to S2PL (Section 6).

2. STREAM CONCURRENCY PROBLEM: EXAMPLES

In the commonly-employed *push-based* stream execution paradigm [4, 7, 8, 12, 16, 17], new events are evaluated by continuous queries immediately upon event arrival without any safeguard mechanism to synchronize the concurrent accesses and updates during stream execution. Since continuous queries may vary in their complexities and event consumption, they may exhibit different processing delays. As a result, events with different timestamps tend to co-exist and be simultaneously executed in the system. Using this push-based execution for pattern queries and reactions for queries (represented as active rules) may raise unexpected *anomalies*. We illustrate the problems with three examples drawn from the HyReminder application[1]. Suppose pattern query Q1 continuously detects the event sequence `(EXIT, !SANITIZE, ENTER) within 45 seconds` for HCWs whose in "safe" status when his `ENTER` event occurs. The checking of a HCW's status is abstracted as a `Read` operation on the shared table storing his current status. Suppose an active rule R1 concurrently monitors the output of Q1: once Q1 produces a match for a specific HCW, R1 changes the HCW's status into "warning". This action is abstracted as a `Write` operation on the shared table. Suppose another query Q2 is also executed. Q2 detects the event sequence `(MASK,EXIT) within 5 sec` for HCWs in "warning" status when his `EXIT` event occurs. Figure 2 depicts the following examples respectively.

EXAMPLE 1. ***Correct Query and Rule Processing.*** *Let us consider processing an event sub-stream* $\{exit^2, enter^{10}$ *and* $exit^{15}\}$ *with the superscript denoting the event's application timestamp. Suppose before application time 10 the HCW was in the "safe" status and* $enter^{10}$ *leads to a match of Q1. Consequently R1 is triggered and updates the HCW's status to "warning". Later Q2 consumes the next input event* $exit^{15}$ *and recognizes that this is an event for a HCW in "warning" status (by accessing the shared table). The above query output and the value of the shared table are both as expected conforming to the desired application semantics.*

EXAMPLE 2. ***Read-too-late Anomaly.*** *Let us process the same event stream as in Example 1 using the push-based execution model. Suppose Q1 evaluates* $enter^{10}$ *first. Then when Q2 evaluates* $exit^2$*, R1 has already updated the HCW's status into "warning". So Q2 would read the HCW status as "warning", though intuitively at application time 2 the status should have been "safe". Subsequently the query processing result of Q2 is "incorrect"- not matching the semantics required by the application. The problem is that Q2 reads the shared table "too late". As depicted in Figure 2(b), the*

[1]We assume that the discussion below is regarding the same individual healthcare worker (HCW) and that the status of a HCW is independent from that of others.

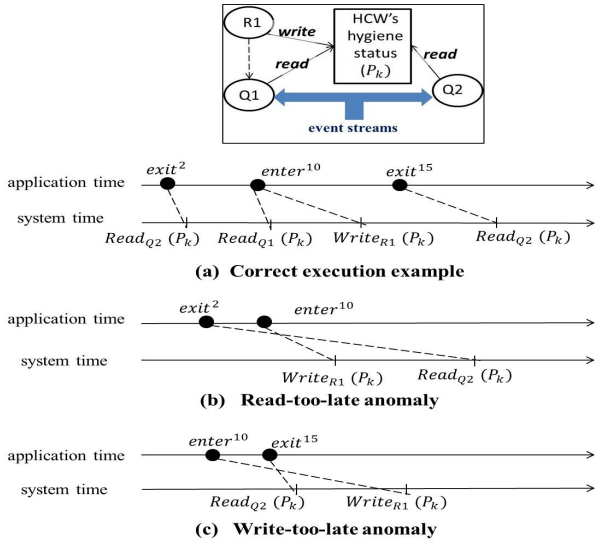

Figure 2: Stream concurrency examples. Input events are consumed in their *application time*; Read and Write are executed by the system with *system time*. Dashed line represents that the input event triggers a corresponding operation. E.g., $Read_{Q1}(P_k)$ denotes the Read performed by query Q1 on shared table P_k.

dashed lines for $Write_{R1}(P_k)$ and for $Read_{Q2}(P_k)$ cross, indicating the conflict between these two operations.

EXAMPLE 3. ***Write-too-late Anomaly.*** *Considering the same Q1, Q2 and R1 as above, now suppose Q2 evaluates $exit^{15}$ first. Thereafter Q1 evaluates $enter^{10}$, and then R1 updates HCW's status into "warning". In this case Q2 should have read the status "warning" because after application time 10 the HCW's status should become "warning". However Q2 reads some other value instead. The problem is that R1 writes "too late": the shared table has already been read by Q2 that theoretically should have executed later. As depicted in Figure 2(c), the dashed lines for $Write_{R1}$ and for $Read_{Q2}$ cross - this time in the opposite order.*

It is important to note that the above identified problems of stream concurrency are general, and would indeed equally arise in alternate stream-centric computational models, other than the active rule model employed above, that express the reactions to pattern queries (see Appendix C).

3. THE ACEP MODEL

To support the specification of actions for pattern queries, we design an Active CEP model that integrates active rules into the CEP context.

Event Instances and Types. The input to the ACEP system is a potentially infinite *event stream* that contains all events of interest. Each event instance (e.g., e_i) represents an instantaneous occurrence of interest. Each event instance has two time-stamps, *application time* and *system time* [21]. The application time for e_i refers to the discrete moment of the occurrence of e_i assigned by the event source, denoted as $e_i.ts$. While the system time of an event instance is assigned by the ACEP engine using the system wall-clock time, denoted as $e_i.sts$. Similar event instances can be grouped into an *event type*. Event types are distinguished by event type names (e.g., EXIT). See Appendix B for more preliminaries of event based systems.

Shared Store. In real-world applications, raw event data is typically augmented with semantically richer information. Semantic information regarding the application hence needs to be maintained in the system, referred to as *shared store* through this paper. For illustration, we assume the shared store is organized using the relational model. A table in the shared store can be either *static*, namely the knowledge is loaded once and not updated throughout the system execution (e.g., the mapping between RFID to HCW-ID), or *dynamic*, namely the information can be changed over time (e.g., a HCW's current hygiene status). In ACEP, a shared store thus may be both readable and updatable by multiple queries and active rules. We abstract all data processed in the ACEP system as *ACEP system state*.

DEFINITION 1. *Let I be the domain of **input event stream**. If i is an input sub-stream in I, then $i =< e_1, e_2, ..., e_n >$ where e_i is an event instance. Let O be the domain of **output event streams**. If o is an output stream in O, then $o = \{ce_1, ..., ce_h\}$ where ce_i is a composite event output by a pattern query. Let P be the domain of **shared tables**. If p is a shared table in P, then $p = \{t_1, t_2, ..., t_m\}$ where t_i is a shared tuple within the system. All data in the ACEP system, including events and shared tuples, together constitute the **ACEP system state**.*

ACEP Query. We now consider basic operations on the ACEP system state prevalent in any ACEP application. For operations over an event stream, we focus on the *dequeue* operation, i.e., to consume the first available event from the head of a stream[2], and the *enqueue* operation, i.e., to append an event to the end of a stream. These queue-based operations are common across most CEP and stream systems [8, 12, 14, 17]. A CEP query consumes the input stream, executes specified query semantics and then appends the result events to the output stream (if any). Formally we define a CEP query as below.

DEFINITION 2. *Let Φ be the domain of **dequeue** operations, Ψ be the domain of **enqueue** operations. A **CEP query**, q, is defined as a function that takes as argument an instance of Ψ on the input stream, i.e., an arrival of input event, and returns an instance of Φ on the input stream and zero or more instances of Ψ on the output stream. That is, $q : I \times \Psi \rightarrow \{I \times \Phi\} \times \{O \times \Psi\}$.*

Operations over the shared store are *Read* and *Write*. An ACEP query can be viewed as a combination of shared store accesses (e.g., to check the HCW's hygiene status) and a CEP query (e.g., to detect the pattern `SEQ(EXIT, !SANITIZE, ENTER)`). Namely, an ACEP query can defined as below.

