# High-Performance Complex Event Processing over Streams

Tawatchai Siripanya

Seminar Event-based Systems: Models and Design
S 19566
Instructor Prof. Dr. Agnès Voisard
July 15, 2012
Computer Science Division
Free University of Berlin, Berlin
tsiripanya@googlemail.com

**Abstract.** The rising demands and increasing number of RFID applications for monitoring and tracking purposes invoke the need of their system performance improvement. Many languages for stream processing lack support for large sliding windows and lack constructs to address non-occurrences of events [2]. SASE can execute complex event queries over real time stream of RFID reading. With SASE, it is possible to handle 40,000 events per second for a highly complex query and support large sliding windows. Moreover, SASE is fast to implement. And, it can query filter and correlate events to match given patterns. It transforms the matched events to meaningful events for further uses. This paper gives an introduction to SASE, presents the experimental results based on [2], and describes the limitations of SASE.

## Table of Contents

# 1 Introduction

Today, Radio-frequency identification or RFID Technology has been used in many applications, both in business processes and industry processes [1]. Mainly, RFID is used for monitoring and tracking purposes. RFID can be used to monitor such as sales in retail stores so that we can know if an item is picked up at a shelf and taken out of the store without being checked out [2]. Moreover, RFID technology can be used in warning system (EWS) such as Tsunami Early Warning System [1]. Typically, events are not generated from the same source. It could be events from sensor reading or events from a stock market. In this paper, events are coming from RFID reading. In retail stores for example, RFID components that are attached to items (e.g. books or pens) are called RFID tags. Also, signals from RFID tags can be read from RFID readers. When these signals enter a system, they can be referred as *events* [1]. And, when an event takes place, this can be referred to its *occurrence* [1]. Furthermore, occurrence is linked with a *time* such as one hour or two hours [1].Additionally, when events can be derived from other events; they can be called *complex events* or *composite events*, whereas *primitive events* cannot be derived from others [1].

The paper poses two requirements in context of large-scale event processing: (1) High volume streams and (2) Extracting events from large windows. To (1), given a situation like in a retail store, where e.g. 30,000 of items being monitoring use RFID, the retails management system receives events constantly whenever items are moved or purchased. Therefore, the complex event processing application shall be able to handle high-volume event streams such as 30,000 events per second. To (2), most of event monitoring applications use a sliding window (e.g., within the past 12 hours) to sequences events of concern [2].The events matched to a query, are widely spread with other events across the large window. Hence, the complex event processing application shall be able to handle with the large window issue too.

The paper aims to introduce **SASE** (Stream-based And Shared Event processing), an event processing system that can execute complex event queries over real time stream of RFID reading. Moreover, it is fast to implement. It can query filter and correlate events to match given patterns. Then, it transforms the matched events to meaningful events for further uses [2].

The remainder of the paper is structured as follows. Section 2 provides background information. It describes the basic terminology for SASE event processing language and gives examples of the usages. Section 3 presents a query plan for implementation of the SASE event processing language. An example of the query plan is also presented here. Then, two optimization techniques are introduced to overcome the two described issues. Section 4 evaluates the SASE event processing language, gives the results of the experiment from [2] and describes the limitations of SASE. Finally, Section 5 concludes the paper.

# 2 A Complex Event Language

For better understanding of languages presented in the next sections, we will give definition of terms that will be used throughout the paper.

## 2.1 Definitions

**Events:** [1] gives a description of an event as *"something that happens" i.e., the state transition of an entity of interest at a certain point in time*. An event can be for example an acceleration measured by the accelerometer in our smartphone while we are tilting its screen or new values from items in a retails store read by RFID readings. In this paper, events are denoted by lower-case letter such as '`a`', '`b`', or '`c`'.
**Event Stream:** An event stream is an infinite sequence of events. It is used as the input of an event processing system.
**Event types:** An event type is used to describe a set of attributes that a class of event must contain. In this paper, event types are denoted by upper-case letter such as '`A`', '`B`', or '`C`'.
**Initial Assumptions**: The following assumptions are used throughout this paper. (1)Each event is assigned a timestamp for a discrete ordered time domain. That is, each event always has a time stamp before it enters the event processing system. And, the timestamp can show the true order of the occurrences of the events. (2)Events are totally-ordered. The possibility to support concurrent events will not be covered in this paper.

