High-Performance Complex Event Processing over Streams

Eugene Wu
Computer Science Division
University of California, Berkeley
sirrice@berkeley.edu

Yanlei Diao
Department of Computer Science
University of Massachusetts Amherst
yanlei@cs.umass.edu

Shariq Rizvi*
Google Inc.
Mountain View, CA
rizvi@google.com

ABSTRACT

In this paper, we present the design, implementation, and evaluation of a system that executes complex event queries over real-time streams of RFID readings encoded as events. These complex event queries filter and correlate events to match specific patterns, and transform the relevant events into new composite events for the use of external monitoring applications. Stream-based execution of these queries enables time-critical actions to be taken in environments such as supply chain management, surveillance and facility management, healthcare, etc. We first propose a complex event language that significantly extends existing event languages to meet the needs of a range of RFID-enabled monitoring applications. We then describe a query plan-based approach to efficiently implementing this language. Our approach uses native operators to efficiently handle query-defined sequences, which are a key component of complex event processing, and pipelines such sequences to subsequent operators that are built by leveraging relational techniques. We also develop a large suite of optimization techniques to address challenges such as large sliding windows and intermediate result sizes. We demonstrate the effectiveness of our approach through a detailed performance analysis of our prototype implementation as well as through a comparison to a state-of-the-art stream processor.

1 INTRODUCTION

Sensor devices such as wireless motes and RFID (*Radio Frequency Identification*) readers are gaining adoption on an increasing scale for tracking and monitoring purposes. Wide deployment of these devices will soon generate an unprecedented volume of events. An emerging class of applications such as supply chain management [14], surveillance and facility management [18], healthcare [14], tracking in the library [26], and environmental monitoring [8] require such events to be *filtered* and *correlated* for complex pattern detection and *transformed* to new events that reach a semantic level appropriate for end applications. These requirements constitute a distinct class of queries that perform real-time translation of data describing a physical world into information useful to end applications.

An expressive, user-friendly language is needed to support this class of queries for complex event processing. For a concrete example, consider shoplifting detection in a retail store; a query accomplishing this consists of a sequence of events that describe the scenario where an item was picked up at a shelf and then taken out of the store without being checked out. Complex event queries like this can address both occurrences and non-occurrences of events, and impose temporal constraints (e.g., order of occurrences and sliding windows) as well as value-based constraints over these events. *Publish/subscribe systems* [1][5][12][25] focus mostly on subject or predicate-based filters over individual events. Languages for *stream processing* [2][7][19] lack constructs to address non-occurrences of events and become unwieldy for specifying complex order-oriented constraints. *Complex event languages* [4][6][15][16][22][30] developed for active database systems lack support for sliding windows and value-based comparisons between events. While it is not our intention to design a brand new language in this work, we leverage existing complex event languages with substantial extensions to address the needs of a wide range of monitoring applications using RFID technology.

Given a suitable language, it is imperative that queries expressed in this language be efficiently executed to meet demanding performance requirements. Most work on complex event languages in the area of active databases lacks implementation details. Stream processing systems in the relational setting [7][9][19][24] are not optimized for complex event processing, whereas event processing systems very recently developed [18][26][29] have not focused on fast implementations. In this work, we investigate a fast implementation of our proposed language. In particular, we address two challenges that arise in the context of large-scale event processing:

- *High volume streams*: The volume of events generated by large deployments of receptors can reach thousands of events per second or higher. For example, a retail management system set up for a large store receives events whenever items are moved from or to the backroom, placed on or picked from a shelf, purchased, or taken out of the store. Complex event processing must be able to keep up with such high-volume event streams.
- *Extracting events from large windows*: Event monitoring applications often apply a sliding window (e.g., within the past 12 hours) to a sequence of events of interest. In many scenarios, such windows are large and the events relevant to a query are widely dispersed with others across the window. Unlike simple event detection that reports only the satisfaction of a query but not how, extracting relevant events to create all possible results causes significant increase in processing complexity.

In this paper, we present ***SASE***, an event processing system that executes complex event queries over real-time streams of RFID readings. These complex event queries filter and correlate events to match specific patterns, and transform the relevant events into new events for the use of external monitoring applications. Stream-based execution of these queries allows a monitoring application to be notified immediately when all relevant events have been received; as a result, time critical actions can be taken to prevent loss in value and mitigate harm to life, property or the envi-

SIGMOD 2006, June 27-29, 2006, Chicago, Illinois, USA

* The work was done when the author was at the University of California, Berkeley.

ronment. Specifically, we make the following contributions:

1. We propose a complex event language that significantly extends existing event languages [6][30] to meet the needs of RFID-enabled monitoring applications. The extensions include flexible use of negation in sequences, parameterized predicates, and sliding windows. The language is compact and amenable to fast implementation, as we demonstrate in this work.
2. We develop a query plan-based approach to implementing the language. This approach is based on a new abstraction of complex event processing, i.e., a dataflow paradigm with native sequence operators at the bottom, pipelining query-defined sequences to subsequent relational style operators. This abstraction is in sharp contrast to the implementation models of existing event systems based on fixed data structures such as finite automata [16], trees [6], or Petri nets [15]. The abstraction also differs from stream systems in that it uses native sequence operators (rather than joins) to handle query-defined sequences.
3. The new abstraction of complex event processing enables us to explore alternatives to optimize for two salient issues that arise in event processing over streams: large windows and large intermediate result sizes. We develop intra-operator optimizations to expedite sequence operations in the presence of large windows, and inter-operator optimizations that strategically push predicates and window constraints down to sequence operators to reduce intermediate result sizes.
4. We demonstrate the effectiveness of the above techniques through a detailed performance study using a range of data and query workloads. We also compare SASE to a state-of-the-art stream processor. The results of the latter study verify the need for using native sequence operators and highly optimized query plans for high-performance complex event processing.

The remainder of the paper is organized as follows. We describe a complex event language in Section 2. We present an overview of a query plan-based approach in Section 3 and a large suite of optimization techniques in Section 4. We report on the results of a detailed performance study in Section 5. Section 6 covers related work. Section 7 concludes the paper.

2 A COMPLEX EVENT LANGUAGE

In this section, we present the complex event language that SASE uses, and illustrate how this language can be used to support a range of emerging RFID-based applications.

2.1 Event Model

We first describe an event model that serves as a basis for the language we define in the next subsection. In this model, the input to an event processing system is an infinite sequence of events, which is referred to as an *event stream*. An event represents an instantaneous and atomic (i.e., happens completely or not at all) occurrence of interest at a point in time [6]. Similar to the distinction between types and instances in database systems and programming languages, our model includes event types that describe a set of attributes that a class of events must contain. Each event, denoted by a lower-case letter (e.g., '*a*'), consists of the name of its type, denoted by an upper-case letter (e.g., '*A*'), and a set of values corresponding to the attributes defined in the type.

Each event is assigned a timestamp from a discrete ordered time domain. We assume that such timestamps are assigned by a separate mechanism before events enter the event processing system and that they reflect the true order of the occurrences of these events. Furthermore, we assume that events are totally-ordered. This latter assumption, which may not be true in all scenarios, is acceptable in our target applications and allows us to focus on the language and processing issues critical to those applications. Support for concurrent events will be addressed in our future work.

2.2 SASE Event Language

The SASE event language is a declarative language that combines *filtering*, *correlation*, and *transformation* of events: it can be used to specify how individual events are filtered, how multiple events are correlated via time-based and value-based constraints, and how query answers are constructed from the correlated events. In the following, we survey the language and define its formal semantics.

2.2.1 Overview of the Language

The overall structure of the SASE language is:

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

We now explain the various constructs using examples drawn from an RFID-based retail management scenario: A RFID tag is attached to every product in a retail store. RFID readers are installed above the shelves, checkout counters, and exits. A reader generates a reading if a product is in its read range. In our examples, we assume that readings at the shelves, checkout counters, and exits are represented as events of three distinct types.

The first query (Q1) retrieves readings at a shelf about a product whose category is food and whose manufacturer has id '1'. In this query, the EVENT clause contains an ***event type test*** "*SHELF-READING*" that retrieves the events of the *SHELF-READING* type from the input stream. The WHERE clause further filters those events by evaluating two ***predicates*** applied to their attributes: the first predicate requires the value of the attribute *category* to be 'food' and the second requires the value of the attribute *manufacturer_id* to be '1'. In general, the WHERE clause can be a boolean combination (using logical connectives $\wedge$ and $\vee$) of predicates that use one of the six comparison operators ($=, \neq, >, <, \geq, \leq$).

```
Q1: EVENT    SHELF-READING
    WHERE    category = 'food' ∧ manufacturer_id = '1'
```

The second query (Q2) detects shoplifting activity: it reports items that were picked at a shelf and then taken out of the store without being checked out. The EVENT clause of this query contains a SEQ construct that specifies a ***sequence*** in particular order; the components of the sequence are the occurrences and non-occurrences of events of interest. In this query, the SEQ construct specifies a sequence that consists of the occurrence of a *SHELF-READING* event followed by the non-occurrence of a *COUNTER-READING* event followed by the occurrence of an *EXIT-READING* event. Non-occurrences of events, also referred to as ***negation*** in this work, are expressed using the '!' sign. For the use of subsequent clauses, the SEQ construct also includes a variable in each sequence component to refer to the corresponding event.