DEFINITION 3. *Let Rd be the domain of Read operations and Wr be the domain of Write operations on shared tables. An **ACEP query**, q', can be defined as a function that: $q' : I \times \Psi \rightarrow \{I \times \Phi\} \times \{P \times Rd\} \times \{O \times \Psi\}$.*

We further abstract a set of one or more operations, including both operations on event streams and on the shared store, as *ACEP system change*, as formally defined below.

DEFINITION 4. *Let Δ be the domain of **ACEP system changes**, if δ is an ACEP system change in Δ, then $\delta \in \{\Phi, \Psi, Rd, Wr\}$.*

[2]Here we mean multi-reader dequeuing (for multi-query).

ACEP Rule. As widely recognized [6, 20], active rules have intricate run-time semantics even in static databases. We are the first to explore active rules in the streaming context and thus focus on the core features. An ACEP active rule is **triggered** by the output of an ACEP query. The **condition** of an active rule is a logical test that, if evaluates to true, causes the action of the active rule to be carried out. Similar to the qualification of a pattern query (Appendix B), the logical test can be based on the attributes of the triggering event or on the content of the shared store. The **action** of an active rule supported in the current ACEP model is the Write operation on the shared store. Such active rules are of great use and ubiquitous in ACEP applications as demonstrated by our motivating examples. Namely, pattern queries read the shared store, while the active rules may write the shared store, both in real-time. Hence the newly updated value of the shared store will in turn affect subsequent query execution. Our model includes a *rule type* that describes the definition of a rule, denoted by upper-case letters (e.g., "R_A"), and a *rule instance* that corresponds to an instantiation of a rule type, denoted by lower-case letters (e.g., "r_{A_i}"). Formally,

DEFINITION 5. *An **ACEP active rule**, r, is a function that takes as argument an ACEP query output and returns a boolean value (indicating whether the rule is triggered or not) and a set of operations on the shared store. That is,* $r : O \times \Psi \rightarrow \{true, false\} \times \{P \times \{Rd, Wr\}\}$.

3.1 ACEP Active Rule Specification

For illustration, we adopt a declarative active rule language (presented in Appendix B) implementing the model described above based on the commonly used ECA format [20, 23]. For ACEP queries, we are interested in *sequential pattern queries* commonly supported in most CEP systems [8, 12, 17], namely the SEQ operator that specifies a particular order in which the events of interest must occur. We now explain the clauses using examples drawn from our motivating healthcare application. Assume each event has the schema *(timestamp, HCW-ID, behavior, location)*. Also assume the current hygiene status of every HCW is stored in a shared table named *workerStatus*, with the schema *workerStatus:(workerID, status)*. In this example, the Read operation (resp. Write) on HCW status is expressed as a SELECT clause (resp. UPDATE) supported by standard SQL.

```
CREATE QUERY Q1 ON estream
PATTERN SEQ(EXIT, !SANITIZE, ENTER)
WHERE [HCW-ID] AND
EXIT.location != ENTER.location AND
'safe'=(SELECT status FROM workerStatus
  WHERE workerID=ENTER.HCW-ID)
WITHIN 45 sec
RETURN ENTER.HCW-ID, ENTER.location

CREATE RULE R1
ON OUTPUT Q1
REFERENCING NEW AS newEvent
FOR EACH EVENT
BEGIN
  UPDATE workerStatus SET status = 'warning'
  WHERE workerID = newEvent.HCW-ID
END
```

In pattern query Q1, the SEQ operator `SEQ(EXIT, !SANITIZE, ENTER)` together with the window constraint `WITHIN 45 sec` detects a specific HCW behavior pattern. The attribute enclosed in the square bracket, i.e., `[HCW-ID]` stands for the equivalence test on this common attribute across an entire event sequence. The qualification `'safe'=(SELECT ...)` specifies that such pattern detection should only be applied to the HCW who is in "safe" status when he enters the patient room. The active rule R1 is triggered by any output event produced by Q1, represented as `ON OUTPUT Q1`. The action of R1 (defined in the `BEGIN...END` block) states that once a match of Q1 is detected, a rule instance of R1 will update the HCW's status to "warning". It is worth noting that Q1 checks the HCW's hygiene status for query evaluation, while the output of Q1 can result in changing the HCW's status (via triggering R1). In short, here we have demonstrated how to define a pattern query's dynamic condition and real-time action based on the shared table.

4. STREAM TRANSACTIONS

4.1 Notion of Correctness

In this section we introduce our notion of correctness for the simultaneous execution of pattern queries and active rules in the stream context. To better capture the time-based properties of active rules, we first design the timestamp assignment mechanism.

Timestamp of active rule instance. At the moment when the triggering event occurs, the change defined by the rule action is assumed to take effect instantaneously. We model this by associating an application timestamp with each rule instance, denoted as $r_j.ts$, and setting the value to be the same as the timestamp of its corresponding triggering event.

Timestamp of an operation on shared store. For a Write operation $Write_i$ on the shared store performed by an active rule instance r_i, we assign $Write_i.ts = r_i.ts$. For a Read operation $Read_i$ to retrieve the value of a shared store at the application time $t1$, we assign $Read_i.ts = t1$.

Distinguishing Features of our notion of application correctness include: First, it targets ACEP applications, such as the motivating healthcare system, which apply real-time effect to the external world, and thus *do not tolerate the undo or redo* of any externally visible output or action. Second, in our target applications, a Write operation represents a real-time effect, e.g., changing a HCW's badge color. Hence the order of executing Writes must confirm to their timestamp order. In our formal definition below, let $Write_i$, $Write_j$, $Read_k$ and $Read_l$ be operations on a shared table. We use the symbol $\prec$ to denote the preceding order of two operations in the system time.

DEFINITION 6. *(Correctness of Real-time Operations). An algorithm for scheduling operations on a shared table performed by pattern queries and active rules is called **correct** if every schedule produced by the algorithm exhibits the following properties: (1) if* $Write_i.ts < Read_k.ts$, *then* $Write_i \prec Read_k$; *(2) if* $Write_i.ts < Write_j.ts$, *then* $Write_i \prec Write_j$; *(3) if* $Read_k.ts < Write_i.ts$, *then* $Read_k \prec Write_i$; *(4) if* $Write_i.ts = Read_k.ts$, *or* $Read_k.ts \leq Read_l.ts$, *or* $Write_i.ts = Write_j.ts$, *then the order of execution conforms to what the application specifies.*

4.2 Stream Transaction Model

When we attempt to exploit the concurrency control principles from static databases [2, 19] to analyze our stream concurrency problem, a fundamental obstacle we encounter that the concept of a transaction has not been established for stream processing. Traditionally a database transaction corresponds to a user program that is invoked when a user explicitly requests so, while our stream query processing is *trig-*

gered by incoming streaming events. Moreover, a database transaction corresponds to the execution of a sequence of one-time queries [2, 19], while in our stream system the pattern queries are *continuously running.* Therefore we must first define the notion of a transaction in the stream context.

DEFINITION 7. *A **stream transaction**, or short s-transaction, in ACEP is a sequence of ACEP system state changes that are triggered by a single input event.*

This definition considers the active rule execution to be an *in-line extension* of the triggering transaction [10]. The top of Figure 4 illustrates four s-transactions corresponding to three input events respectively. To focus on covering the core requirements drawn from ACEP applications, we first assume there is *no system failure* of the ACEP engine. We then discuss the system recovery for ACEP engine in Appendix H. We also assume that there are no application-specific constraints other than the pattern query semantics defined in Section 3 and the specifications defined in Definition 6 that must hold.

A database transaction has been traditionally defined to be an encapsulated sequence of user operations that must be ACID [19]. However, these ACID properties cannot directly apply to our s-transactions due to significantly different transactional models. We thus have proposed a mapping of the classical ACID properties to ***stream-ACID properties*** (s-ACID), as specified in Appendix D.

5. S-TRANSACTION SCHEDULING

Our s-transaction scheduling algorithms have two objectives: first, to guarantee the *correct* execution of pattern queries and active rules as per Definition 6; second, to assure the near-real responsiveness as required by high performance stream processing. The strict application-time based correctness requirement are in conflict with the high system responsiveness requirement, posing great challenges.