## 2.2 SASE Event Language

There are input events and output events in SASE event language. The input events are usually called *primitive events* and the output of the query are called *composite events* which can be used as an input of other systems. The following section describes the SASE event query language. And, it has the following structure:

```
EVENT     <event pattern>
[WHERE    <qualification>]
[WITHIN   <windows>]
```

Next, we will describe the clauses of the SASE event language include `EVENT`, `WHERE`, and `WITHIN`. Then, will give an example using these clauses. The examples are based on [2].

**EVENT <event pattern>:** The `EVENT` clause specifies event sources or event type described earlier. That is, we can use `EVENT` clause to tell the system what are the event sources available by specifying its name in the clause. For example: There are two event sources `SHELF-READING` and `DOOR-READING`. Then, we can specify its statement like this: `EVENT SHELF-READING, DOOR-READING`

**[WHERE <qualification>]:** With `WHERE` clause we can evaluate if a given predicate applies its conditions. This is the example: `WHERE category = food`
In the example, our predicate is called "`category`" and its attribute is "`food`". Additionally, we can write many predicates as we want in the `WHERE` clause using the combination of logical connective $\wedge$ and $\vee$.

**[WITHIN <windows>]:** The within clause specifies the time period in which the events of interest must be occurred. For instance, we are interested in the events which occur within the time period of 10 hours, we can write the `WITHIN` clause like this: `WITHIN 10 hours`

As seen above, we are able to use SASE event language to specify the event sources using `EVENT` clause, evaluate the predicate using `WHERE` clause, and specify the time period using `WITHIN` clause. Next, we will give an example of the combination of these three clauses in a query.

**Example of SASE event language**

Suppose our query called "`Q1`", we have two events sources come from `SHELF-READING` and `DOOR-READING`. We want to evaluate only the food category in the shelf reading and the doors reading. Furthermore, the time period we are interested in is within 10 hours. Then, we can write the query like this.

```
Q1: EVENT SHELF-READING, DOOR-READING
WHERE category = food
WITHIN 10 hours
```

Now, there are extra features come with SASE event language that allow us to specify the query more precisely. These include: (1) Logical connective operators $\wedge$ and $\vee$. (2) Comparison operators ( $=, \neq, <, >, \leq,$ and, $\geq$). (3) Negation operator (`!`). (4) Parameterized predicates. (5) Equivalent Test. Meaning that, the equality comparisons on a common attribute across an entire event sequence are typical in RFID application [2]. (6) SEQ operator that tells the system that the events are read sequentially.

**An example how to use (1) and (2):**

```
WHERE category = 'food' ∧ manufacturer_id ='1'
```

**An example how to use (3):**

```
EVENT  SHELF-READING, !(DOOR-READING)
```

That means, the event `DOORS-READING` can be ignored from the system.

**An example how to use (4):**

```
EVENT  SHELF-READING a, !(DOOR-READING) b
```

That means, ‘a’, and ‘b’ are the references of the event sources. Formally, we called ‘a’ and ‘b’ as events, SHELF-READING as event type of ‘a’ and DOOR-READING as the event type of ‘b’.

**An example how to use (5):**

```
a.category=b.category
```

From the example, in short hand we can write it as [category] for the one attribute called “category”. If we have more than one attribute, we can write it like this [attribut**1**, attribute**2** ,…, attribute**n**] for **n** numbers of attribute.

**An example how to use (6):**

```
  EVENT  SEQ SHELF-READING a , !(DOOR-READING b)
```

The events are read starting from event a, and then event b sequentially.

The following example shows the combinations of all described features above.