The WHERE clause of Q2 uses the variables defined previously to form predicates that compare attributes of different events. To distinguish from simple predicates that compare to a constant like those in Q1, we refer to such predicates as ***parameterized predicates*** as the attribute of the later event addressed in the predicate is compared to a value that an earlier event provides (a similar notion was proposed in [10]). The parameterized predicates in this query compare the *id* attributes of all three events in the SEQ construct for equality. Equality comparisons on a common attribute across an entire event sequence are typical in RFID-based applications. For ease of exposition, we refer to the common attribute used for this purpose as an equivalence attribute, and the set of equality comparisons on this attribute as an ***equivalence test***. Our language offers a shorthand notation: an equivalence test on an attribute (e.g., *id*) can be simply expressed by enclosing the attribute name in a pair of square brackets (e.g., [*id*], as shown in the comment on the WHERE clause in Q2). Moreover, if an equivalence test further

requires all events to have a specific value (e.g., '1') for the attribute *id*, we can express it as [*id*='1'].

Finally, the query Q2 also contains a WITHIN clause to specify a time period, e.g., 12 hours, in which the events of interest must occur. In our language, the time period is expressed as a ***sliding window***, as in most stream languages.

Q2: EVENT SEQ(*SHELF-READING x*, ! (*COUNTER-READING y*), *EXIT-READING z*)
WHERE $x.id = y.id \wedge x.id = z.id$ /* or equivalently, [*id*] */
WITHIN 12 hours

Summary of language features. The above examples demonstrate the use of the constructs of our language. As stated previously, our language draws on complex event languages [4][6][16] [22][30] developed for active databases. In comparison, it supports not only basic constructs such as *sequence* and *negation* that existing event languages have, but also crucial new features that many emerging applications require. In particular, our language:

- offers flexible use of negation in event sequences, a significant extension to any existing event language supporting negation;
- adds *parameterized predicates* for correlating events via value-based constraints;
- includes *sliding windows* for imposing additional temporal constraints; and
- resolves the semantic subtlety of negation when used together with sliding windows, which none of the prior work considers.

The addition of these features enables our language to capture a wide variety of event correlations.

Output. Given a sequence of events as input, the output of a SASE query is also a sequence of events. Each result event represents a unique match of the query. Take the query Q2 for example. A result is created for Q2 if a *SHELF-READING* event and an *EXIT-READING* event satisfy the SEQ construct as well as the WHERE and WITHIN clauses. These two input events represent a unique match of the query, hence called the matching component events of the query. For each unique match of the query, the result event contains the *concatenation* of all the attributes of those matching component events. As such, a result event provides all necessary information that monitoring applications may require for conducting further actions. Unlike previous work that focuses on complex event "detection" (i.e., only reporting that an event query is satisfied but not how) [6][15][16][18], we explicitly report what events are used to match the query. This significantly increases the complexity of query processing, as we shall show in Sections 3 and 4.

In the rest of the paper, we refer to an event in an input sequence as a *primitive event*, and one in an output sequence as a *composite event*, as it is composed from a few input events. It is worth noting that the design of the SASE language follows our vision of a fully compositional language—the language would allow the output of a query to be used as input to another. The fact that a SASE query takes a sequence of (primitive) events and produces a sequence of (composite) events enables an extension to full compositionality. This extension is further discussed in Section 2.4.

2.2.2 Formal Semantics

We formally define the semantics of our language by translating its language constructs to algebraic query expressions. To begin with, each event type A_i is a query expression. An event operator connects a number of query expressions to form a new expression. Semantics is added to a query expression by treating it as a function mapping the underlying discrete time domain onto the boolean values *True* or *False* (similar to [6]). For example, the semantics of a base expression A_i, represented as $A_i(t)$, is that at a given point t in time, $A_i(t)$ is *True* if an A_i type event occurred at t, and is *False* otherwise. Below, we describe the set of operators that SASE supports and the semantics of expressions that they form.

ANY operator. The ANY operator can be used in the EVENT clause of a query. It takes a set of event types as input and evaluates to *True* if an event of any of these types occurs. Formally, it is defined as follows:

$\underline{\text{ANY}(A_1, A_2, \ldots, A_n)}\ (t) \equiv \exists\ 1 \le i \le n\ \ A_i(t)$

It outputs the event that occurred at time t as a result.

SEQ_ operator. In the absence of negation, a SEQ construct in the EVENT clause is translated to an expression with a SEQ_ operator. SEQ_ takes a list of n (n>1) event types as its parameters, e.g., SEQ_(A_1, A_2, …, A_n). It specifies a particular order in which the events of interest should occur. It, however, allows an arbitrary number of events to appear between the two events addressed by two consecutive parameters. This operator is formally defined as:

$\underline{\text{SEQ\_}(A_1, A_2, \ldots, A_n)}\ (t) \equiv \exists\ t_1 < t_2 < \ldots < t_n = t\ \ A_1(t_1) \wedge A_2(t_2) \wedge \ldots \wedge A_n(t_n)$

The ANY operator can be used inside the SEQ construct, e.g., SEQ(A_1, ANY(A_{21}, …, A_{2m}), …). The semantics of the corresponding expression can be defined by combining the semantics of SEQ_ and ANY. The definition is omitted in this paper in the interest of space.

A result created by SEQ_ contains the concatenation of all the attributes of the matching component events of the sequence.

SEQ_WITHOUT operator. In the presence of negation, a SEQ construct in the EVENT clause is translated into an expression using a SEQ_WITHOUT operator. Let $S1$ denote A_{11}, …, A_{1m} and $S2$ denote A_{21}, …, A_{2n}. When these event types are used in the SEQ construct without the '!' symbol, we refer to them as *positive components* of SEQ_WITHOUT. Let $\{B\}$ denote an event type that is not allowed to appear, referred to as a *negative component* of SEQ_WITHOUT.

$\underline{\text{SEQ\_WITHOUT}(S1, \{B\}, S2)}\ (t) \equiv \exists\ t_{11} < \ldots < t_{1m} < t_{21} < \ldots < t_{2n} = t$
$A_{11}(t_{11}) \wedge \ldots \wedge A_{1m}(t_{1m}) \wedge A_{21}(t_{21}) \wedge \ldots \wedge A_{2n}(t_{2n}) \wedge (\forall t_i \in (t_{1m}, t_{21})\ \neg \text{B}(t_i))$

This operator specifies that no event of the B type can appear between the two event sequences $S1$ and $S2$.

There are two special cases of SEQ_WITHOUT. The first case, referred to as *negated start*, disallows any event of the B type to appear before the event sequence $S2$, which is denoted as SEQ_WITHOUT($\{B\}$, $S2$). The second case, *negated end*, disallows any event of the B type to appear after the event sequence $S1$, denoted as SEQ_WITHOUT($S1$, $\{B\}$). These two cases are of practical use only when used in combination with the WITHIN clause. Their definitions are postponed until we present the definition of WITHIN.

More general cases of SEQ_WITHOUT include: (1) a negative component can be a single event type or a set of event types connected using an ANY operator, and (2) negative components can be arbitrarily interleaved with positive components. Due to space constraints, we omit formal definitions of these cases in this paper.

A result created by SEQ_WITHOUT only includes attributes of the events that match the positive components of SEQ_WITHOUT; negative components do not contribute to the content of the result.

Selection operator. Recall that the WHERE clause of a query is boolean combination (using $\wedge$ and $\vee$) of simple and parameterized predicates. This clause is translated to an expression with a selection operator (σ). The semantics of the expression is defined for two cases: SEQ_ and SEQ_WITHIN. For SEQ_(A_1, …, A_n), assume that variables x_1, …, x_n refer to the respective events in the sequence. With negation, e.g., SEQ_WITHOUT(A_{11}, …, A_{1m}, $\{B\}$, A_{21}, …, A_{2n}), an additional variable x_b is used to refer to each negative component. Let P denote a set of predicates connected using $\wedge$ and $\vee$.

A selection operator applied to SEQ_ can be defined as:

$\underline{\sigma(\text{SEQ\_}(A_1, \ldots, A_n), \text{P})}\ (t) \equiv \exists\ t_1 < \ldots < t_n = t\ \ A_1(t_1) \wedge \ldots \wedge A_n(t_n) \wedge (\text{P})$

Note that if P contains a predicate referring to the $x_i.a_j$ attribute but the event denoted by x_i does not contain an a_j attribute, the predicate evaluates to *True* by definition in our language. This is designed to accommodate ANY operators used in the SEQ construct.

To define σ for SEQ_WITHOUT, we rewrite P into a disjunctive normal form $P_1 \vee \ldots \vee P_p$ with each P_i representing a conjunction of predicates. We further rewrite P_i as $P_{i+} \wedge P_{i-}$, with P_{i+} denoting the conjunction of those predicates that do not involve a variable referring to a negative component, and P_{i-} representing the rest. Then σ applied to SEQ_WITHOUT can be defined as follows:

$\sigma(\text{SEQ\_WITHOUT}(A_{11}, \ldots, A_{1m}, \{B\}, A_{21}, \ldots, A_{2n}), P)\ (t) \equiv$
$\sigma(\text{SEQ\_WITHOUT}(A_{11}, \ldots, A_{1m}, \{B\}, A_{21}, \ldots, A_{2n}), P_{1+} \wedge P_{1-})\ (t) \vee \ldots$
$\sigma(\text{SEQ\_WITHOUT}(A_{11}, \ldots, A_{1m}, \{B\}, A_{21}, \ldots, A_{2n}), P_{p+} \wedge P_{p-})\ (t)$

$\sigma(\text{SEQ\_WITHOUT}(A_{11}, \ldots, A_{1m}, \{B\}, A_{21}, \ldots, A_{2n}), P_{i+} \wedge P_{i-})\ (t) \equiv$
$\exists\ t_{11} < \ldots < t_{1m} < t_{21} < \ldots < t_{2n} = t$
$A_{11}(t_{11}) \wedge \ldots \wedge A_{1m}(t_{1m}) \wedge A_{21}(t_{21}) \wedge \ldots \wedge A_{2n}(t_{2n}) \wedge (P_{i+}) \wedge$
$\forall t_i \in (t_{1m}, t_{21})\ \neg(B(t_i) \wedge (P_{i-}))$

