Event Correlation and Pattern Detection in CEDR

Roger S. Barga[1] and Hillary Caituiro-Monge[2]

[1] Microsoft Research, One Microsoft Way, Redmond WA
barga@microsoft.com
[2] UC Santa Barbara, Computer Science Department, Santa Barbara, CA
caituiro@cs.ucsb.edu

Abstract. Event processing will play an increasingly important role in constructing distributed applications that can immediately react to critical events. In this paper we describe the **CEDR** language for expressing complex event queries that *filter* and *correlate* events to match specific patterns, and transform the relevant events into new composite events for the use of monitoring applications. Stream-based execution of these standing queries offers instant insight for users to see what is occurring in their systems and to take time-critical actions.

1 Introduction

"By 2008 event processing will be mainstream, with most new business systems in large companies set up to emit vast amounts of event information. Applications are going to start to get very chatty..." *David McCoy, Gartner Group 2005*

A vendor receives a purchase order from a customer, a physician orders a medication change for a patient, an investor executes a trade, a new employee joins a company, a retail outlet makes a sale. Every kind of business is driven by a myriad of such events that occur in its environment. It is not an exaggeration to say that business events are the real drivers of business processes because events represent changes in the state of the business. This is not to say that enterprise web services are not key participants in business processes—they are, but as providers of data and function not as drivers of the process state. Events are also central to creating views of business processes, whether automated or not.

Over the last few years a great deal of attention has been focused on making service level interactions uniform across the industry, such as those based on the WS-* standards, and much progress has been made. Far less attention has been paid to the variety of ways in which events are defined, handled and propagated. Major platform vendors offer event brokers as a feature of their message oriented middleware, and also offer business activity monitoring (BAM) as a feature of their business process stack. But, as in the case of data management in pre-database days, every usage area of events today tends to build its own special purpose infrastructure for defining, handling and propagating events. We see the need for a common event fabric running across web services. Such fabric should allow event sources to fire events "into the event cloud", without caring who consumes them, the subscribers to events can register "standing queries" describing what they are interested in. The queries can be as

simple as simply filtering events or require that events be *filtered* and *correlated* for pattern detection, and that these events are then *transformed* to composite events that reach a semantic level appropriate for end applications. These requirements constitute a unique class of queries that perform real-time transformation of events describing a physical world into information more useful to end applications and users.

2 Events, Messages and Data

An event is a record of some occurrence in the physical or digital world. Each event explicitly (as part of its data) or implicitly (interpretation rule) has a type that indicates what happened, and schematized data that describes the details. A message is a means of transmitting data from a source to one or more targets. A message also typically has a type and schematized data. It is thus natural to ask if the two are different. The difference between them is not in form but in semantics. Unlike a message, an event has neither a targeted receiver nor an interaction pattern—there is no such thing as a reply to an event. Instead, the important part of the semantics of an event is causality – what events caused this to occur and what will be the consequences of the current event. This *causality* is not in the event itself – it can only be inferred interpretation of the payload of multiple events, possibly from different sources.

Of course, events can be caused by messages and messages are used to transmit events to interested observers and processors. We illustrate the message/event relationship and idea of causality with an example. A purchasing subsystem receives a shipment status message. The status code and details inform the recipient that a shipment is delayed by 3 weeks. The purchasing subsystem is configured to generate a specific type of event when processing shipment delay messages. There are event transformation services installed in the "event fabric" to process events of this type based on the reference numbers in the content by fetching relevant application data to transform the event into a more meaningful event such as "delivery for Intel processor parts order 72134 delayed by 3 weeks". This same type of "parts delivery delayed" event can also occur as a result of manual action by a purchasing agent receiving a phone call or FAX, illustrating that different occurrences of the same type of event may have different causality chains. Through a further chain of rule-based event transformations, an event "PC production-line 4 in Memphis at risk of closure in 3 weeks" is generated. The "Memphis PC Production-line Managers" group has an alert subscription for this type of, and the final event generates an alert message sent to the group.

3 Complex Event Queries over Event Streams

In this section we outline the event correlation and pattern detection language for CEDR, short for **C**omplex **E**vent **D**etection and **R**esponse, an event processing system currently under development at MSR.