We introduce three solutions for s-transaction scheduling to address this challenge. The first solution, called *Single-Event-Initiated* (SEI) scheduler, requires minimum change of an existing CEP engine and hence is easy to use. The second solution adapts a general-purpose concurrency control mechanism, namely the *Strict Two-phase Locking* (S2PL [2]), to the ACEP context, demonstrating that our proposed notion of s-transactions allows us to leverage existing concurrency control approaches. The third solution, called *Low-Water-Mark* (LWM), successfully combines the optimistic and pessimistic principles to maximize concurrent execution in our stream context and achieves high system responsiveness without compromising the correctness. Preliminary features common across all three algorithms are:

- **Orderness of stream.** Following the state-of-the-art literature [9, 12, 17], we initially assume events are fed into our system in strictly increasing application time order. Later we relax this assumption and extend the schedulers to function also over disordered streams in Appendix F.
- **Rule execution order.** For ACEP real-time applications, we assume that the execution order of rule instances that have the same timestamp makes no difference on the appearance of observable actions. This is a reasonable assumption commonly used in state-of-the-art active database literature [20, 23]. It is also practical in our target applications, since it is the application administrator's responsibility to ensure active rules are well-defined.

S2PL		requested	
		Read	**Write**
held	**Read**		X
	Write	X	X

LWM		requested	
		Read	**Write**
held	**Read**		
	Write	maybe	X

Figure 3: Lock compatibility (X - incompatibility).

- **Termination of s-transaction**. Our current ACEP model uses the no-cascading-trigger assumption. That is, rules are triggered by the stream output of queries while rules do not produce any stream output. Hence cycles in triggering will not arise. We also assume every single s-transaction is finite. Consequently, all s-transactions in ACEP are guaranteed to terminate. This model, while simple, has practical utility as demonstrated by our motivating applications. Similarly, most DBMSs in the literature [6, 10] or in practice (e.g., Oracle) either require finite rule specification or they place a hard-coded threshold on the number of cascading trigger calls allowed.

5.1 Single-Event-Initiated Scheduler

Based on Definition 7, the most direct scheduling strategy would be to take a single input event at a time and to execute all affected queries and rules to converge, i.e., until no more actions are queued to be executed and all output has been generated. This is an intuitive solution due to its simplicity and thus ease of adoption within any CEP engine. However it suffers from the following drawbacks. First, since it only permits one input event to be processed at a time, it may cause a large delay due to blocking a significant number of input events. Second, s-transactions that would not conflict are unnecessarily delayed. This may underload inter-operator buffers and waste processing cycles.

5.2 Strict 2PL Scheduler

We now relax the single s-transaction at a time constraint. When multiple s-transactions are executed concurrently, an s-transaction obtains "locks" on the shared tables it accesses to prevent other s-transactions from accessing these tables at the same time and thereby incurring the risk of errors. We now adopt and adapt the Strict Two-Phase Locking approach (S2PL) to our ACEP context. Below we assume locks are performed at the table level, yet our schedulers can be easily extended to support locks with multiple granularities.

If an s-transaction T_i intends to write the shared table P_k, T_i needs to have a *write lock* on P_k, denoted as $xl_i(P_k)$. If a transaction T_i wishes to read P_k then T_i requests a *read lock*, denoted as $sl_i(P_k)$. Mimicking the classical S2PL [2], our S2PL inserts locks into an s-transaction ahead of all query and rule processing (known as Phase I). All locks applied by an s-transaction T_i are released after the s-transaction has successfully ended (known as Phase II).

The lock insertion algorithm (Algorithm 1) used in Phase I determines Reads and Writes possibly requested by analyzing active rules and queries. We also employ the lock ordering mechanism to avoid deadlock [2]. Consequently, under our in-order-stream assumption, Algorithm 1 ensures no deadlock will occur hence no rollback will be performed. We have thus shown the successful application of a pessimistic concurrency control mechanism originally designed for static databases, namely S2PL, to the ACEP context. However, S2PL incurs a large synchronization delay due to the rigorous lock incompatibility as depicted in Figure 3.

EXAMPLE 4. *Let us consider Examples 2 and 3. First for s-transaction T_1 created for input event exit[2], the lock*

Algorithm 1 InsertLock

```
1: for each input event e_i of type E_i do
2:    initiate s-transaction T_i
3:    // accepting-event-type is defined in Appendix B
4:    if E_i = accepting-event-type of query Q_i
      AND ∃ rule R_j monitors Q_i's output as triggering event
      AND action of R_j contains update on P_k then
5:       insert write lock xl_i on P_k
6:    end if
7:    if ∃ pattern query Q_i
      Q_i consumes e_i and Q_i access P_k then
8:       insert read lock sl_i on P_k
9:    end if
10: end for
```

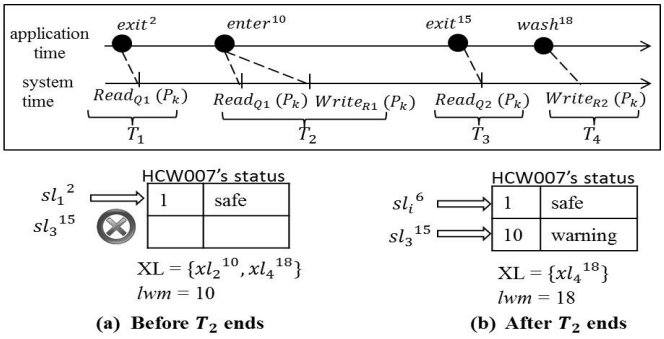

Figure 4: S-transactions and LWM Example.

inserted by S2PL is $\{sl_1(P_k)\}$. *Then for* T_2 *created for* $enter^{10}$, *the locks are* $\{sl_2(P_k), xl_2(P_k)\}$. *Next for* T_3 *created for* $exit^{15}$, *the lock is* $\{sl_3(P_k)\}$. *As we can see, since* T_1 *holds the read lock first,* $xl_2(P_k)$ *from* T_2 *is delayed until* T_1 *releases it. So the read-too-late anomaly is prevented. Similarly,* $sl_3(P_k)$ *held by* T_3 *must wait until* T_2 *unlocks* P_k. *Consequently the write-too-late anomaly is also prevented.*

5.3 Low-Water-Mark Scheduler

We now propose the Low-Water-Mark scheduling strategy (LWM) customized for stream-transaction execution. The pseudocode of LWM and a formal proof of its correctness are in Appendix E.

Consistent application-time based timestamping. A distinguishing characteristic of an s-transaction is that operations issued by an s-transaction have concrete application-time based constraints (as per Definition 6). For that reason, we propose to *timestamp an s-transaction* (say T_i) based on the application timestamp of T_i's triggering input event (say e_i). That is, $T_i.ts = e_i.ts$. This assignment is consistent with our previous timestamp management for the shared store and active rules (Section 4.1), which is a core concept underlying our proposed notion of correctness.

Our scheme of application-time based timestamping is in contrast to the mechanism employed by classical Multi-version Concurrency Control protocol (MVCC) or Timestamp Concurrency Control (TSCC). The later generates timestamps using the system counter in a *first-come-first-served* manner [2, 19]. While our scheme models the real-time application logic that at the moment when the input event occurs, the operations triggered by this event are assumed to take effect instantaneously. The detailed comparison between MVCC and LWM is in Appendix E.2.

Given our timestamping scheme, to allow different s-transactions to access the specific value of the shared store that is appropriate for each s-transaction's application time based progress, we propose to *maintain historic records (multi-versions)* of each shared table. Specifically, when a Write operation updates a shared table, say P_k, the new record of P_k is appended to the end of the sequence of historical records. This technique designed to increase the concurrent execution level mimics MVCC. However, as we demonstrate below, MVCC cannot be applied to solve our s-transaction scheduling problem directly, as it fails to meet our correctness requirement.