**Final Example**

```
Q2: EVENT  SEQ SHELF-READING a , !(DOOR-READING b)
WHERE [category] ∧  a.category = ‘food’ ∧ a. manufactur-
er_id =’1’
WITHIN  10 hours
```

### 2.3 Formal Semantics

SASE event language defines its semantics by translating its language constructs to algebraic query expression. Let us define $A_i$ as an event type and it is also a query expression [2]. Each event type has its time stamp (t). The expression $A_i(t)$ is true if an $A_i$ event type occurred at the time t, otherwise false.

There are five operators that SASE supports including ANY operator, SEQ_operator, SEQ_WITHOUT operator, Selection (σ) operator, and WITHIN_operator. Due to the page limitation, we are not able to provide more details and give examples of SASE formal semantic. Its formal semantics are self`-explaining. More details on this can be found in [2].

### 2.4 Examples of application using SASE language

The previous sections, we have introduced SASE event language and its formal semantics. In this section we will give two examples to demonstrate on how to use

them. The first example is for retails management system and the second for the healthcare system. Both of them are based on the examples in [2].

**Retail Management**

EVENT SEQ (SHELF-READING x, SHELF-READING y,
! (ANY(COUNTER- READING, SHELF-READING) z) )
WHERE [id] ∧ x.shelf_id ≠ y.shelf_id ∧ x.shelf_id = z.shelf_id
WITHIN 1 hour

The query is set to handle misplaced inventory. We can see the sequence SEQ consists of reading the item at SHELF−READING x, then followed by the SHELF−READING y. Any reading of the item at the COUNTER- READING and the item of the SHELF−READING back again are not considered. We can distinguish between two shelves using the predicate "x.shelf_id ≠y.shelf_id". Also, in the predicate "x.shelf_id = z.shelf_id", the "z" is referred to a negative component of SEQ. It tells us that the ANY operator reading from SHELF−READING is not from the SHELF−READING x [2].

**Healthcare System**

EVENT SEQ (MEDICINE−TAKEN x, MEDICINE−TAKEN y)
WHERE [name='John'] ∧ [medicine='Antibiotics'] ∧
(x.amount + y.amount) > 1000
WITHIN 4 hours

The example of the healthcare system above would be detected if John has taken the other medicines that adversely with the antibiotics in the prescription. The system can also determine if John has taken his medicine within 4 hours [2].

## 3 Implementation

The section 2 describes SASE complex event language. Now, in this section, we will show how to use the event language to implement a query plan. The first section we will introduce essential operators using in SASE for the query plan. Then, we will give an example (based on [2]) of the query plan. After that, we will give more details about sequence scan and constructions (SSC) which are important part for reading of the input into the system. Finally, the optimization techniques will be introduced at the end of the section.

### 3.1 A Basic Query Plan: Operators

There are six important operators to implement the SASE event language in the query plan. These operators are including; `Sequence Scan (SS)`, `Sequence construction (SC)`, `Selection (σ)`, `Window (WD)`, `Negation (NG)`, and `Transformation (TF)`. These operators will later be used in the next sections to construct a query plan.

**Sequence Scan and Sequence construction (SSC):** Sequence Scan and Sequence construction are always used together. The first thing we have to know is a so called "sub-sequence type". For example, our query looks like this: `SEQ (A,B,!C,D)`. `The` "sub-sequence type" of the query should be `(A, B,D)`.

**Selection (σ):** The same as in traditional relational database, the selection operator filters each event sequences using simple and parameterized predicates. The event sequence is then emitted to the output, if the conditions satisfied the predicate.

**Window (WD):**The window operator is used to determine if the temporal difference between the first and the last events of an event sequence less than the window T which specified in a `WITHIN` clause.

**Negation (NG):** The negation operator is used to filter the negative components of a sequence (`SEQ`). For example, we have `SEQ (A,B,!C,D)`, the operator will check if there exists a "`c`" event that came between the event "b", and the event "`d`".

**Transformation (TF):** At the final step, the transformation operator is then used to transform each event sequence as the output which we called the composite event.

### 3.2 A Basic Query Plan: Example

The previous section describes six operators' necessaries for a basic query plan. In this section, we will give an example on how to use them to construct our query plan. Firstly, we have to create our query. Let us call our new query $Q_3$.