3.1 From Web Service Message Streams to Event Streams for Query Processing

This discussion outlines how we go from a *physical message stream* flowing over an event bus or WS-Event channel, to a *logical event stream* for processing against a set of standing queries. Our raw input stream consists of messages with an arbitrary but known schema for each message type that provides the necessary metadata information about the *message payload*. Input to our processing system is an infinite sequence of events, referred to as an *event stream*. An event represents an instantaneous and atomic occurrence of interest at a point in time. Similar to the distinction between types and instances in programming languages, our model includes event types that describe a set of attributes that a class of events must contain. Each event instance, denoted by a lower-case letter (e.g., '*e*'), consists of the name of its type, denoted by an upper-case letter (e.g., '*E*'), and a set of values corresponding to the attributes defined in the event type which we refer to as the *payload*. In order to *transform* a message into an event for processing we augment the record with the following fields:

- Start_Valid_Time: the time at which the event instance can be considered for query processing. If an operator in our system is processing an event instance at a time before (less than) this timestamp, this event instance can not yet be considered (seen) by the operator.
- End_Valid_Time: the time at which an event instance can no longer be considered for query processing. This value can be set by a time-to-live (ttl) attribute or sliding window specification, expressed in the query, or by a corresponding delete (retraction) of the event coming into the system.

So, the schema of an event instance for processing is a union of message schemas or possibly some mapping of incoming message type to its associated event type, and temporal schema: <start valid time, end valid time>. Most messages will already have a timestamp from a discrete ordered time domain. We assume timestamps are assigned by a separate mechanism before events enter our system and reflect the true occurrence time of these events in the "real world", so we use this timestamp as start valid time – if an incoming event does not have a timestamp, we assign the value of the system clock. The corresponding end valid time value is set to infinity by default – we assume the fact this event represents will stand true forever, but will adjust depending on the semantics of the query. It is important to note the transformation from a raw message to logical input event preserves the arrival order of event instances.

3.2 CEDR Language

In this section, we present the CEDR language for expressing queries over event streams. The language is declarative and based on three independent aspects: 1) an event pattern expression that identifies event types and order, using operators and constructs that specify how event instances are *filtered*, and how multiple events are *correlated* via time-based and value-based constraints to detect *patterns*; 2) an optional *lifetime* during which the pattern may be detected; 3) an optional *selection* expression that specifies how complex events are constructed from correlated events.

The overall structure of the CEDR language is:

EVENT_NAME <string>
EVENT_PATTERN <event type and order expression>
WHERE < temporal and data qualification predicates>
[LIFETIME <window>]
NOTIFY <selection conditions for the construction of a complex event>

To introduce the constructs in our language, we use examples drawn from systems monitoring in which operations management monitors events stemming from network servers, database servers and network devices across the company. The first query (Q1) looks for events that indicate a software upgrade is being installed on a machine.

Q1: UPDATE_MACHINE
EVENT_PATTERN **INSTALL**
WHERE (*software_type* = 'SP' AND *version_id* = '2')

In Q1, the EVENT PATTERN clause contains a filter for ***event type*** "*INSTALL*" that retrieves only event instances of the *INSTALL* type from the input stream. The WHERE clause further filters these event instances by evaluating two ***predicates*** applied to their attributes: the first predicate requires the value of attribute *software_type* to be 'SP' and the second predicate requires the value of attribute *version_id* to be '2'. In general, the WHERE clause is responsible for *filtering* event instances and can be a boolean combination, using logical connectives AND and OR, of predicates that use one of the six comparison operators ($=, \neq, >, <, \geq, \leq$).

Our second example, Q2 detects a failed software upgrade by filtering the event stream to retrieve only those events that report an upgrade was installed on a machine, followed by a *shutdown* without the occurence of a subsequent *restart*. The EVENT clause of this query contains a SEQUENCE construct that specifies events must occur in a particular order; the components of the sequence are the occurrences and non-occurrences of events of interest. In this query, the SEQUENCE construct specifies the occurrence of an *INSTALL* event followed by a *SHUTDOWN* event, and the non-occurrence of a *RESTART* within a fixed window of time. Non-occurrences of events are expressed using the **NOT** operator and bounded by a time interval, expressed by the WITHIN operator. For the use of subsequent clauses, the SEQUENCE construct also includes a variable in each component to refer to the corresponding event.