WITHIN_ operator. The WITHIN clause of a query is translated using a WITHIN_ operator. This operator requires the specified event or event sequence to occur within a window T. Formally, it is defined for SEQ_ and SEQ_WITHOUT as follows:

$\text{WITHIN\_}(\text{SEQ\_}(A_1, \ldots, A_n), T)\ (t) \equiv \exists\ t\text{-}T < t_1 < \ldots < t_n = t\ A_1(t_1) \wedge \ldots \wedge A_n(t_n)$

$\text{WITHIN\_}(\text{SEQ\_WITHOUT}(S1, \{B\}, S2), T)\ (t) \equiv$
$\exists\ t\text{-}T < t_{11} \ldots < t_{1m} < t_{21} < \ldots < t_{2n} = t$
$A_{11}(t_{11}) \wedge \ldots \wedge A_{1m}(t_{1m}) \wedge A_{21}(t_{21}) \wedge \ldots \wedge A_{2n}(t_{2n}) \wedge (\forall t_i \in (t_{1m}, t_{21}) \neg B(t_i))$

Definitions are also given below for the two special cases of SEQ_WITHOUT, namely, negated start and negated end:

$\text{WITHIN\_}(\text{SEQ\_WITHOUT}(\{B\}, S2), T)\ (t) \equiv \exists\ t\text{-}T < t_{21} < \ldots < t_{2n} = t$
$A_{21}(t_{21}) \wedge \ldots \wedge A_{2n}(t_{2n}) \wedge (\forall t_i \in (t\text{-}T, t_{21})\ \neg B(t_i))$

$\text{WITHIN\_}(\text{SEQ\_WITHOUT}(S1, \{B\}), T)\ (t) \equiv \exists\ t\text{-}T + 1 = t_{11} < \ldots < t_{1m} < t$
$A_{11}(t_{11}) \wedge \ldots \wedge A_{1m}(t_{1m}) \wedge (\forall t_i \in (t_{1m}, t]\ \neg B(t_i))$

It is important to note our special treatment of negated end when WITHIN is applied: the first event in *S*1 is required to occur at time *t*-*T*+1. We add this constraint to avoid anomalies. Take "WITHIN_(SEQ_WITHOUT(*A*, {*B*}), *T*)" for example. Without the constraint that a type *A* event must occur at *t*-*T*+1, the query can be satisfied by any *A* event not immediately followed by a *B* event, because we can simply choose *t* as the point in time right after the *A* event to satisfy the query. As we expect such matches to be uninteresting to most users, we add this additional constraint to ensure that an *A* event occurred at *t*-*T*+1 and no *B* event followed it until time *t*.

A final note is that in the absence of negation, a WITHIN clause can be expressed using a predicate, i.e., the difference in time between the first and last events of a sequence is within T. With negation, especially negated start and end, a WITHIN clause can no longer be expressed using a predicate; a WITHIN_ operator as defined above is needed to capture the correct semantics.

2.3 Example Applications

In the previous section, we illustrated our language using two examples. Our language is in fact suitable for many tasks in retail management [14] and a wide range of applications including healthcare [14], surveillance and facility management [18], environmental monitoring [8], network security [23], etc. In the following, we demonstrate the expressiveness of our language using more examples from retail management and healthcare:

Retail management. Besides shoplifting, another important task in retail management is handling misplaced inventory [14], which currently takes an immense amount of time of retail personnel. The combination of RFID technology and our event language provides a means to automate this process, saving tremendous human effort as well as expediting shelf replenishment. A query handling misplaced inventory can be written in our language as:

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 specifies that a misplacement case consists of a reading of an item at Shelf 1, followed by a reading of the same item at Shelf 2, which is not followed by any reading of the item at a checkout counter or back at shelf 1. The predicate "*x*.shelf_id ≠ *y*.shelf_id" ensures that the two first *SHELF-READING*s refer to different shelves. The predicate "*x*.shelf_id = *z*.shelf_id", with *z* referring to a negative component of SEQ, ensures that if the ANY operator returns a *SHELF-READING*, the reading is not from Shelf 1.

Healthcare: The pharmaceutical industry is moving toward a standard in which RFID tags will be placed on pill bottles, affording a healthcare system an opportunity to develop solutions for medical compliance. When RFID readers are placed in the environment where medicines are kept, the system can track if the right medications are being taken at the right time by the right person [14]. For example, the following query can be used to raise an alert if a patient has taken an overdose of antibiotics in the past 4 hours.

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

Another example would be to detect if John has taken other medicines that adversely interact with the antibiotics in his prescription. Many other examples where our language is applicable include safeguarding equipment use, activity monitoring for the elderly, etc. Queries for them are omitted due to space constraints.

2.4 Limitations

It is important to note that the goal of this work is to provide an event language that is compact yet useful to today's RFID-based monitoring applications. Our language currently has several limitations, which we will address in our future work.

Hierarchy of complex event types. Our language allows queries to transform events from primitive types to complex types, but currently not from complex types to (even more) complex types. The latter can be achieved by adding language constructs that feed the output a query as input to another. In this paper, however, we focus on the former simpler problem and seek a fast implementation of it, which serves as an important step towards more sophisticated processing that real-world applications may later require.

Total order on events. Recall that our language assumes total ordering of events. A known issue with this assumption [30] arises in the following scenario: A composite event usually obtains its timestamp from one of its primitive events; when such composite events are mixed together with primitive events to detect more complex events, the assumption of total order on all events no longer holds. This, again, will be considered when we address full compositionality of the language.

Aggregates. Our language can be extended to support aggregates such as count( ) and avg( ). As aggregates over streams have been extensively studied in the field of stream processing [2][7][9][13], we expect to adopt many stream processing techniques in our system. This topic, however, is beyond the scope of this paper.

3 A QUERY PLAN-BASED APPROACH

Having described our complex event language, we next present a query plan-based approach to implementing this language. Our approach is motivated by the observation that most existing event systems use implementation models based on fixed data structures such as trees [6], directed graphs [18], finite automata [16], or Petri nets [15]. In these models, query execution strictly follows the internal organization of a specific data structure and is unable to explore alternative approaches to evaluating the query. Furthermore, we find it hard to extend such implementations to support a richer query language for emerging advanced applications. In con-

trast, our approach employs an abstraction of complex event processing that is a dataflow paradigm with pipelined operators as in relational query processing. As such, it provides flexibility in query execution, ample opportunities for optimization, and extensibility as the event language evolves.

In our new abstraction of complex event processing, a key data structure for the dataflow is the query-defined event sequence. Such event sequences play a central role in translating the query input into the query output. Constructing these sequences, however, has either been done using expensive join operations in stream systems, or been ignored or under-addressed in other event-related systems. Our approach is unique in its way of handling these sequences: We devise native operators to read query-specific event sequences efficiently from continuously arriving events. These operators are used to form the foundation of each plan, feeding the event sequences to the subsequent operators. This arrangement allows the subsequent operators to be implemented by leveraging existing (e.g., relational) query processing techniques.

In this section, we describe basic query plans. A large suite of optimization techniques will be presented in the next section.

3.1 A Basic Query Plan

A query plan in SASE consists of a subset of six operators: *sequence scan*, *sequence construction*, *selection*, *window*, *negation*, and *transformation*. For a concrete example, consider query Q3:

Q3: 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 this query, A, B, C, D represent four distinct event types. The WHERE clause contains a set of conjunctive predicates: (1) two equivalence tests on the respective attributes denoted by $attr_1$ and $attr_2$, which are common attributes of A, B, C, and D, (2) a simple predicate on $attr_3$ of a type A event, and (3) a parameterized predicate that compares a type A event and a type D event on $attr_4$ using '<'. The letter T represents a specified window size.

A basic plan for Q3 and a dataflow created from an example event stream are illustrated in Figure 1. In the event stream presented at the bottom of the figure, a lower-case letter (e.g., 'a') represents an event of the type denoted by its corresponding upper-case letter (e.g., 'A'), and the number below each event is its assigned timestamp. Above the flow, rounded rectangles represent operators in the plan. From bottom-up, these operators are:

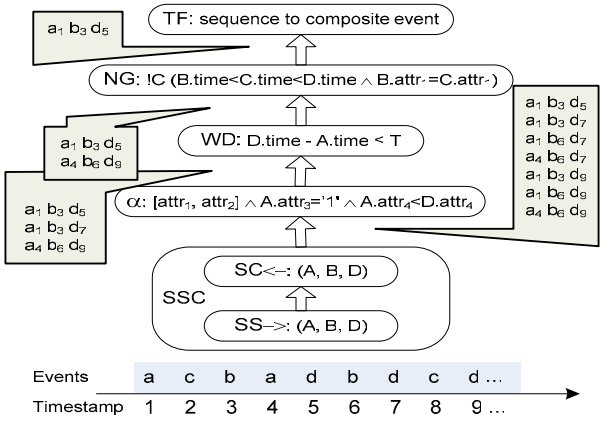


Figure 1: An Execution Plan for Query Q3

Sequence scan and construction (SSC). Sequence scan and sequence construction are always used together, forming a component referred to as SSC. For a query using the SEQ construct in our language, SSC handles the positive components of SEQ, which make up a *sub-sequence type* of the original SEQ specification. For example, the sub-sequence type for Q3 is (A, B, D), which results from removing "!C" from SEQ(A, B, !C, D).

SSC transforms a stream of events into a stream of event sequences; each event sequence represents a unique match of the SSC sub-sequence type. In Figure 1, the output of SSC is illustrated with seven event sequences created from the bottom stream. Each event sequence consists of three fields corresponding to the respective components of the sub-sequence type (A, B, D). The event in each field is denoted by a lower-case letter for its type and a subscript for its timestamp.

Internally, SSC contains a sequence scan operator (SS→) that scans the event stream to detect matches of a sub-sequence type, and a sequence construction operator (SC←) that searches backward (in a data structure summarizing the event stream) to create all event sequences. They will be explained in detail shortly.