$Q_3$: EVENT SEQ(A $x_1$, B $x_2$, ! (C $x_3$), D $x_4$)
WHERE [$attr_1$, $attr_2$] $\wedge$ $x_1.attr_3$ = '1' $\wedge$ $x_1.attr_4 < x_4.attr_4$
WITHIN T

In the query above, we have the parameterized `x1, x2, x3, x4` and their associated distinct event types `A,B,C,D`. Firstly, the query will match a sequence of an 'a', 'b','d' event without any 'c' event between the 'b' and the 'd' event. Secondly, an equivalent test on the `attr1` and `attr2` will be performed. Beside the equivalent test, there are two predicates to be evaluated. The first one is to evaluate if the $attr_3$ of the 'a' event equal to '1'. In the second predicate, the $attr_4$ of the event 'a' and the event 'd' has to be compared using the operator '<'. Finally, the letter 'T' is our specified windows size. We can see the illustration of an execution plan for the query $Q_3$ in the figure 1. The execution of operators is pipelined, such that event sequences constructed are pipelined through subsequent operators and then added to the final output.

As can be seen in the figure 1, the SSC operator creates seven event sequences. Each of them consists of three fields. For instance; the first event sequence consists of $a_1$, $b_3$, and $d_5$. A subscript of each event represents its timestamp. The operators: selection, window, and negation stepwise filter out the six of seven events created earlier by SSC. Then, each event sequence is converted to a composite event by the transformation operator in the final step.

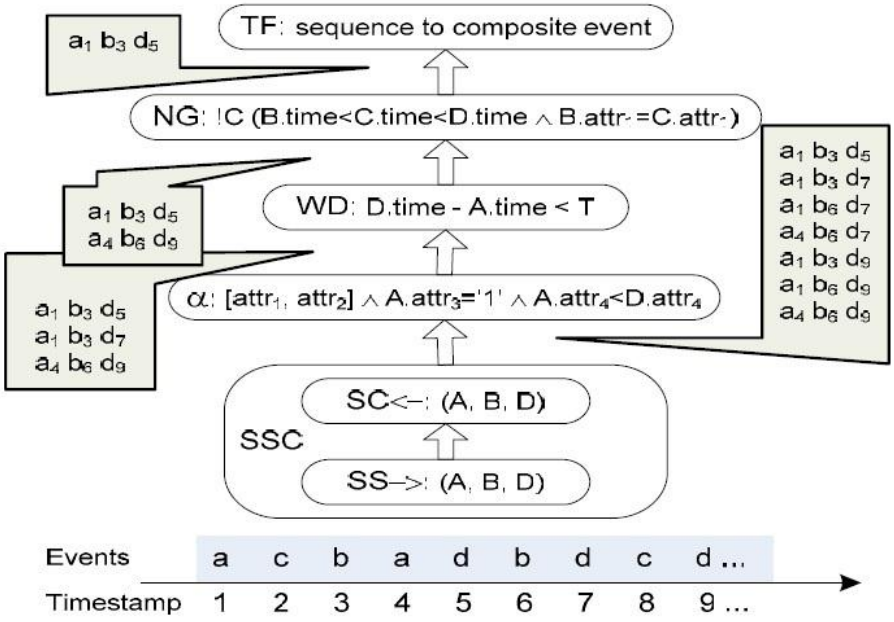


**Fig. 1.** An Execution Plan for the query $Q_3$ [2]

### 3.3 Sequence Scan and Sequence Construction (SSC)

In this section, we will describe sequence scan and sequence construction in detail. We can use a Non-deterministic Finite Automata (NFA) to represent an event sequences. The figure 2 illustrates a NFA created for the sub-sequence type (`A, B, D`). The state '`0`' is the starting state, whereas the state '`3`' is the accepting state, denoted using two concentric circles. The state '`1`' represents successful recognition of an event '`A`' and the state '`2`' for the event '`B`' respectively.