Demonstration of failure of MVCC. In MVCC [2, 19], every shared table P_k has an associated Read-timestamp, denoted as $P_k.Rts$, which is set to the maximum of all executed readers' timestamp. If a transaction T_i wants to write P_k and $Ti.ts < Pk.Rts$, then MVCC will *abort* T_i and *restart* T_i with a new, larger timestamp. First, abort of a transaction, representing an undo of externally visible output or action, is not acceptable in ACEP applications, as stated in Section 4.1. Second, restarting a transaction with a different timestamp is equally unacceptable. In LWM, an s-transaction's timestamp reflects the application time when its issued operations should take effect. Hence if a restart were to occur, the newly set timestamp would lead the s-transaction to read and write the wrong version thus incur erroneous results.

Low-water-mark based scheduling. To avoid those undesired side-effects described above, we leverage the approaches of integrating locking and multiversioning [2]. Our key novel technique is an intelligent strategy to *consistently render the synchronization order* of both locking and time-stamping. As the first step, LWM assigns an application-time based timestamp for each lock, after determining locks needed by an s-transaction using Algorithm 1. Specifically, for each write lock $xl_i(P_k)$ (resp. read lock $sl_i(P_k)$) inserted into an s-transaction T_i, we set $xl_i(P_k).ts = T_i.ts$ (resp. $sl_i(P_k) = T_i.ts$).

Observation 1. *Given the multi-versioned shared tables and timestamped locks, a write lock* $xl_i^{t1}(P_k)$ *represents a potential Write on the shared table* P_k *at application time* $t1$. *When* $Read_i(P_k)$ *intends to access* P_k *at application time* $t2$, *to avoid the write-too-later anomaly, it is safe to execute the Read only if* $t2 \leq t1$. *That is, we need to make sure* $Read_i(P_k)$ *obtains the value of* P_k *that will never be affected by any future Write.*

Hence, we introduce the *low-water-mark* (*lwm* for short, a special control parameter) based mechanism to guarantee the correctness. Intuitively *lwm* represents the oldest timestamp among all the timstamps of the write locks on a shared table. And *lwm* is updated whenever a write lock is added or released. *lwm* is formally defined below.

Definition 8. *Given the write lock queue* $XL = \{xl_1^{t1}, ..., xl_n^{tn}\}$ *on a shared table* P_k, *the* ***low-water-mark (lwm)*** *of* P_k, *denoted as* lwm_{P_k}, *is defined to be* $lwm_{P_k} = MIN_{j=1}^{n}\{xl_j.ts | xl_j \in XL\}$.

Our LWM scheduler then synchronizes Reads and Writes based on *lwm*s respectively:

- A read lock $sl_i^{ts}(P_k)$ is granted if $sl_i.ts \leq lwm_{P_k}$; otherwise $sl_i^{ts}(P_k)$ is delayed until $lwm_{P_k} > sl_i.ts$. Intuitively, a Read is guaranteed to access the right record of P_k after all Writes earlier than the Read have completed.
- LWM grants a write lock $xl_i^{ts}(P_k)$ only if $xl_i^{ts}(P_k)$ becomes the oldest write lock among all write locks held on P_k,

namely only if $xl_i(P_k).ts = lwm_{P_k}$. Basically a Write is not allowed to jump the queue of Write requests on P_k.

Given this novel granting strategy, we now can *relax the lock compatibility* (Figure 3). On the one hand, a read lock does not block acquiring a write lock on the shared table. That is, $sl_i^{t1}(P_k)$ will not block $xl_j^{t2}(P_k)$ even if $t1 < t2$. On the other hand, a write lock not necessarily to block a read lock. Namely, $xl_j^{t2}(P_k)$ will not block $sl_i^{t1}(P_k)$ if $t1 < t2$. These compatibilities in turn enable significantly more s-transactions to be executed concurrently.

EXAMPLE 5. *In Figure 4 we illustrate LWM using Examples 2 and 3. Before the s-transaction T_2 created for input event enter*[10] *ends, the read lock sl_1^2 is granted and the corresponding Read accesses the proper version, i.e.,* `(1, safe)`*; while the read lock sl_3^{15} is delayed due to the* `lwm` *constraint. After T_2 ends, the* `lwm` *is updated to application time* 18*. Consequently sl_3^{15} is granted and the corresponding Read obtains* `(10, warning)`*. At this time, any Read operation with timestamp earlier than 18 can be executed immediately and guaranteed to read the right value.*

6. EXPERIMENTAL EVALUATION

We have implemented our proposed ACEP technology within HP CHAOS CEP engine [9]. All queries and rules drawn from our real-world heathcare application are maintained in the *query-rule pool*. The query and rule parameters are listed in Table 1. Further details of our experimental setup are given in Appendix G.1. In this section we study the performance of our s-transaction scheduling algorithms, namely SEI, S2PL and LWM. The performance metrics we measure include *throughput*, *average query latency* L_{query}, *average rule latency* L_{rule} and *average combined latency* $L_{combined}$, as specified in Appendix G.2. For S2PL and LWM, we employ the locking at two-level hierarchy of the shared store: *table-level*, e.g., all HCWs' status, and *tuple-level*, e.g., each individual HCW's status.

Varying Number of Reads vs. Throughput. We evaluate the throughput while varying the average number of Read operations per s-transaction ($readN$) from 1 to 6. The average number of Write operations per s-transaction ($writeN$) is set to 0.25 by selecting the corresponding query-rule pairs from the pool. S2PL and LWM perform locks at tuple level. Figure 5 shows that LWM scales more gracefully than S2PL. This is mainly because LWM is capable to grant significantly more locks concurrently.

We next measure the throughput when the lock granularity for S2PL and LWM is at the coarsest level, i.e., $lockG = table$. Figure 6 shows the results for the three schedulers when we vary $readN$ from 1 to 6. SEI's performance degrades rapidly as $readN$ increases. Because more processing delay per s-transaction leads to a longer blocking period for subsequently arriving events, which in turn causes more processing delays to accumulate. Compared to Figure 5, the performances of both S2PL and LWM degrade due to the coarse lock granularity reducing the concurrency level. However LWM still outperforms the other schedulers, i.e., LWM achieves 2.5x higher throughput than S2PL and 3.4x higher than SEI on average, and gains 3x higher throughput than S2PL and 4.5x higher than SEI when $readN = 6$.

Varying Number of Writes vs. Throughput. We here consider the throughput while varying the average number of Write operations per s-transaction ($writeN$) from 0.125 to 1.0. The average Reads per s-transaction ($readN$) is set to 1. The lock granularity is set to be at tuple level. Figure 7 shows that the throughputs drop as $writeN$ increases. One main reason is that the average processing complexity per s-transaction rises. The increasing cost also comes from more write-locks being requested. LWM beats S2PL due to allowing more overlaps among s-transactions.

Par.	Description
$seqL$	length of the pattern in a query
win	time-based window size (in sec)
$readN$	average num. of Read per s-transaction
$writeN$	average num. of Write per s-transaction
$lockG$	lock granularity (table/tuple)

Table 1: Parameters for queries and rules.

Next we set the lock granularity to be at table level. As per Figure 8, the throughput of S2PL and LWM become more sensitive to $writeN$ compared to Figure 7. Especially when $writeN = 1$, S2PL performs similar to SEI, because on average one table-level write lock is held for every s-transaction and S2PL in fact serializes every s-transaction while synchronizing very few (if any) concurrent executions.

Overhead of Schedulers: Varying Pattern Length vs. Latency. In this set of experiments, we set up the *underloaded* scenario, in which the input rate is set to be much less than the saturation level of the engine. In this scenario, no transaction conflict will occur hence no transactional scheduling is needed. We first measure the empirical baseline of L_{query} and L_{rule} by executing a single query-rule pair without running any transactional scheduler. The baseline results are depicted in Figure 9.