Selection (σ). As in relational query processing, a selection operator here filters each event sequence by applying all the predicates including simple and parameterized ones. If the evaluation succeeds, the event sequence is emitted to the output. In Figure 1, three out of seven input event sequences pass the selection.

Window (WD). The window operator imposes the constraint of the WITHIN clause. For each event sequence, it checks if the temporal difference between the first and last events is *less than* the specified window T. In the example in Figure 1, T is assumed to 6, and as a result, the second input event sequence is filtered out.

Negation (NG). The negation operator handles the negative components of a SEQ construct which have been ignored by SSC. In the example of Figure 1, for each input event sequence, this operator checks if there exists a 'c' event that arrived between the 'b' and 'd' events in the sequence and has the same value of $attr_1$ as the 'b' event (thus the same value as the 'a' and 'd' events). If such a 'c' event exists, the event sequence is removed from output. In Figure 1, the second input event sequence to NG is filtered out.

Transformation (TF). Finally, the transformation operator converts each event sequence to a composite event by concatenating attributes of all the events in the sequence.

The execution of the above operators is pipelined: if an arriving event constitutes a match of a query with some previous events, a corresponding event sequence is emitted from SSC right away, pipelined through the subsequence operators, and added to the final output. Such processing is crucial to returning results in a timely fashion so that monitoring applications can trigger fast reaction to the current situation. The implementation of selection, window, and transformation is straightforward. In the following, we explain the implementation of SSC and negation in more detail.

3.2 Sequence Scan and Construction

For sequence scan, a useful approach has been to adopt *Nondeterministic Finite Automata* (NFA) to represent the structure of an event sequence [11][16]. Furthermore, the NFA-based approach can be extended to handle sequence construction, as proposed in YFilter [11] in the context of XML message filtering. We adapted these techniques in a basic implementation of sequence scan and construction (SSC), which is sketched in this subsection. Our main contributions, however, lie in (1) a large set of optimizations developed in this framework for event processing over streams (as opposed to small XML messages), and (2) efficient support for many features missing in XML filtering such as parameterization, windowing, and negation. These advanced techniques are presented in Sections 3.3 and 4.

Sequence scan (SS→). For each SSC sub-sequence type, an NFA is created by mapping successive event types to successive NFA states. For example, Figure 2 shows an NFA created for the sub-sequence type (A, B, D), where state 0 is the starting state,

state 1 is for successful recognition of an A event, state 2 is for the recognition of a B event after that, and likewise state 3 is for the recognition of a D event after the B event.[1] State 3, denoted using two concentric circles, is the (only) accepting state of the NFA. Note that states 1 and 2 contain a self-loop marked by a wildcard '*'. Given an event, these states allow the NFA to loop at the same state, which can occur simultaneously with a forward transition if the type of the event matches that associated with the transition.

To keep track of these simultaneous states, a runtime stack is used to record the set of active states at a certain point and how this set leads to a new set of active states as an event arrives. Figure 2 shows the evolution of a runtime stack (from left to right) for the event stream shown at the bottom. Each active state instance in the stack has one or two predecessor pointers specifying the active state instance(s) that it came from. State 0 is made active at each point to initiate a new search for every arriving 'a' event.

Sequence construction (SC←). Once an accepting state is reached during sequence scan, sequence construction is invoked to create the event sequences that the most recent event has completed. An approach to sequence construction is to extract from the runtime stack a single-source DAG (*Directed Acyclic Graph*) that starts at an instance of the accepting state in the rightmost cell of the stack and traverses back along the predecessor pointers until reaching instances of the starting state. Such a DAG is illustrated using thick letters and edges in the stack in Figure 2, at the instant when the event d_9 is encountered. Event sequences can be generated by enumerating all possible paths from the source to the sinks of the DAG. For each path, edges that connect two instances of the same state (representing a self-loop) are omitted; the remaining edges produce a unique event sequence, which contains the events that triggered the transitions denoted by those edges (see [11] for more details). Figure 2 also shows the three event sequences created from the highlighted DAG in the stack.

A simple algorithm for searching a DAG [11] has the complexity of $O(P)$, where P is the number of paths extracted from the DAG and in the worst case can be exponential. We improved on it by using a single *Depth First Search*, thus reducing the complexity to $O(E)$, where E is the number of edges in the DAG. Since each active state instance has at most two predecessors, E is bounded by $O(2LS)$, where L is the length of the sub-sequence type (thus the number of the states in the NFA), and S is the number of events in the stream. In practice, S can be set to the window size W by using a simple optimization that dynamically checks window constrains in the DAG search, thus yielding the $O(2LW)$ complexity.

3.3 Negation

As mentioned previously, a negation operator (NG) handles the negative components of the SEQ construct in a query which have been ignored by SSC. For each input event sequence, NG performs two tasks for each negative component: (1) check if an event of the type specified in the negative component appeared in a specific time interval; and (2) if such an event exists, check if it satisfies all the relevant predicates. Any event that passes both checks evaluates the current event sequence to *False*. In the following, we focus on the compile-time and runtime support for task (1). The support for task (2) is straightforward and will not be further discussed.

At compile-time, the time interval for task (1) is generated as follows: For a sequence such as SEQ(A, !B, C), the interval is defined as (A.timestamp, C.timestamp), where A and C are bound to the 'a' event and the 'c' event contained in each event sequence. For SEQ(!A, B), the window size T is used to set the interval to be

[1] For a sub-sequence type that contains an ANY() operator, e.g., (A, B, ANY(D, E)), a simple extension is to label the corresponding transition with the set of event types connected using "or".

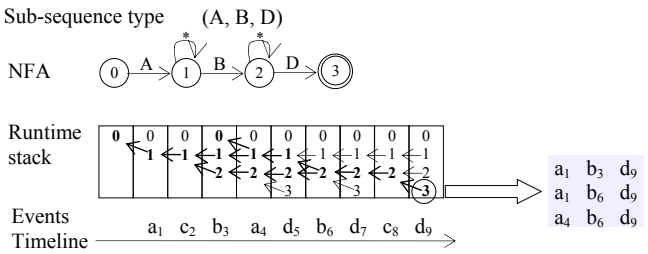


Figure 2: NFA-based Sequence Scan and Construction

(B.timestamp-T, B.timestamp). The handling of SEQ(A, !B) is somewhat special. Recall that given a window T, this query disallows a 'b' event to follow an 'a' event within the window T (as described in Section 2.2.2). Accordingly, the interval created for this query is (A.timestamp, A.timestamp+T). In addition, the negation operator is marked as "postponed by T", which indicates to the runtime system that the evaluation of each event sequence needs to be postponed by a period of length T after its arrival.

The runtime system provides indexing support in addition to postponed evaluation (if necessary). Given a time interval, retrieving all the events that occurred in the interval can be supported by using a standard relational indexing technique. For performance reasons, we use an advanced technique called *partitioned indexing* in this work. The idea is to partition an event stream by timestamp. If δ is the partition size, all the events of timestamp $\in (\delta*i, \delta*(i+1)]$ go into partition i. For each event type involved in negation, say 'B', we build a separate index over each partition. More specifically, when a type B event arrives, if it belongs to the most recent partition, it is inserted to the type B index over this partition; otherwise, a new partition is created and the event becomes the first entry in the type B index. Then during evaluation, given a time interval, a negation operator quickly identifies all the partitions that potentially overlap with this interval, and probes the type B indexes over these partitions to retrieve all the relevant 'b' events. A practical benefit of this approach is that we can garbage-collect an entire partition with all its indexes in one step, once the partition has completely fallen out of the sliding window.

4 OPTIMIZATION TECHNIQUES

We presented a basic query plan for complex event processing in the previous section. This plan has not been optimized to address two salient issues that arise in stream-based event processing: *large sliding windows* and *large intermediate result sizes*. As mentioned in Introduction, large sliding windows are commonly used in monitoring applications. Sequence construction from events widely dispersed in large windows can be an expensive operation. Moreover, if a large fraction of event sequences created cannot lead to final results, tremendous work in sequence construction is wasted and high overhead is incurred in subsequent operators. As in traditional database systems, such intermediate result sizes affect query processing performance. Since stream-based processing usually has stringent performance requirements, reduction of intermediate result sizes is of paramount importance in our context.

In this section, we explore alternative query plans to optimize complex event processing for the above two issues. We develop intra-operator optimizations to expedite sequence scan and construction (SSC) in the presence of large windows, and inter-operator optimizations that strategically push predicates and windows down to SSC to reduce intermediate result sizes. A mechanism shared by all these optimizations is to index relevant events both in temporal order and across value-based partitions.

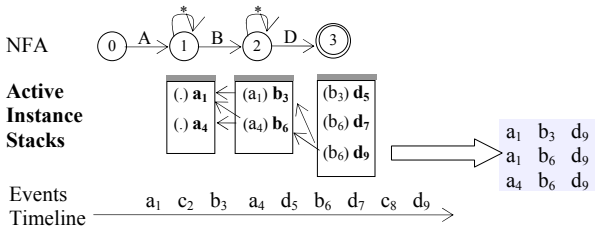


Figure 3: SSC using Active Instance Stacks

4.1 Optimizing Sequence Scan and Construction

As described in Section 3.2, the basic algorithm for sequence construction searches the runtime stack from the most recent event all the way back to the oldest relevant event (the oldest event in the current window that contributes to the query). This can be highly inefficient when queries use large windows. Therefore, we employ an auxiliary data structure, ***Active Instance Stack*** (***AIS***),[2] to expedite sequence construction. The algorithm works as follows:

Sequence Scan. In sequence scan, the NFA execution runs as before. In addition to that, an active instance stack is created at each NFA state to store the events that triggered transitions to this state; such events are referred to as *active instances* of this state. Following the example in Figure 2, Figure 3 shows the content of three AIS stacks after the event stream at the bottom of the figure is received. In each stack, from top-down, the active instances (in bold letters) represent the temporal order of their occurrences. From left to right, a series of three stacks capture the sequencing requirements of the query. Between adjacent stacks, the temporal order relevant to the query is encoded using an extra field of each active instance *e*, which stores the *most recent instance in the previous stack* (RIP) at the moment when *e* occurred. Take the active instance b_6 in the B stack in Figure 3. The most recent instance in the A stack before b_6 is a_4, so the RIP field of b_6 is set to a_4, as shown in the '(' ')' preceding b_6. This RIP field tells that any instances in the A stack up to a_4 can be matched with b_6 if event sequences involving b_6 need to be created. The information of RIP can be visualized by drawing two virtual edges from b_6 to a_1 and a_4 (shown by the arrows in Figure 3).