Sequences Scan (SS→).A NFA is created by mapping successive event types to successive NFA states for each sub-sequence type. Unlike the starting and accepting states, each state can contain a self-loop, denoted using (`*`). A self-loop occurrence and the state transition occurrence (from the current state to a new state), both occurrences can be occurred simultaneously. We use "Runtime stack" to keep track these

simultaneous states. The evolution of a runtime stack (from left to right) for the event stream can be seen in the figure 2.

Sequence Construction (SC←). The sequence construction is invoked after an accepting state is reached. Firstly, a so called "single-source directed Acyclic Graph (DAC)" has to be extracted from the runtime stack. To do so, we start the extraction at an instance of the accepting state and traverse back along the predecessor pointers. We stop the traverse, when it reaches instances of the starting state.

As we can see in the figure 2, sequence construction triggered by the occurrence of '`d9`', denoted using thick letters and edges in the runtime stack. Event sequences can be generated by enumerating all possible paths from the source to the sinks of the DAG. But, edges that connect to two instances of the same state (i.e. self-loop) must be omitted in order to generate unique event sequences. We can use depth-first search to enumerate all possible paths [2].

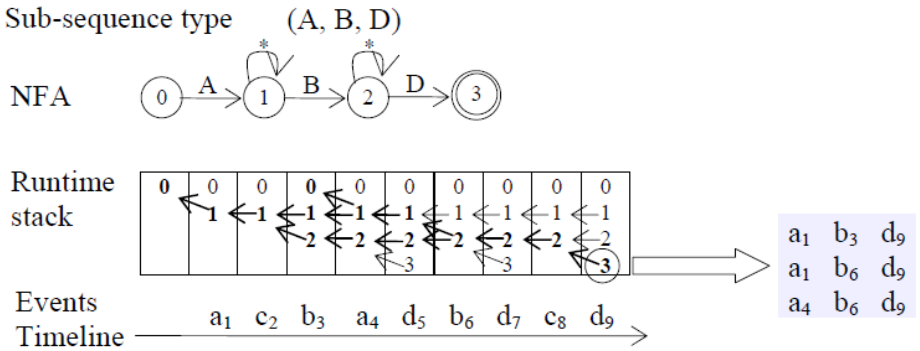


**Fig. 2.** NFA-based Sequence Scan and Construction [2]

### 3.4 Optimization Techniques

This section will introduce two techniques to improve the two issues include large sliding windows and large intermediate result sizes. These techniques include intra-operator to expedite sequences operations in the presence of large sliding windows and inter-operator to push predicates and window constraints down to sequence operators to reduce the size of the intermediate result.

**To expedite Sequence Construction**

We use the technique called "**Active Instance Stack (AIS)**" to expedite sequence construction. Unlike creating only one runtime stack for a NFA as described in the last section, we must create AIS at each state of the NFA to store event transitions. That means, we have to create three AISs for the NFA in the last section. And, we call the event that triggered transition from one state to a new state "**an active instance**".

As is can be seen in the figure 3, we have three event types `A,B,C`. The number of AIS we have to create is three which depends on the number of the event type. Then, we store the recognize events in a temporal order of each AISs. In comparison to a

runtime stack, just only one pointer is stored in an active instance. This pointer is the most recent instance in the previous stack (RIP) at the time when the event that triggered the transition occurred. For example, as can be seen in the figure 3, the RIP of the stack B is $a_4$ because the most recent instance in the stack A before $b_6$ is $a_4$. The figured 3, illustrates the employment of AIS. The RIP pointers are denoted in the brackets " ( . . ) ".