And then we execute the schedulers in the underloaded scenario so as to observe their overhead. Figure 10 depicts L_{query}, L_{rule} and $L_{combined}$ relative to the baseline (as per Figure 9). For example, 2% in the figure corresponds to 2% more latency than the baseline. In this experiment the pattern query length ($seqL$) varies from 2 to 6, and S2PL and LWM perform locks at the tuple granularity. SEI incurs 3x to 6x larger combined latency than S2PL and 3x to 5x larger than LWM. The primary cause is that S2PL and LWM only invoke locks for s-transactions that may potentially read or write the particular tuple. Instead SEI gives every s-transaction the exclusive access to all shared tuples, hence brings significant overhead. LWM bears a slightly larger combined latency than S2PL in this underloaded scenario due to maintaining multi-versions and *lwm*s.

Varying Input Rate vs. Latency. We measure the latency while varying the input rate from 50 to 20000 events/sec. The parameters are $readN = 1$, $writeN = 0.25$, $seqL = 3$, $lockG = tuple$. Figure 11 shows the latency relative to the baseline (Figure 9). We observe that the latency rises as the input rate increases because more events have to be held back waiting to be processed as more events arrive per unit of time. LWM incurs 1/9 of the combined latency of SEI and 1/3 of the combined latency of S2PL when input rate is 2000 events/sec. This confirms the effectiveness of our design of LWM maximizing concurrent execution.

Varying Input Rate vs. Memory. From Figure 12 we can see that, the memory consumption for all schedulers increases with input rate as expected. LWM spends extra memory to maintain multi-versions of the shared store. However, when input rate ≥ 5000, the memory used to buffer input events due to processing delays becomes dominant.

7. RELATED WORK

The scheme of supporting active rules in ACEP borrows several principles from active databases [6, 20, 23], e.g., the

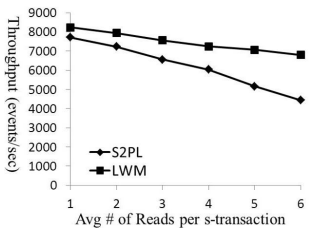

Figure 5: *readN* vs. throughput (*lockG*=tuple).

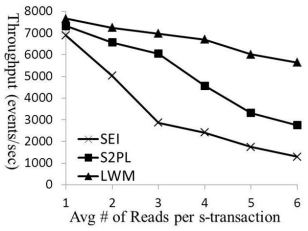

Figure 6: *readN* vs. throughput (*lockG*=table).

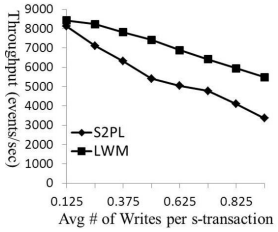

Figure 7: *writeN* vs. throughput (*lockG*=tuple).

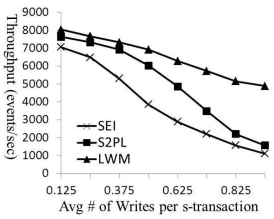

Figure 8: *writeN* vs. throughput (*lockG*=table).

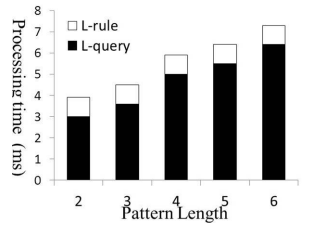

Figure 9: Baseline latency.

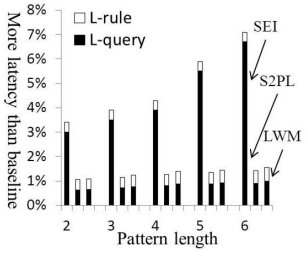

Figure 10: Overhead of schedulers.

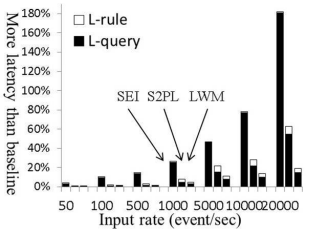

Figure 11: Input rate vs. latency.

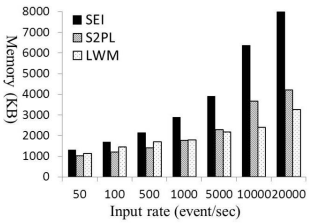

Figure 12: Input rate vs. memory.

"in-line" coupling model [10] and the ECA format. However, ACEP rules differ significantly from active rules in static databases. ACEP rules have the unique formalization of being triggered by the output of continuous queries over infinite streams (Sec. 3). Its distinguishing application time based correctness (Def. 6) and challenging requirement of near-real-time responsiveness (Sec. 5) are also special to the stream context. To our best knowledge, no previous work addressed the issues of defining active rules in the CEP context, nor in the more general data stream processing context.

CEP technologies exhibit sophisticated capabilities for pattern matching in huge volume event streams [1, 4, 8, 12, 16, 17]. However, effort in supporting actions of pattern queries, especially actions that will subsequently affect the pattern matching results, is lacking. Our ACEP technology tackles this unsolved problem by supporting active rules within the CEP context. See additional related work on stream processing in Appendix H.

Transaction processing has been intensively explored in RDBMS. See [2] for a survey of this field. To the best of our knowledge our work is the first attempt at introducing the transaction concept into the stream processing context. Our invention of the stream-oriented ACID proposition and stream-transaction allows us to leverage existing concurrent control principles and further optimize them.

8. CONCLUSION

We have proposed the ACEP technology for supporting active rules in CEP engines to meet the emerging need of specifying a pattern query's real-time actions that may in turn affect query results. We are the first to identify the problem of concurrency control in stream execution, and our innovation of stream-transactions is the first attempt of introducing transactional concepts into stream environments. We then design three s-transaction scheduling algorithms that achieve high responsiveness without compromising correctness. We successfully apply ACEP to a real-world application and experimentally demonstrate its effectiveness.

Acknowledgments. We thank David Lomet and Rimma Nehme for their valuable feedbacks.

9. REFERENCES

[1] M. H. Ali and C. Gerea et al. Microsoft cep server and online behavioral targeting. *VLDB Demo*, pages 1558–1561, 2008.

[2] P. Bernstein and E. Newcomer. *Principles of Transaction Processing*. Elsevier, second edition, 2009.

[3] J. M. Boyce and D. Pittet. Guideline for hand hygiene in healthcare settings. *MMWR Recomm Rep.*, 51:1–45, 2002.

[4] Sybase CEP. Coral8 engine. `http://www.sybase.com/products`.

[5] B. Chandramouli, J.Goldstein, and D. Maier. On-the-fly progress detection in iterative stream queries. *VLDB*, pages 241–252, 2009.

[6] A. Aiken et al. Behavior of database production rules: termination, confluence, and observable determinism. *SIGMOD Rec.*, pages 59–68, 1992.

[7] A. Arasu et al. Stream: the stanford stream data manager. *IEEE Data Eng. Bull.*, 26(1):19–26, 2003.

[8] A. Demers et al. Cayuga: A general purpose event monitoring system. *CIDR*, pages 412–422, 2007.

[9] C. Gupta et al. Chaos: A data stream analysis architecture for enterprise applications. *CEC*, pages 33–40, 2009.

[10] D. Dayal et al. The hipac project: Combining active databases and timing constraints. *SIGMOD Rec.*, 17(1):51–70, 1988.

[11] D. Wang et al. Active complex event processing: Applications in realtime health care. *VLDB Demo*, pages 1545–1548, 2010.

[12] E. Wu et al. High-performance complex event processing over stream. *SIGMOD*, pages 401–418, 2006.

[13] M. Cherniack et al. Scalable distributed stream processing. *CIDR*, 2003.

[14] M. Liu et al. Sequence pattern query processing over out-of-order event streams. *ICDE*, pages 784–795, 2009.

[15] N. Dindar et al. Dejavu: declarative pattern matching over live and archived streams of events. *SIGMOD*, pages 1023–1026, 2009.

[16] R. S. Barga et al. Consistent streaming through time: A vision for event stream processing. *CIDR*, pages 363–374, 2007.

[17] Y. Mei et al. Zstream: A cost-based query processor for adaptively detecting composite events. *SIGMOD*, pages 193–206, 2009.

[18] U.S. Centers for Disease Control and Prevention. `http://www.cdc.gov/ncidod/dhqp/hai.html`.

[19] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1992.