Sequence Construction. Sequence construction is initiated for each active instance of the accepting state. With active instance stacks, the construction is simply done by a depth first search in the DAG that is rooted at this instance and contains all the virtual edges reachable from the root (note that our implementation only uses the RIP field of each active instance without creating the virtual edges). Each root-to-leaf path in the DAG corresponds to one unique event sequence. The three event sequences created for the active instance d_9 are also shown in Figure 3.

4.2 Pushing Predicates Down

Having described active instance stacks for improving sequence construction, we now turn to address intermediate results sizes. An important optimization for this purpose is to evaluate predicates early in a query plan as in database systems. In SASE, we develop a series of optimizations to strategically push simple and parameterized predicates down to SSC.

[2] Active Instance Stacks (AIS) in SASE appear similar to PathStacks for XML pattern matching [3] in stack arrangements. The stack operations, however, differ significantly: the content of an AIS is determined by NFA transitions and advanced transition filtering presented throughout this section, while the content of a PathStack comes from XML index lookup and node comparison.

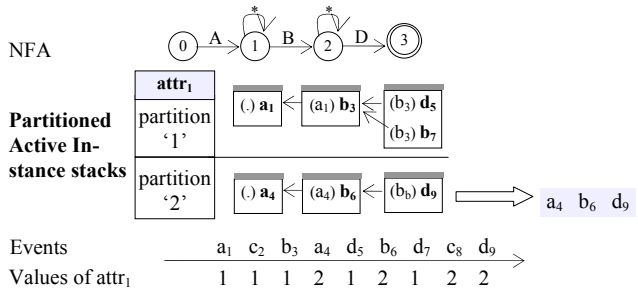


Figure 4: Partitioned Active Instance Stacks (PAIS)

4.2.1 Pushing an equivalence test down to SSC

In RFID-enabled applications, queries commonly use equivalence tests to correlate events that refer to, for example, the same RFID tag, the same patient, the same medicine, etc. Evaluating such predicates in SSC can prevent many unnecessary event sequences from being constructed. In the following, we present a scheme for pushing one equivalence test down to the sequence scan operator.

An equivalence test essentially partitions an event stream to many small ones; events in each partition have the same value for the attribute used in the equivalence test (referred to as the equivalence attribute). One straightforward solution is to partition the stream first and then run the query plan bottom-up for each partition. For better performance, we use an advanced technique, called ***Partitioned Active Instance Stack*** (***PAIS***), that provides two benefits: (1) it simultaneously creates the partitions and builds a series of active instance stacks for each partition during sequence scan, and (2) it incurs no overhead (e.g., partitioning cost) for those events whose types are irrelevant to a query.

The basic idea of PAIS is that at each state, active instances are partitioned based on their values of the equivalence attribute; an active instance stack is created for active instances in the same partition. Furthermore, this stack is connected to the stack in the corresponding partition at the previous state using the AIS algorithm in Section 4.1. Figure 4 shows such an arrangement for the SSC sub-sequence type and event stream used in the previous examples. The equivalence test pushed to SSC is on the attribute $attr_1$. The value of $attr_1$ in each event is shown below the event in the stream. The PAIS algorithm is based on two modifications of the AIS algorithm during sequence scan, described as follows:

Attribute-based transition filtering: At any state except the start state, when the NFA decides to make a transition for the current event (e.g., transition from state 1 for b_6), PAIS retrieves the value of the equivalence attribute from the event (e.g., value '2' from b_6) and checks if the active instance stack in the corresponding partition at the current state (e.g., partition '2' at state 1) is empty. A non-empty stack means previous events of the same attribute value (e.g., a_4) exist, so the transition to the new state is necessary. Otherwise, the current event is dropped.

Stack maintenance: Once a transition is made, the current event is added to the active instance stack at the new state based on its value (e.g., b_6 is added to the stack in partition '2' at state 2), and its field of the most recent event at the previous state is set to the last instance in the corresponding partition at the previous state (e.g., set to a_4 for b_6).

With PAIS, sequence construction is only performed in stacks in the same partition, producing significantly fewer results. In Figure 4, the construction for d_9 only produces one event sequence, compared to three before.

4.2.2 Pushing multiple equivalence tests down to SSC

Queries can contain multiple equivalence tests, for example, to correlate events that refer to the same patient taking the same medication but at different points in time. Intermediate result sizes

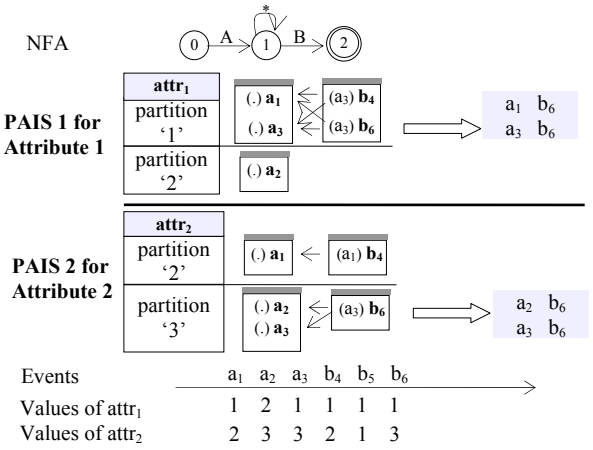


Figure 5: Multiple Partitioned Active Instance Stacks (Multi-PAIS)

can be further reduced if all equivalence tests can be pushed down to SSC. A naïve extension to the PAIS algorithm is to create multi-attribute partitions and build a series of active path stacks for each partition. This approach, however, does not scale as the number of partitions grows exponentially, incurring high memory overhead when the domain size of each equivalence attribute is large. Next, we propose two alternative approaches to pushing multiple equivalence tests to SSC without incurring significant memory overhead.

Eager Filtering in SS→. The first approach, called ***Multi-PAIS***, pushes all equivalence tests to sequence scan, in hopes to filter more events in the "transition filtering" step of the PAIS algorithm. For ease of exposition, we consider a simple subsequence type (A, B) and two equivalence tests on $attr_1$ and $attr_2$. Figure 5 shows the PAIS arrangement for it: at each NFA state, a collection of partitioned active instance stacks, denoted as $PAIS_i$, is created for each equivalence attribute $attr_i$. To understand the content of these stacks, we describe how the stacks are constructed using the Multi-PAIS algorithm:

Cross-attribute transition filtering: At any state except the start state, when the NFA suggests a transition for the current event (e.g., a transition for b_6 from state 1), the event is filtered by (1) for each $attr_i$, probing $PAIS_i$ at the current state (e.g., $PAIS_1$ and $PAIS_2$ at state 1) and retrieving the relevant stack, denoted as $stack_i$, and (2) intersecting all the $stack_i$ (i=1, 2, …). A non-empty result of the intersection means for the current event (e.g., b_6) there is a previous event (e.g., a_3) that matches on all equivalence attributes. In the positive case, the transition is made. In the example of Figure 5, the cross-attribute filtering fails for b_5, so it is dropped. Note that b_5 can not be filtered if we only have a PAIS over $attr_1$.

Multi-stack maintenance: At the new state after the transition, the current event is added to the appropriate stack in each $PAIS_i$ for $attr_i$. For example, at state 2, b_6 is added to the stack in partition '1' of $PAIS_1$ and the stack in partition '3' of $PAIS_2$.

Although Multi-PAIS performs aggressive filtering in sequence scan, superfluous results can be produced in sequence construction. Here, sequence construction can be run in any of the $PAIS_i$ (i=1, 2, …). Back to the example in Figure 5, no matter which $PAIS_i$ we choose, two event sequences are created for b_6, although only a_3 actually matches b_6 on both $attr_1$ and $attr_2$. Take $PAIS_1$ for example: b_6 is erroneously matched with a_1, because they have the same value of $attr_1$—this is exactly what $PAIS_1$ tries to remember, disregarding the information of other attributes. This reveals the problem of "lossy" encoding when multiple PAISs are created separately for individual attributes. This problem cannot be avoided without creating multi-attribute partitions. As a result, the selection operator outside SSC is still needed to filter out the superfluous results created by the Multi-PAIS algorithm.

Dynamic Filtering in SC←. The second approach, ***Dynamic Filtering***, pushes one equivalence test (the most selective one when statistics is available) to sequence scan, and then pushes all other equivalence tests to sequence construction. These equivalence tests are performed in the search over the DAG embedded in the active instance stacks (see Section 4.1). Specifically, if instances on a root-to-leaf path in the DAG have the same values for each equivalence attribute, an event sequence is created; otherwise, the path is ignored. The further details are omitted here due to space constraints. Compared to Multi-PAIS, Dynamic Filtering cannot filter as many events in sequence scan, thus having more instances in the stacks, but does not need to pay for the overhead of cross-attribute transition filtering and multi-stack maintenance.

SASE can also push simple predicates (i.e., predicates applied to individual events) to sequence scan in SSC. The details are omitted in the interest of space.