Q2: FAILED_UPGRADE
EVENT_PATTERN **SEQUENCE**(*INSTALL x*, *SHUTDOWN y*,
NOT (*RESTART z*, **WITHIN** 5 minutes))
WHERE *((x.machine_id = y.machine_id)* AND *(y.machine_id = z.machine_id))*
/* shorthand for this test is **CorrelationKey**(machine_id, Equal) */

The WHERE clause in Q2 uses variables defined previously to form predicates that compare attributes across different events. One of the more powerful aspects of any

language for event pattern detection is *correlation*, the ability of an event instance to be linked to specific instances of other events through payload values. To distinguish this from simple predicates that compare to a constant, such as those in the first example, we refer to such predicates as ***parameterized predicates.*** The parameterized predicates in Q2 compare the *machine_id* attributes of all three events in the SEQUENCE construct for equality. Since this is such a common comparison, we introduce shorthand to simplify writing such parameterized predicates.

Equality comparisons on a common payload attribute across entire event sequences are typical in distributed monitoring applications. For ease of use, we refer to the common attribute used for this purpose as a *correlation key*, and the set of comparisons on this attribute as the ***similarity test*** that checks for equality or inequality. Our language offers simple shorthand for an equivalence test on common attributes, specifically **CorrelationKey**(*attribute*, *Equal* **|** *Unique*). In addition, we offer predefined operators for temporal correlation, such "**BEFORE**", "**AFTER**", "**EQUAL**", "**WITHIN**" which automatically extract the timestamp field from all event instances.

In CEDR an optional LIFESPAN clause is used to define a temporal interval during which a query is active or "*of interest*" and may be detected. In our model, a lifespan is either *Active* or *Inactive* and is controlled by two events, referred to as *initiator* and *terminator*. An occurrence of an initiator event activates the lifespan, while the occurrence of a terminator event deactivates the lifespan. A lifespan can also expire, as defined by its maximal duration. While *active*, any query associated with the lifespan is also *active* and CEDR will attempt to detect the pattern in the event stream.

Lifespan Pattern
Name – *name of the lifespan pattern*
Initiator – *conditions for lifespan initiation*
Event: identifies the initiating event;
Condition: optional conditions the event must satisfy;
Terminator – *describes conditions for lifespan termination*
Event: identifies the initiating event;
Condition: optional conditions the event must satisfy;
Duration: max duration the lifespan will remain active;

A lifespan can be used to implement *state based* pattern detection – when the system is in a particular state, as indicated by the activation of a lifespan, a select set of queries is activated. Each state can define a unique set of queries (events to watch for) and filters to eliminate (drop) redundant events. In short, LIFESPAN enables a system to compartmentalize functionality and provide context for event processing. CEDR is not the only system to include lifetimes – the concept is identified in [2] and implemented in AMIT [22] and Rapide [1].

Given a sequence of event instances as input, the output of a standing query in CEDR is also a sequence of event instances, where each output event instance represents a unique match or *detection*. The NOTIFY clause allows a programmer to specify ex-

actly how the complex event that signifies detection is composed. Using query Q2 for example, a resulting event can be created to indicate an upgrade has failed to install on the machine. Unlike previous work that focuses only on the detection of standing queries, but not reporting how the query was actually satisfied, we provide the means to explicitly report the events used to match the query. This significantly increases the complexity of the underlying runtime, since it must accurately track event instances.

In CEDR, the NOTIFY clause can specify for each *event type* in the pattern expressing which individual *instances* of this event type are to be chosen. CEDR supports four options, which can be specified for each operand in the pattern expression: *First*, *Last*, *Each* and *Cond*; the default is *First*. *First* and *Last* are minimum-oriented selection strategies [2]. *First* (*Last*) selects from the event collection instances with the oldest (youngest) timestamps. *Each* selects all permissible event instances associated with the operand, possibly resulting in the creation of multiple complex events after pattern detection. The final option, *Cond*, is a user supplied predicate that is applied to each event instance associated with the operand – only if the condition is satisfied will the instance be selected.

In example Q2, the pattern expression includes a sequence operator and two operands (event types) to detect an **INSTALL** event followed by a **SHUTDOWN** event. There may be several instances of the install event, each indicating installation of a different patch, before a shutdown is detected. If the pattern indicating a flawed upgrade is detected, the user may wish to raise a complex event identifying *all software patches* installed, which can be easily accomplished using the selection condition *Each*. We hasten to note, typically NOTIFY creates an instance of an event type and selection conditions are used to *bind* variables for composing this new event instance (marshaling the payload for the new event instance) but we omit this discussion for brevity.