[20] N. W. Paton and O. Diaz. Active database systems. *ACM Computing Surveys*, 31(1):63–103, 1999.

[21] U. Srivastava and J. Widom. Flexible time management in data stream systems. *SIGMOD*, pages 263–274, 2004.

[22] F. Wang, S. Liu, and P. Liu. Complex rfid event processing. *VLDB Journal*, pages 913–931, 2009.

[23] J. Widom. The starburst rule system: Language design, implementation and applications. *IEEE DE Bull.*, 15(4), 1992.

e^{ts}	An input event with application timestamp ts
Qi	An ACEP query
Rj	An ACEP active rule
P_k	A shared table in the shared store
$Read_{Qi}(P_k)$	A Read performed by query Qi on P_k
$Write_{Rj}(P_k)$	A Write performed by rule Rj on P_k
T_i	A stream-transaction
$sl_i(P_k)$	A read lock held by T_i on P_k
$xl_i(P_k)$	A write lock held by T_i on P_k
$sl_i^{ts}(P_k)$	A read lock held by T_i with timestamp ts
$xl_i^{ts}(P_k)$	A write lock held by T_i with timestamp ts

Table 2: Summary of Symbols

APPENDIX

A. MORE MOTIVATING APPLICATIONS

Our work on supporting real-time reactions for pattern queries is motivated by numerous emerging applications, besides healthcare. Below we list two representative ones.

Algorithmic trading. Algorithms developed for automated trading strategies often categorize financial products, say stocks, into various expected-operation groups like "to hold", "to buy" and "to sell" in real-time. Pattern queries monitor the fast-changing trends for a specific group of stocks. At the same time, the results of detected pattern may modify the group of a stock, e.g., a stock may be changed from "to hold" to "to buy". It is critical to provide a way of defining such real-time actions of pattern queries, and to correctly execute such actions given they will change query results.

Fraud detection. In the application of real-time fraud detection, pattern queries selectively consume transactions issued by the sources in the "watch-lists". The outcomes of a query may result in adding a source to "alert-list" or removing a source from the "watch-list". Given the potential impact of millions of dollars, it is vital to support such agile reactions and to control the side-effects to queries themselves caused by the reactions.

B. PRELIMINARIES

Pattern Query Specification. Pattern queries specify how individual events are filtered and multiple events are correlated via time-based and value-based constraints. The syntax of defining a pattern query over event streams we adopt here is commonly used in the literature [12, 17]:

```
CREATE QUERY <query-name>
PATTERN <event-pattern> ON <event-stream>
[WHERE <qualification>]
[WITHIN <window>]
RETURN <output-specification>
```

The `event-pattern` describes an event pattern to be matched. The sequence (`SEQ`) pattern is a core functionality for pattern queries that specifies a particular order in which the events of interest must occur. In the SEQ pattern, we say an event e_i is a positive (resp. negative) event if there is no "!" (resp. with "!") symbol used before its respective event type. The `qualification` clause imposes predicates on event attribute, as in state-of-the-art CEP engines. In addition, in our ACEP system, the predicate can also read the shared store. The `window` clause describes the maximum time span during which events that match the pattern must occur. The `output-specification` clause defines the expected output stream form by the pattern query.

The *accepting event type* refers to the last event type specified in a sequence pattern. For example, in Q1 = `SEQ(EXIT, !SANITIZE, ENTER)`, `ENTER` is considered to be the accepting event type. We focus on the SEQ with positive accepting event type in this work, while negative accepting event handling refers to [14].

Timestamp of Composite Event. As assumed in the literature [8, 12, 17], a composite event output by a pattern query is assigned the application time when the last event instance that composes the composite event occurs. Specifically, given a composite event instance ce_i constructed based on a set of event instances, $I = \{e_1, ..., e_n\}$ that match the query, $ce.ts = MAX_{j=1}^{n}\{e_j.ts | \forall e_j \in I\}$.

ACEP Rule Definition Language. For illustration, we adopt a declarative active rule language implementing our ACEP model described in Section 3 based on the commonly used ECA format [20, 23].

```
CREATE [OR REPLACE] RULE <rule-name>
ON OUTPUT <query-name>
[REFERENCING NEW AS <new-event-name>]
[FOR EACH EVENT]
[WHEN <trigger-condition>]
<action-body>
```

C. ALTERNATE MODELS

While we have chosen the active rule model as foundation of our solution, alternate formalizations to express the reactions for pattern queries are also possible. By describing these alternate models below, we show that no matter which computational model is employed, the issues of concurrent accesses and updates during stream execution would still need to be tackled. For this, our core innovation, including the correctness notion, stream-transactions and the scheduling strategies (Sections 4, 5) could continue to be an elegant solution for executing other models.

Feedback Stream Model. We could collect the results of pattern queries into an intermediate stream and then feed this stream back to the queries. In each query, a predicate could read such feedback stream to determine the selective consumption of original input events. Specifically, we could extend an CEP engine with the following functionalities: (1) Allow a query to delete and/or modify an event in an intermediate stream. (2) Allow multiple queries to publish their results into one single intermediate stream. Note that there needs to be some atomicity and synchronization between the read and removal from such intermediate streams. Now we employ this feedback stream processing model to express our example application logic in Figure 1.

(i) We create an intermediate stream named *InGreenStatus*, which has the schema *(worker-ID, timestamp)* representing that the worker with the *worker-ID* is on the green status since the *timestamp* moment. The operations on this intermediate stream include: `APPEND` a new event (when a worker becomes in green status); `REMOVE` an existing event (when a worker is not longer in green status); `TEMPORAL-JOIN` with a given `to-join-workerID` and `to-join-timestatmp`. `TEMPORAL-JOIN` works as follows: it first checks whether there exists an event, say e_i, in *InGreenStatus* with the *worker-ID* equals to the `to-join-workerID`; if there exists such e_i, then it checks whether $ei.timestamp \leq$ the `to-join-timestamp`. Similarly, create two additional intermediate streams: *InYellowStatus* and *InRedStatus* with the same semantics as above.

(ii) We then define query Q1. Q1 takes two input streams: the original input stream and the intermediate stream *InGreenStatus*. Q1 first detects the pattern `SEQ(EXIT, !SANITIZE, ENTER)`. And then for a matched event sequence, say ce_i, Q1 performs `TEMPORAL-JOIN` on *InGreenStatus* with `to-join-workerID` = ce_i.*worker-ID* and `to-join-timestamp` = $ce_i.ts$. That is, Q1 checks whether the worker with the *worker-ID* was on green status at the moment when his `ENTER` event occurred. If the `TEMPORAL-JOIN` checking returns "true", Q1 inserts an event of the worker into *InYellowStatus*. Namely, a matched Q1 pattern will lead the worker to yellow status. At the same time, Q1 removes the corresponding event of the worker in *InGreenStatus*, because the worker is not longer in green status. Similarly, we also define query Q2 and Q3.

Given the above design, the read-too-late and write-too-late anomalies still arise. For example, when Q1 tries to read *InGreenStatus* for checking worker007's status at application time 15, the event *(worker007, 12)* should have been already inserted by Q2. However, Q2 in fact inserted such event after Q1's checking. In this case, Q1 should have read the information that "worker007 was on green status at time 15" but read some other value instead.

State Transition Machine Model. We could model the interaction between queries and actions of queries using a state transition table in the form of *(current value of a shared table P_k (v-current), set of queries to execute ($\{Q_i\}$), new value of the store (v-new))*. The semantics are when an input event e_i arrives, if the current value of P_k equals to *v-current*, then the queries in $\{Q_i\}$ should consume e_i for evaluation. If a query in $\{Q_i\}$ produces matches, then the value of P_k is updated to *v-new*. In this setting, atomicity and synchronization mechanisms are still needed when multiple queries read and update P_k simultaneously.