4.3 Pushing Windows Down

Similar to predicates, window constraints can also be evaluated early in SSC to reduce the number of event sequences created. As mentioned in Section 3.2, windows can be pushed to sequence construction (SC←) that uses Depth-First-Search (DFS) over a DAG contained in the runtime stack. Similarly, when active instance stacks are used, windows can also be dynamically checked in the DFS over the DAG embedded in the active instance stacks. We call this algorithm ***Windows in SC***. We also offer a technique that further pushes windows down to sequence scan (SS→), thus referred to as ***Windows in SS***. This technique offers two benefits: (1) it performs window-based filtering of events, so fewer events are actually added to active instance stacks; and (2) it dynamically prunes active instance stacks by removing events that have fallen out of the sliding window. The latter is important in stream processing where runtime data structures need to be pruned to avoid memory depletion. We omit the details due to space limitations.

Windows in SS and *Windows in SC* can be used together: The former filters some of the events so they are not added to active instance stacks and prunes expired instances from stacks. The latter searches those stacks and performs window checking on-the-fly for each event sequence to be generated.

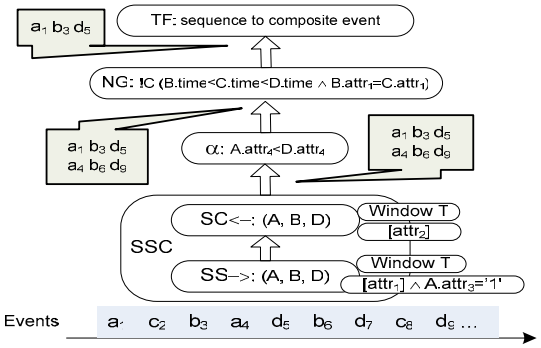


Figure 6: An Optimized Plan for Query Q3

4.4 Putting It All Together

Now we apply the optimization techniques presented in this section to Query Q3 (from Section 3.1). A resulting plan is shown in Figure 6. Compared to the basic plan in Figure 1, this plan has the following differences: (1) the window operator is pushed to both SS→ and SC←, as described above; (2) the equivalence test over $attr_1$ (assumed to be the more selective one between $attr_1$ and $attr_2$) is pushed down to SS→; (3) the simple predicate $A.attr_3 = '1'$ is also pushed to SS→; and (4) the equivalence test over $attr_2$ is pushed to SC←, instead. Figure 6 also shows a dataflow created

for the event stream at the bottom. Here, SSC in the optimized plan only produces two event sequences (as opposed to seven in Figure 1), so the intermediate result sizes have been greatly reduced.

5 PERFORMANCE EVALUATION

In this section, we present a detailed performance analysis of SASE. We demonstrate the effectiveness of its query processing and optimization techniques. We also compare SASE to a state-of-the-art stream processor to provide insights into the strengths and limitations of different design and implementation strategies.

5.1 Experimental Setup

We implemented all the techniques presented in the previous sections in a Java-based prototype system. All the experiments were performed on a workstation with a Pentium III 1.4 Ghz processor and 1.5 GB memory running Sun J2RE 1.5 on Fedora Linux 2.6.12. We set the JVM maximum allocation pool to 1 GB, so that virtual memory activity had no influence on the results.

To test the system, we implemented an event generator that creates a stream of events using the parameters shown in Table 1. In our experiments, we considered 20 events types and 5 attributes for each event type excluding the timestamp. For each attribute, the number of possible values this attribute can take (the domain size) was chosen from the range [10, 10,000]. We did not consider events with more attributes because the additional attributes are not used in our queries and can be projected out before entering SASE.

Table 1: Parameters for event generation

Parameter	Description	Values used
T	Number of event types	20
$\theta 1$	Zipf distribution of occurrences of event types	0
A	Number of attributes per event	5
V_i (i=1…5)	Number of values allowed for attribute $attr_i$	[10, 10,000]

We also created a query generator based on the parameters listed in Table 2. Among them, *EP* specifies the number of equivalence tests (each contains equality comparisons across all events in a sequence on a specific attribute), and *IP* determines the number of other parameterized predicates each of which is an inequality comparison between two events. The size of the sliding window, W, is specified using the number of events.

Table 2: Parameters for query generation

Parameter	Description	Values used
L	Length of the sequence in each query	2-6
$\theta 2$	Zipf distribution of event types in a sequence step	0
EP	Num. of equivalence tests per query	1-2
IP	Num. of other parameterized predicates per query	0-1
SP	Num. of simple predicates per query	0-1
N	Num. of negations in the sequence	0-2
W	Window size	10K-100K

In this study, we define query selectivity as the number of results generated per event (averaged over a sequence of events). Based on probability theory, we derived formulas to approximate true query selectivity using our query workload parameters. For example, the formula below is for a query with one equivalence test over $attr_i$ and no negations.

$$\text{Query Selectivity} = (W \text{ choose } L)/(T^L * V_i^{L-1} * W) \quad (1)$$

In our experiments, we used such formulas to choose appropriate settings in data and query generation to control query selectivity.

The performance metric used in all our experiments is throughput, that is, the number of events processed per second. In each run of an experiment, we used an execution model that switches between event generation and event processing, and computed throughput as follows:

Repeat
(1) Create a batch of W events based on current configuration;
(2) Start stopwatch;
(3) Execute on the current batch;
(4) Stop stopwatch;
(5) Compute throughput as an average over the last 6 batches;
Until throughput converges;

The criterion for convergence is such that the difference between the throughput computed for the current batch and that for the previous batch is within a threshold (set to 5%), and this trend holds true for 3 successive batches.

5.2 Optimizations of Sequence Construction

We begin our study by examining the effectiveness of our optimization of sequence scan and construction (SSC). We compare two algorithms: the Basic algorithm (presented in Section 3.2) that constructs event sequences from the runtime stack used by the NFA,[3] and the AIS algorithm (presented in Section 4.1) that builds active instance stacks for sequence construction.

In this set of experiments, we used the following template for creating queries: EVENT SEQ(E_1, E_2, … , E_L) WHERE [$attr_1$] WITHIN W, where E_1, E_2, … E_L represent different event types. Each query contains a single equivalence test over attribute $attr_1$. In order to decouple the impact of optimizations for predicate evaluation from this study, we did not evaluate the equivalence test in this set of experiments. Instead, we "simulated" the effect of the equivalence test on query selectivity by increasing the number of event types by a factor of $V_1^{(L-1)/L}$ (derived from formula (1)). Predicate evaluation is the focus of the next set of experiments. We pushed windows down to SSC for their evaluation.

Experiment 1-Varying domain size V_1. In the first experiment, we considered a modest window size of 10,000, and examined the performance of the two algorithms as the query processing load varies within the fixed window. To do so, we fixed the path length at 3, and varied the domain size V_1 of $attr_1$ (used in the equivalence test) from 100 to 10,000. In this range, the query selectivity decreases from 0.2 (one result every 5 events, an extremely high number) to 0.2×10^{-4} (one result every 50,000 events).

Figure 8 shows the throughput results of the two algorithms. Note that the X-axis is presented in a logarithmic scale. As can be seen, AIS outperforms Basic by a large factor when the domain size is relatively small, e.g., x18 at the point of 100. In the range of small domain sizes, sequence construction is invoked frequently and significant numbers of results are generated in each invocation. As Basic has a cost proportional to the window size for sequence construction, frequent sequence construction magnifies its overhead. AIS avoids this problem by using active instance stacks, resulting in remarkably improved performance. As the domain size increases, both algorithms improve, because the number of query results decreases. They become close at the point of 10,000 where less than 1 result is created over each period of 10,000 events.

Experiment 2-Varying window size W: In this experiment, we investigate each algorithm's sensitivity to large window sizes. We fixed V_1 at 10,000 and L at 3, and varied W from 10,000 up to 100,000. As we set V_1 large, query selectivity is high and W only has a modest impact on it, e.g., from 0.2×10^{-4} to 0.2×10^{-2}.

The results are shown in Figure 9. As W increases, the Basic algorithm decreases its throughput much faster than the AIS algorithm. The reasons are two-fold. First, sequence construction in Basic incurs a cost linear to W, whereas AIS searches a DAG embedded in the active instance stacks, whose depth is only L. Second, the runtime stack that Basic uses grows large with increasing

[3] The basic algorithm is an improved version of YFilter [11]. Although we did not directly compare to YFilter, the results reported here provide insights into the performance gains that SASE may have over YFilter.

values of W, causing significant memory overhead. With active instance stacks, AIS eliminates the need of using the runtime stack other than the top element for the most recent event, thus avoiding the penalty of excessive memory usage.

5.3 Optimizations for Predicate Evaluation

In this set of experiments, we evaluate the effectiveness of our techniques for pushing predicates down to SSC to reduce intermediate result sizes. For query generation, we added various predicates to the basic template: EVENT SEQ(E_1, E_2, E_3) WHERE [$attr_1$] WITHIN 10000. We used AIS for sequence construction and pushed windows down to SSC in all these experiments.

In an initial experiment, we evaluated the PAIS algorithm (as described in Section 4.2.1) for pushing the first equivalence test down to the sequence scan operator (SS→) in SSC. We compared it to a basic query plan that evaluates predicates in the selection operator outside SSC. The latter actually could not complete the experiment as it created too many (e.g., hundreds of millions of) intermediate results. These initial results show that pushing at least one equivalence test down to SSC is a must. In the following, we investigate the efficient evaluation of additional predicates.