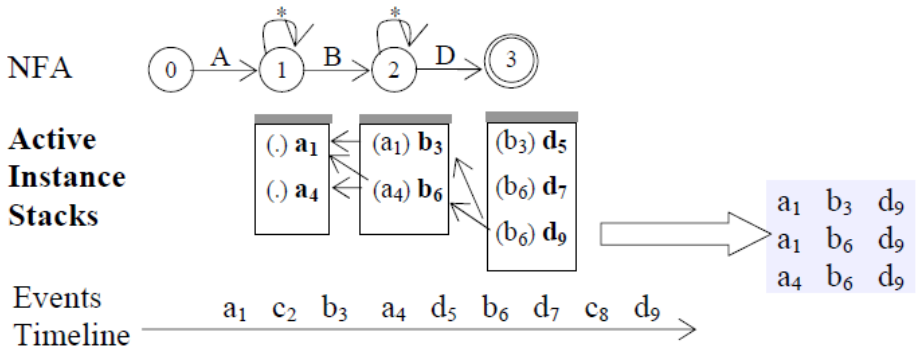


**Fig. 3.** Active Instance Stacks (AIS) [2]

**Pushing Predicate Down to reduce the intermediate result sizes**

To reduce the intermediate result sizes, we have to evaluate predicates early in a query plan. That means we have to re-order the steps what we introduced in the section 3.1. In the section 3.1, we start with steps SSC, Selection, Window (WD), Negation (NG), and Transformation (TF). To reduce the size of the intermediate result, we have to push the steps Selection and Window in the SSC step. That means, the SSC step, the Selection step and the Window (WD) step have to be done simultaneously.

## 4 Evaluation

In the last section, we have shown how SASE event language processing can be implemented with basic query plan. As we have seen, we can also optimize SASE to fulfill a particular condition such as faster scanning and constructing the sequence events and reduce the sizes of intermediate result. Now, we can show you the result of the SASE framework in terms of its performance. We will first describe how the experiment has been setup together with the final result.

### 4.1 Experimental Setup and Results

The experiment in [2] comprised of the following environments: Java-based prototype system. All the experiments were performed on a workstation with a Pentium III 1.4 Gigahertz (GHz) processor and 1.5 Gigabyte (GB) memory running Sun Java 2 Runtime Environment (J2RE) version 1.5 on Fedora Linux 2.6.12. Also, the experiment in [2] had set the Java virtual machine (JVM) maximum allocation pool to 1 GB, so that virtual memory activity had no influence on the results. Furthermore, 20 type of events and 5 attributes for each event type were generated using an event generator simulation program. The experimental result reports that SASE can processes up to 40,000 events per second [2].

### 4.2 SASE Limitations

There is no perfect system in the real world and in the field computer science. Also, SASE framework can deal only with the total ordered events as defined in the assumption. Usually, a composite event has its timestamp from one of its primitive events. The assumption will not valid if both of them are mixed together in order to detect more complex events. Further, SASE is missing possibility to match Kleene closures. That means the possibility to specify one or more occurrences of the same event type in the event pattern. These Kleene closures required to do aggregation.

## 5 Conclusion and Perspectives

In this paper, we have described the two challenges of the complex event processing based on RFID application include large intermediate result sizes and large sliding windows. Then, we have introduced the theoretical introduction to SASE complex event processing include SASE event language and a query plan based approach to implement the SASE event language. The query plan consists of many steps. These steps are including Sequence Scan (SS), Sequence construction (SC), Selection (σ), Window (WD), Negation (NG), and Transformation (TF). However, the first query plan has not been optimized. Then, we have introduced the optimized query plan. The first optimization aims to expedite the sequence scan so that SASE can scan event streams faster. This technique called intra-operator. The second optimization aims to reduce the sizes of the intermediate result. To optimize, the Selection step and the Window step have to be pushed into the SSC step so that the unwanted events can be filtered out earlier. This technique called inter-operator. Finally, the experiment has shown that SASE can processes up to 40,000 events per second. The significant limitation of SASE is that we cannot match Kleene closures, which are required to do aggregation. We believe that this limitation shall be improved, and more features e.g. supporting simultaneous queries, and distributed event processing will be added in the future.

# 6 References

1. A. Voisard and H. Ziekow. Modeling trade-offs in the design of sensor-based event processing infrastructures. Information Systems Frontiers 14(2), pp. 317-330 (2012)
2. E. Wu, Y. Diao, and S. Rizvi. High-Performance Complex Event Processing over Streams. In Proceedings of the 2006 ACM SIGMOD international conference on Management of data, pp. 407–418 (2006)