D. STREAM-ACID PROPERTIES

s-Atomic (stream-atomic): all operations stimulated by a single input event should occur in their entirety.

s-Consistent (stream-consistent): the execution of stream transactions must guarantee to start in a correct ACEP system state and then end leaving the ACEP system state correct (as per Definition 6).

s-Isolated (stream-isolated): the ACEP system state changes triggered by a single input event must appear to be executed as if no other input events are being processed at the same time, i.e., no unexpected interactions among s-transactions.

s-Durable (stream-durable): the output of the pattern queries must satisfy the "permanently valid" property of query output defined in [14]. That is, at any given time point, all output events from the system so far satisfy the ACEP query semantics.

E. LOW-WATER-MARK ALGORITHM

We now give the pseudocode of the Low-water-mark scheduling algorithm and prove its correctness. The overall workflow of the algorithm is: the Adding & Timestamping-Lock procedure inserts appropriate locks, and timestamps locks and passes the locking information to the Maintaining-LWM procedure. The Granting-Lock procedure determines whether to delay or execute Read and Write operations based on the low-water-mark of a particular shared table. The Releasing-Lock procedure releases locks, and passes the updated locking information to Maintaining-LWM. The algorithm and proof below assume locking is performed at shared table level. But they can be straightforwardly extended to support locking at different granularities.

Algorithm 2 LWM(using object-oriented design)

```
1: class Scheduler {
2: Map<STransaction, List¡SharedTable¿> lockTable
3:
4: void RequestRead(Read_i^ts(P_k)){
5: while ts > P_k.GetLWM() do
6:    //blocking
7: end while
8: Read_i^ts(P_k) is executed }
9:
10: void RequestWrite(Write_i^ts(P_k)){
11: while ts! = P_k.GetLWM() do
12:    //blocking
13: end while
14: Write_i^ts(P_k) is executed }
15:
16: void TimestampLock() {
17: InsertLock() // given in Algorithm 1
18: xl_{R_j}.ts = e_i.ts
19: sl_{Q_i}.ts = e_i.ts }
20:
21: void ReleaseLock(T_i) {
22: List<SharedStore> sStore = lockTable.GetValue(T_i)
23: for each store in sStore do
24:    store.xLockQueue.ReleaseLock(T_i)
25: end for
26: }
27:
28: class SharedTable{
29: Map<time,string> versions
30: XLockQueue xLockQueue
31: SLockQueue sLockQueue
32: string Read(){}
33: void Write(){}
34: time GetLWM(){} }
35:
36: class XLockQueue{
37: List<XLock> locks
38: time lwm
39:
40: void AddLock(xl_{R_j}){
41: locks.add(xl_{R_j})
42: MaintainLWM()}
43:
44: void ReleaseLock(T_i){
45: locks.Remove(all locks held by T_i)
46: MaintainLWM()}
47:
48: void MaintainLWM(){
49: lwm = MIN_{i=1}^{n}(lock_i.ts|∀lock_i ∈ locks)} }
50: //denoted as Up_i(P_k) in the proof}
```

E.1 Proof of Correctness of LWM

Our proof follows the proof paradigm presented in [2]. To prove that LWM scheduler is correct, we have to prove that all executions that could be produced by it guarantee the S-ACID properties. We start by proving the s-Consistent property using two steps. First, given the LWM algorithm we characterize the properties that all of its granted operation sequences must have. Second, we prove that any operation sequence with these properties is guaranteed to meet the correctness specified in Definitions 6.

For the first step, we list all orderings of operations that we know must hold. The function names used below refer to the member functions defined in Algorithm 2. We use $Up_i(P_k)$ to denote the update of the *lwm* of a shared table P_k, which is shown in line 49 in Algorithm 2.

First, from `InsertLock()` we know that every *potential* Write on P_k will add a write-lock $xl_i(P_k)$ into the lock queue of P_k at the beginning of creating the s-transaction. If

$Write_i(P_k)$ is executed later, we know $xl_i(P_k) \prec Write_i(P_k)$ and $xl_i(P_k).ts = Write_i(P_k).ts$. From `RequestWrite()` we also know that when $Write_i(P_k)$ is executed, it must hold that $lwm_{P_k} = Write_i(P_k).ts$. From `ReleaseLock()`, after the $Write_i(P_k)$ completes, lwm of P_k is updated. That is, $Write_i(P_k) \prec Up_i(P_k)$. Therefore, we can derive the following proposition.

PROPOSITION 1. *Let $Write_i(P_k)$ be a Write operation produced by the LWM scheduler, $xl_i(P_k)$ be the write-lock held for $Write_i(P_k)$ on shared table P_k, and lwm_{P_k} be the current lwm of P_k when $Write_i(P_k)$ is executed. Then $xl_i(P_k) \prec Write_i(P_k) \prec Up_i(P_k)$.*

Second, `TimestampLock()` is executed as soon as an input event arrives (before any query or rule processing). Clearly, the locks added for sequential input events are invoked in their timestamp order. In addition, from `RequestWrite()`, we see that the granting of locks is sorted by the timestamp of locks. This leads to the following proposition.

PROPOSITION 2. *Let $xl_i(P_k)$ and $xl_j(P_k)$ be the write locks held by s-transactions T_i (created for input event e_i), T_j (created for input event e_j) respectively. If $e_i.ts < e_j.ts$, then $xl_i \prec xl_j$ and $xl_i.ts < xl_j.ts$. If $xl_i.ts < xl_j.ts$ then $Write_i(P_k) \prec Write_j(P_k)$.*

Third, from `RequestRead()` we know that for a granted Read, $Read_i(P_k).ts < lwm_{P_k}$ must hold. It implies that $Read_i.ts$ falls inbetween two different values of lwm_{P_k} (at different times). That is, there is a previous lwm'_{P_k} and a later lwm''_{P_k}, such that $lwm'_{P_k} \leq Read_i.ts < lwm''_{P_k}$. We thus conclude the following proposition.

PROPOSITION 3. *Without loss of generality, let a $Up_i(P_k)$ operation result in $lwm_{P_k} > Read_i(P_k).ts$, then $Up_i(P_k) \prec Read_i(P_k)$ in system time.*

Summing the above properties gives the following lemma.

LEMMA 1. *The LWM scheduler guarantees the s-consistent property for all s-transactions executed in the ACEP system.*

Proof: The proof is based on the cases in Definition 6. Let $Write_i$, $Write_j$, $Read_k$ and $Read_l$ be operations produced by the LWM scheduler on a shared table P_k, and lwm_{P_k} be the lwm of P_k. For case (1), when $Write_i(P_k).ts < Read_k(P_k).ts$. By Proposition 1, we have $Write_i(P_k) \prec Up_i(P_k)$. If $Write_i(P_k).ts < Read_k(P_k).ts$, without loss of generality, we assume $Up_i(P_k)$ results in $lwm_{P_k} > Read_k(P_k)$. Then by Proposition 3, we have $Write_i(P_k) \prec Up_i(P_k) \prec Read_k(P_k)$. Namely, $Write_i(P_k) \prec Read_k(P_k)$. For case (2), when $Write_i(P_k).ts < Write_j(P_k).ts$, directly derived from Proposition 2, we have $Write_i(P_k) \prec Write_j(P_k)$. For case (3), when $Read_k(P_k).ts < Write_j(P_k).ts$, the LWM scheduler does not explicitly enforce the order of their execution. Hence we exhaust two scenarios: (i) if $Read_k(P_k) \prec Write_j(P_k)$, this is the correct result as expected; (ii) if $Write_j(P_k) \prec Read_k(P_k)$, since the scheduler maintains multi-version of P_k and $Write_j(P_k)$ will not affect the value $Read_k(P_k)$ to read, the result equals to scenario (i). For the case (4), because the LWM scheduler does not change the order of execution specified by the application, the correct execution order is ensured. □

THEOREM 1. *The LWM scheduler guarantees the S-ACID properties for all execution produced in ACEP.*

Proof: Here we prove each s-ACID property respectively.