Experiment 3-Two equivalence tests: In this experiment, we added a second equivalence test [$attr_2$] to the basic query template, and compared three strategies to evaluate it: (1) evaluating it in Selection outside SSC, (2) pushing it all the way down to sequence scan (SS→) using the Multi-PAIS algorithm (see 4.2.2), and (3) pushing it down to sequence construction (SC←) using the Dynamic Filtering algorithm (also see Section 4.2.2). Assuming that we can push the more selective equivalence test down to SS→ (when statistics is available), this experiment seeks a strategy appropriate for the second equivalence test corresponding to the selectivity of the first one already pushed down. To do so, we varied V_1 (domain size of $attr_1$) from 10 to 10000 while fixing V_2 (domain size of the $attr_2$) at 20 or 5. In the healthcare scenario, for example, V_1 would be for the patient name and V_2 for the medicine name.

The results for V_2=20 are reported in Figure 10(a). Again the X-axis is in a logarithmic scale. This figure shows that Dynamic Filtering outperforms the other two by a wide margin when the domain size V_1 is relatively small (e.g., ≤500). Surprisingly, by doing eager filtering in SS→, Multi-PAIS yields throughput even worse than Selection. As V_1 increases, the difference among three algorithms decreases, as the query selectivity increases. After the point of 500, three algorithms perform similarly.

The results for small values of V_1 are of particular interest. Two factors contribute to these results. First, in sequence scan, Dynamic Filtering and Selection only evaluate the 1st equivalence test, while Multi-PAIS also evaluates the 2nd equivalence test. By doing so, Multi-PAIS reduces the number of invocations of sequence construction (as verified by our profiling results), but at an extra cost that does not exist in the other two algorithms. Second, in sequence construction, Multi-PAIS actually creates much more results than Dynamic Filtering (but somewhat less than Selection), despite a lower number of invocations. For example, Figure 10(b) shows the actual number of results (in a logarithmic scale) created over a period of 10000 events. Due to the lossiness of its stack encoding, Multi-PAIS creates many superfluous results, as discussed in Section 4.2.2. In contrast, Dynamic Filtering can filter out many unnecessary results during sequence construction. Combining both factors, Dynamic Filtering performs the best, and Multi-PAIS is the worst. The overhead of Multi-PAIS is magnified in the case of V_2=5 where the 2nd equivalence test is less selective. Details are omitted here in the interest of space.

The results of this experiment imply that if we push down the more selective equivalence test, say [$attr_1$], to SS→, there are two main cases to consider for [$attr_2$]: If [$attr_1$] is selective, we can use any strategy for [$attr_2$]. Otherwise, pushing the even less selective [$attr_2$] to SS→ is not effective; instead, a better way is to evaluate it dynamically in SC←. Therefore, we always use Dynamic Filtering for the 2nd equivalence test in the following experiments.

Experiment 4-Adding more predicates: In the next experiment, we further added simple and generic parameterized predicates. Due to space constraints, we only summarize the results here: Pushing down simple predicates always helps reduce intermediate results, thus improving throughput. Once equivalence tests and simple predicates are pushed to SSC, evaluating other parameterized predicates in Selection incurs little overhead.

Other experiments. We also ran experiments to evaluate the techniques for handling windows and negations. We omit details of these experiments in the interest of space. In summary, pushing windows down to both sequence scan and sequence construction is effective in reducing intermediate results. The cost of processing negation is modest when the intermediate result sizes are small, and can become more significant otherwise. This suggests that we might even consider pushing negation down to SSC.

In the rest of this section, SASE was configured based on the results reported in the above experiments: Specifically, it uses Active Instance Stacks for sequence construction; for typical queries such as query Q3, it pushes equivalence tests, simple predicates, and windows down to SSC, as illustrated in Figure 6.

5.4 Comparison to TelegraphCQ

In this section, we compare SASE to a relational stream processor, TelegraphCQ (TCQ) [7], developed at the University of California, Berkeley. We chose to compare to TCQ because it is a full-fledged stream processor with the software publicly available. In addition, TCQ has a well-supported user community, which facilitated this comparative study.

As TCQ does not support negation, we used a relatively simple template for query generation: EVENT SEQ(E_1, E_2, … , E_L) WHERE [$attr_1$ (, $attr_2$)?] WITHIN W. Queries were created based on specific settings of L, W, V_1 (domain sizes of $attr_1$), and V_2 (domain size of $attr_2$), if used. Then, each event query was translated to the TCQ language. For example, a query created with L=3, W=10,000, and one equivalence test [$attr_1$] can be expressed in TCQ as:

```
WITH
R AS     (SELECT * FROM ES e WHERE e.event = 'E1')
S AS     (SELECT * FROM ES e WHERE e.event = 'E2')
T AS     (SELECT * FROM ES e WHERE e.event = 'E3')
(SELECT *
 FROM   R r      [RANGE BY 10000]
        S s      [RANGE BY 10000]
        T t      [RANGE BY 10000]
 WHERE  r.attr1 = s.attr1 AND r.attr1 = t.attr1 AND
        s.time > r.time AND t.time > s.time)
```

The TCQ query first uses the WITH clause to create separate streams for event types E_1, E_2, and E_3 (referred to as event type streams). It then uses SELECT-FROM-WHERE to express the event sequence. In the FROM clause, it applies a RANGE BY construct to each event type stream; the sliding window over the event sequence is automatically captured by having ranges of the window size in each stream. The WHERE clause specifies the equivalence test and the temporal order of E_1, E_2, and E_3 as join predicates.

We set up the TCQ system as follows. We first confirmed that although the TCQ server spans multiple processes, all query processing takes place in a single backend process. Based on this, we plugged in our code to only measure the performance of the query processing backend. We also turned off inter-process communication to make sure that such activity had no effect on our results. Moreover, we made efforts to help TCQ choose the best plan when

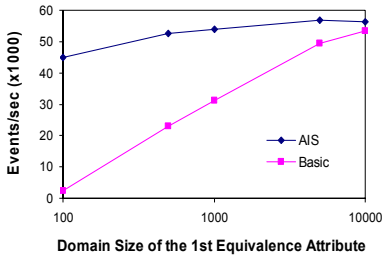


Figure 8: Varying domain size V_1 of the equivalence attribute

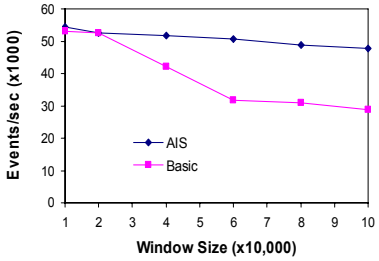
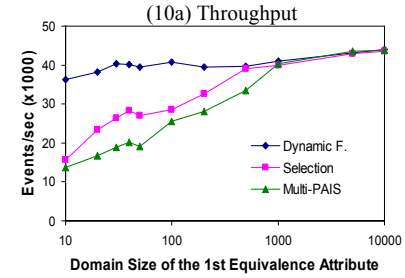


Figure 9: Varying window size W



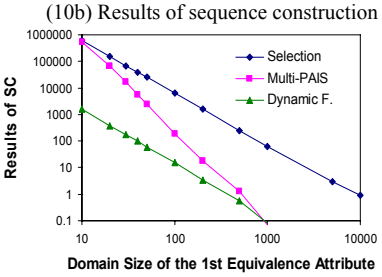


Figure 10: Three strategies for evaluating a 2nd equivalence Test (domain size V_2=20)

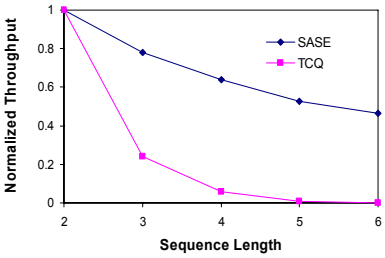


Figure 11: Varying sequence length

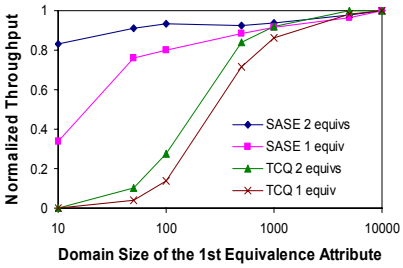


Figure 12: Varying domain size V_1 of $attr_1$ (2nd equivalence test over $attr_2$ with V_2=20)

multiple join predicates are available; we achieved this by providing hints on the most selective join predicate to the optimizer.

The performance metric is still throughput. However, as TCQ and SASE differ significantly in architecture and implementation platform, we do not directly report those numbers in this study. Instead, we report on "normalized throughput" obtained as follows: as the query processing load changes from light to heavy in each experiment, we use the throughput for the lightest workload to normalize other measurements (thus they are all under 1). This approach not only ensures a fair comparison but also allows us to gain insights into tradeoffs between various evaluation strategies.

Experiment 5-Varying sequence length L. In this experiment, we investigate each system's sensitivity to the sequence length by varying L from 2 to 6. We used one equivalence test over $attr_1$ with V_1=100 (larger values of V_1 tend to produce no results for long sequences). W was set to 10,000.

The results are shown in Figure 11. It can be clearly seen that as L increases, SASE scales much better than TCQ. Specifically, TCQ experiences a sharp drop from L=2 to L=3 and degrades to less than 0.01 when L≥5. In contrast, SASE decreases more gracefully and reaches 0.5 with L=6. These results can be explained as follows. As in most stream processors, TCQ uses an n-way join to handle an equivalence test over an event sequence. This certainly incurs high overhead when the sequence length is high. Moreover, TCQ only considers equality comparisons in joins. Therefore, temporal constraints for sequencing, e.g., "s.time > r.time", are evaluated only after the join. In contrast, SASE uses the NFA to naturally capture sequencing of events, and the PAIS algorithm to handle the equivalence test during NFA execution, yielding much better scalability.

Experiment 6-Varying domain size V_1. In this experiment, we compare the performance of SASE and TCQ as query selectivity varies. We set L to 3 and W to 10,000. We first used one equivalence test over $attr_1$ and varied the domain size of $attr_1$ (V_1) from 10 to 10,000. The results are shown by the two curves labeled with "1 equiv" in Figure 12. Note that the measurements were normalized using the throughput for the rightmost point (e.g., 10,000). Figure 12 shows that as V_1 decreases from 10,000 to 10 (from right to left), the performance of TCQ drops much faster than SASE. The benefit of SASE over TCQ stems from its ability to prune more intermediate results. In particular, in the bottom sequence scan operator, SASE uses the NFA to check sequencing of events and the PAIS algorithm inlined with the NFA execution to perform the equivalence test. In contrast, by using a 3-way join and postponing the evaluation of temporal constraints, TCQ suffers from significantly increased intermediate results sizes.

We further added a second equivalence test [$attr_2$] to investigate how the two systems would utilize it. The domain size of $attr_2$ (V_2) was set to 20. The results are shown by the curves labeled with "2 equivs" in Figure 12. Both systems perform better now: SASE significantly improves its performance especially when V_1 is relatively small, whereas the performance gain of TCQ is rather limited. SASE's behavior is attributed to pushing the second equivalence test down to sequence construction, which significantly reduces the number of event sequences that it generates. The way that TCQ handles the second equivalence test is to apply its equality comparisons as selection filters after the corresponding joins for the first equivalence test. In the case of a 3-way join among R, S and T over $attr_1$, assuming that R and S are joined first, the TCQ optimizer is often able to push the filter $R.attr_2 = S.attr_2$ right after the join between R and S, thus reducing the work to be done by the join between S and T. This technique is shown to be less effective than the Dynamic Filtering algorithm in SASE that pushes the entire second equivalence test down to SSC.

The above results imply that a relational stream processor such as TCQ is not designed or optimized for complex event processing. The approach that SASE takes, in particular, using native operators to handle event sequences and highly optimized plans to reduce intermediate result sizes, is indeed necessary. The above results also prove the specific techniques that SASE uses to be effective and scalable for complex event processing.

6 RELATED WORK

Much related work has been covered in the previous sections. We briefly discuss other related work in a broader set of areas below.

Publish/Subscribe. Traditional publish/subscribe systems [1] [5][12][25] provide predicate-based filtering of individual events. SASE significantly extends these systems with the ability to handle correlations among events and transform primitive events into new composite events. Recent work on advanced pub/sub [10] offers an

expressive language to specify subscriptions spanning multiple events, similar to the language in SASE. In comparison, it supports negation in a limited way. Its implementation, based on an NFA-based mechanism, focuses on multi-query optimization but has not addressed issues related to creating composite events as final results and managing intermediate results, whereas SASE uses a large suite of techniques to handle them for good performance.

Sequence databases. SQL-style languages have been proposed to support order in data with a new data model and an order-aware algebra [20], and to support sequence queries that perform time series operations such as computing running aggregates [27][28]. These languages do not offer flexible use of negation. The SEQ system [28] uses relational techniques to implement sequence queries, whereas SASE uses an NFA-based mechanism and many optimizations in this framework to handle event sequences.

Event Processors. A few event processors have been recently developed. CompAS [18] provides a holistic approach to filtering primitive events and detecting composite events. HiFi [13] aggregates events along a tree-structured network on various temporal and geographic scales and has limited support for complete event processing [26]. Siemens RFID middleware [29] offers a temporal data model and declarative rules for managing RFID data but no solid implementation. These systems lack the expressiveness to support our target applications and optimizations for high-volume event processing.

7 CONCLUSIONS AND FUTURE WORK

In this paper, we presented SASE, a complex event processing system that efficiently executes monitoring queries over streams of RFID readings. We first proposed a complex event language that allows queries to filter and correlate events and transform the relevant ones into new composite events for output. The language provides features such as sequencing, negation, parameterization, and windowing necessary for emerging RFID-based monitoring applications. We then presented a query plan-based approach to implementing this language, which uses native operators to construct event sequences while leveraging relational techniques for other processing tasks. We also described a large set of optimizations for handling large windows and reducing intermediate result sizes. We demonstrated the effectiveness of SASE in a detailed performance study. Results of this study show that SASE can process 40,000 events per second for a highly complex query in a Java-based implementation. Results obtained from a comparison between SASE and a state-of-the-art stream processor confirm that SASE's native sequence operators and optimized plans provide much better scalability for demanding workloads.

We plan to continue our research in the following directions. First, we will extend our language by adding aggregates and explore issues related to compositionality. Second, it will be useful to compare SASE to recently developed advanced pub/sub and event processing systems for insights into the strengths of each approach. Finally, for deployment in RFID-based applications, we will also enhance SASE with support for simultaneous queries, disk-based indexing of events, and distributed event processing.

ACKNOWLEDGEMENTS

We thank Mary Fernandez, Michael Franklin, Tyson Condie, and the anonymous reviewers for their valuable comments.

8 REFERENCES

[1] Aguilera, M.K., Strom, R.E., Sturman, D.C., Astley, M., and Chandra, T.D. Matching events in a content-based subscription system. In *Proc. of Principles of Distributed Computing*, 1999.

[2] Arasu, A., Babu, S., and Widom, J. CQL: A language for continuous queries over streams and relations. In *DBPL*, 1-19, 2003.

[3] Bruno, N., Koudas, N., and Srivastava, D. Holistic twig joins: Optimal XML pattern matching. In *SIGMOD*, 310-321, 2002.

[4] Carey, M.J., Livny, M., and Jauhari, R. The HiPAC project: Combining active databases and timing constraints. In *SIGMOD Record*, 17(1), 1988.

[5] Carzaniga, A., and Wolf, A.L. Forwarding in a content-based network. In *SIGCOMM*, 163-174, 2003.

[6] Chakravarthy, S., Krishnaprasad, V., Anwar, E., and Kim, S. Composite events for active databases: Semantics, contexts and detection. In *VLDB*, 606-617, 1994.

[7] Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M.J., Hellerstein, J.M., Hong, W., et al. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

[8] Chandy, K.M., Aydemir, B.E., Karpilovsky, E.M., et al. Event webs for crisis management. In *Proc. of the 2nd IASTED Int'l Conf. on Communications, Internet and Information Technology*, 2003.

[9] Cherniack, M., Balakrishnan, H., Balazinska, M., Carney, D., et al. Scalable distributed stream processing. In *CIDR*, 2003.

[10] Demers, A., Gehrke, J., Hong, M., Riedewald, M., et al. Towards expressive publish/subscribe systems. In *EDBT*, 627-644, 2006.

[11] Diao, Y., Altinel, M., Zhang, H., Franklin, M.J., and Fischer, P.M. Path sharing and predicate evaluation for high-performance XML filtering. *TODS*, 28(4), 467-516, Dec. 2003.

[12] Fabret, F., Jacobsen, H.A., Llirbat, Pereira, J., Ross, K.A., and Shasha, D. Filtering algorithms and implementation for very fast publish/subscribe systems. In *SIGMOD*, 115-126, 2001.

[13] Franklin, M.J., Jeffery, S., Krishnamurthy, S., Reiss, F., Rizvi, S., Wu, E., Cooper, O., Edakkunni, A., and Hong, W. Design considerations for high fan-in systems: The HiFi approach. In *CIDR*, 2005.

[14] Garfinkel, S. and Rosenberg, B. RFID: Applications, security, and privacy. Addison-Wesley, 2006.

[15] Gatziu, S and Dittrich, K.R. Events in an active object-oriented database system. In *Proc of the 1st Int'l Conference on Rules in Database Systems*, 23-39, 1993.

[16] Gehani, N.H., Jagadish, H.V., and Shmueli, O. Composite event specification in active databases: Model and implementation. In *VLDB*, 327-338, 1992.

[17] Galton, A., and Augusto, J. C. Two approaches to event definition. In *Proc. of the 13th Int'l Conference on Database and Expert Systems Applications* (*DEXA*), 547-556, 2002.

[18] Hinze, A. Efficient filtering of composite events. In *Proc. of the British National Database Conference*, 207-225, 2003.

[19] iSpheres. iSpheres EPL server/05 event processing language guide. http://www.ispheres.com.

[20] Lerner, A. and Shasha, D. AQuery: Query language for ordered data, optimization techniques, and experiments. In *VLDB*, 345-356, 2003.

[21] Lieuwen, D. F., Gehani, N., and Arlein, R. The Ode active database: Trigger semantics and implementation. In *ICDE*, 412-420, 1996.

[22] Meo, R., Psaila, G., and Ceri, S. Composite events in Chimera. In *EDBT*, 56-76, 1996.

[23] Michel, C. and Mé, L. Adele: An attach description language for knowledge-based intrusion detection. In *Proc. of the 16th Int'l Conf. on Information Security: Trusted Information*, 353-368, 2001.

[24] Motwani, R., Widom, J., Arasu, A., Babcock, B., Babu, S., Datar, M., et al. Query processing, approximation, and resource management in a data stream management system. In *CIDR*, 2003.

[25] Oki, B., Pfleugl, M., Siegel, A., and Skeen, D. The information bus: An architecture for extensible distributed systems. In *SOSP*, 58-68, 1993.

[26] Rizvi, S., Jeffery, S.R., Krishnamurthy, S., Franklin, M.J., Burkhart, N., et al. Events on the edge. In *SIGMOD*, 885-887, 2005.

[27] Sadri, R, Zaniolo, C, Zarkesh, A., et al. Expressing and optimizing sequence queries in database systems. *TODS*, 29(2), 282-318, 2004.

[28] Seshadri, P., Livny, M., and Ramakrishnan, R. The design and implementation of a sequence database system. In *VLDB*, 99-110, 1996.

[29] Wang, F. and Liu, Peiya. Temporal management of RFID data. In *VLDB*, 1128-1139, 2005.

[30] Zimmer, D. and Unland, R. On the semantics of complex events in active database management systems. In *ICDE*, 392-399, 1999.