(1) **s-Isolated**. LWM achieves *snapshot isolation* level on the shared store. To prove LWM implements snapshot isolation, we have to prove that: (a) all Reads made in an s-transaction T_i see a consistent snapshot of the shared store; (b) T_i successfully completes only if no Writes it has executed conflict with any concurrent Writes since that snapshot [2]. Since `InsertLock()` along with `TimestampLock()` are executed as soon as an s-transaction is created, we have that if $T_i.ts < T_j.ts$ then $sl_i.ts < xl_j.ts$ for any read lock sl_i made in T_i and any write lock in T_j. Also from the lwm based granting scheme in `RequestWrite()` and `MaintainLWM()`, we know that if $sl_i.ts < xl_j.ts$, then $Read_i \prec Write_j$ for the corresponding Reads and Writes. Consequently, LWM guarantees that the version (of a shared table) being read by a Read in T_i will not be updated by any Write in T_j. Hence, the proposition (a) is assured in LWM. From `RequestWrite()`, we see that the granting of write locks is sorted by the timestamp of s-transactions. Moreover, we have assumed that the execution order of rule instances that have exactly the same timestamp makes no difference on the appearance of observable actions (in Section 5). We thus know that LWM guarantees the proposition (b). By proving these two propositions we show that LWM implements snapshot isolation.

(2) **s-Consistent**. This has been proven by Lemma 1.

(3) **s-Atomic**. First, our definition of an s-transaction implies that an s-transaction has the "all-or-nothing" proposition. Second, once an event is processed by LWM, all the triggered operations will take place as a whole, under our assumption of "no system failure" (specified in Section 4.2). Hence LWM keeps the s-Atomic property for every s-transaction.

(4) **s-Durable**. According to our ACEP model, an ACEP system can be viewed as a combination of complex event pattern matching and shared store accesses and updates. We next prove the s-Durable property of LWM in these two aspects respectively. First, LWM does not change the underlying sequential pattern matching. Hence, given the reasonable assumption that the pattern matching scheme (without active rule execution) is correct, the pattern matching output of LWM is permanently valid. Second, the Reads and Writes on the shared store scheduled by LWM have been proved to be "correct" (via Lemma 1). In conclusion, LWM ensures that all outputs are s-Durable. □

E.2 Comparison with MVCC

The key differences between LWM and MVCC can be described w.r.t. two aspects. First, *different ways of handling write-to-late anomalies.* MVCC aborts and then restarts the transaction when an outdated Write occurs. While in our LWM, because of the lock associated with each operation and the low-water-mark based lock granting methodology, such outdated Write will be correctly prevented. Put differently, in LWM, a Read is guaranteed to access the right version after all Writes earlier than the Read have completed. Second, *different timstamping mechanisms* and *different supports for disordered streams.* As presented Section 5.3, in MVCC the timestamps of transactions are generated in a first-come-first-served manner. While in LWM, since we target real-time applications, the timestamps of transactions are set to the triggering input event's application timestamp.

As a result, MVCC, if we attempted to apply it to our problem, would have to rely on the input stream being totally ordered to correctly synchronize operations. In contrast, for LWM, we design the advanced punctuation based technique that allows LWM to work gracefully for disordered stream (Appendix F).

F. DISORDERED STREAMS

We now sketch how to extend the proposed s-transaction schedulers to function over disordered streams. First note that our notion of correctness (Definition 6) is independent of whether the input events are in application timestamp order or not. Since SEI and S2PL synchronize executions on a first-come-first-served basis, they rely on the input stream being totally ordered in their application time. We thus propose to apply existing buffer-based techniques for out-of-order stream handling [14, 16], so that events are fed into SEI or S2PL in strict order.

For LWM, we propose to employ *punctuations* that represent the input stream progress - a common paradigm most advanced out-of-order event processing techniques rely on [14, 16]. A punctuation event, denoted as c_i^{ts}, represents that the engine will not see an input event e_i^{ts} with $e_i.ts < c.ts$, i.e., an event that is earlier in application time than the punctuation event. LWM then maintains the timestamp of the *m*ost *r*ecent *p*unctuation (MRP), denoted as *mrp*. The LWM pseudo code (Algorithm 2 in Appendix E) can be remedied by changing two lines:

Line 5: **while** $ts > P_k.GetLWM()$ || $ts > mrp$

Line 12: **while** $ts! = P_k.GetLWM()$ || $ts > mrp$

The proof of the correctness of this modified algorithm follows the methodology in Appendix E.1.

G. EXPERIMENTS

G.1 Experimental Setup

HP CHAOS system indeed employs multi-threaded processing [9]. Our extension for ACEP also uses multi-thread programming. Specifically, each stream operator is executed by a thread. We execute ACEP on Intel Core 2 Duo CPU 3.0GHz with 3.21GB of RAM running Windows 7 and Java JRE 6.

We use the real-world application workload collected from UMass Medical School Hospital, where our HyReminder system is being deployed for clinical studies. Our input stream adapter feeds workloads collected from the hospital with arrival patterns modeled as the uniform process and with various arrival rates.

Referring to the hand hygiene regulations applied in US hospitals [3], we have created ACEP queries and active rules using the following methodology: (1) model the specific sequence of HCW behaviors using *pattern queries*; (2) model the HCW's hand hygiene performance with three status, namely "safe", "warning" and "violation". The status of each HCW is maintained via a shared tuple within a *shared table*; (3) model the logic of a HCW status transitions, namely a certain sequence of behaviors leading the HCW from one status to another status, using *active rules*. We refer an active rule R_j together with the pattern query Q_i whose output is monitored by R_j as a *query-rule pair*. All the application query-rule pairs are maintained in the *query-rule pool*.

Term	Description
e_i^{sts}	An event with system timestamp sts
out_i^{sts}	A query output with system timestamp sts
$numOut$	Number of query outputs so far
T_{ini_i}	System time when the rule instance is created
T_{fin_i}	System time when the rule instance finishes
$numRules$	Number of rule instances executed so far

Table 3: Symbols for performance metric

G.2 Performance Metrics

The performance terms are listed in Table 3. The performance metric *throughput* is measured as number of events processed per second. Namely, given a batch of input events of size $numIn$ ($numIn$ is set to be much larger than the maximum window size of all queries), suppose the system time span taken to process the batch is T_{proc}, then *throughput* $= numIn/T_{proc}$. The metric *average query latency* L_{query} is the average time difference between the time a pattern query Q_i produces an output and the maximum arrival time of any of the event instances that is composed into that output (both system time). Given $out_i = \{e_1, ..., e_m\}$, L_{query} can be measured by:

$$L_{query} = \frac{\Sigma_{i=1}^{numOut}(out_i.sts - max\{e_j.sts | e_j \in out_i, 1 \leq j \leq m\})}{numOut} \quad (1)$$

The metric *average rule latency* L_{rule} corresponds to the average time difference between the time a rule instance is initiated and the time the rule instance finishes, i.e.,

$$L_{rule} = \frac{\Sigma_{i=1}^{numRules}(T_{fin_i} - T_{ini_i})}{numRules} \quad (2)$$

The metric *average combined latency* $L_{combined}$ corresponds to the sum of the average query latency and the average rule latency, i.e.,

$$L_{combined} = L_{query} + L_{rule} \quad (3)$$

H. ADDITIONAL RELATED WORK

[22] presents a methodology to define the reactions for RFID event detection. But it only deals with "trivial" reactions like sending a message or logging events, that do not affect the event detection in turn. Such scenario is simpler than the query-rule interaction problem we study in this paper - thus the concurrency can be safely ignored in [22].

The recovery strategies sketched in STREAM [7] and Aurora [13] stream engines focus on stream-oriented data backup. Because stream sources do not stop delivering data to the engine while the system is down, requiring the engine to "catch up" following a crash. However, [13, 7] do not incorporate the transaction concept, i.e., the notion of consistency in these stream engines does not tie to ACID properties. Therefore these existing stream-specific recovery mechanisms may need to be further re-examined so to adjust them to serve as crash recovery mechanisms consistent with stream transaction semantics employed in the ACEP system. We expect at logging, much like for traditional recovery, but now also including to log the state of stateful sequence operators and the intermediate operations conducted by ACEP rules, will form the foundation for a solution in this new context. Re-do protocols will be applicable, while un-do will not be suitable. To achieve the near real-time responsiveness required to stay in syn with the physical world being monitored, alternate strategies including possibly intelligent load shedding may also need to be incorporated as part of an effective solution.