Q2: FAILED_UPGRADE
EVENT_PATTERN **SEQUENCE**(*INSTALL x*, *SHUTDOWN y*,
NOT (*RESTART z*, **WITHIN** 5 minutes))
WHERE **CorrelationKey**(machine_id, Equal)
NOTIFY *Each* x, *First* y

The simple examples presented in this section were intended to highlight constructs of CEDR. Our language builds on event languages [20, 16, 2] originally developed for active databases, triggers [17] and continual queries [19]. In addition to event constructs such as *sequence*, CEDR offers additional features, such as i) the use of negation in event sequences, ii) *parameterized predicates* in the WHERE clause for correlating events via value-based constraints and iii) *time to live (TTL) and various sliding windows for event instances* in the LIFETIME clause for expressing additional temporal constraints for pattern detection. Finally, CEDR enables event composition, which allows the output of one query to be used as input to another. The fact that a CEDR query can take a sequence of input events and produce a sequence of composite (complex) events as output enables full compositionality. Together these features provide rich functionality and control over event pattern specification and detection.

3.3 Semantics of the Language

In this section, we present the semantics of CEDR by translating selected language constructs to algebraic query expressions. Each event type E_i is a query expression. An event operator connects 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*. For example, the semantics of a base expression E_i, represented as $E_i(t)$, is that at a given point t in time, $E_i(t)$ is *True* if an E_i type event occurred at t, and is *False* otherwise. Below, we describe the set of operators that CEDR supports and the semantics of expressions that they form.

Event Sequencing (Order) – The ability to synthesize events based upon the ordering of previous events is a basic and powerful event language construct.

Operator	Description
ALL [E1 . . . ,Ek]	A conjunction of events $E1. . .Ek$ with no order importance. It takes a set of event types as input and evaluates to *True* if instances of each event type occur. Formally, defined as: $\mathbf{ALL}(E1, E2, \ldots, En)(t) \equiv \forall\ 1 \leq i \leq n\ E_i(t)$. It outputs all instances that occurred at time t as a result.
ANY [E1...,Ek]	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: $\mathbf{ANY}(E1, E2, \ldots, Ek)(t) \equiv \exists\ 1 \leq i \leq k\ E_i(t)$. It outputs the event that occurred at time t as a result.
SEQUENCE [E1… Ek]	SEQUENCE takes a list of n (n>1) event types as its parameters – these parameters specify a particular order in which events must occur. Arbitrary events may appear between any two events in the sequence. The formal definition is: $\text{SEQUENCE}(E1, E2, \ldots, En)(t) \equiv \exists\ t_1<t_2<\ldots<t_n\ \bullet\ E_1(t_1)\wedge E_2(t_2)\wedge\ldots\wedge E_n(t_n)$. The operators ANY and ALL can be used inside a SEQUENCE, e.g., $\text{SEQUENCE}(E_1, \text{ANY}(E_{21}, \ldots, E_{2m}), \ldots)$. The semantics of the resulting expressions are defined by combining semantics of SEQUENCE along with ANY (or ALL).

Counting – Counting operators reduce the number of events flowing to the application, thus reducing the amount of state required in the process. State is maintained by CEDR and there is value in this separation. For example, a "valuable customer" pattern may be triggered after three consecutive purchases. To change it to four purchases requires simply changing the event pattern definition – not the business application.

Operator	Description
ATLEAST [n, E1...Ek]	A minimal conjunction of *n* events out of *E*1. . .*Ek* with no order importance. ATLEAST takes a list of k (k>1) event types as its parameter, along with an integer n, and evaluates to *True* if n or more of these k events occur. **ATLEAST** (n, *E*1, *E*2,…, *Ek*)(*t*) ≡ $\Sigma\ 1 \leq i \leq k\ E_i(t) \geq$ n. It outputs all event instances true at time *t* as a result.
ATMOST [n, E1...Ek]	A maximal conjunction of *n* events out of *E*1. . .*Ek* with no order importance. While the definition of **ATMOST** follows directly from the definition of **ATLEAST**, an important difference is that some temporal expression must be supplied to cancel the accumulation of state for patterns that use this operator.

Absence – The event service can track the *non-occurrence* of an expected event, such as a customer not answering an email within a specified time. So-called *non-events* have great utility in business processes. Further, set operations apply to non-events: for example, events indicating that *none* of a group of customers has responded, or *all* or some *threshold number* of customers has answered are valid and useful events.

Operator	Description
NOT [E1,…,Ek, scope]	None of the events *E*1…*Ek* has occurred within the detection lifespan. The **NOT** operator requires an explicit bound on the detection window, which can be specified using an optional **SCOPE** clause, which specifies the amount of time in the detection lifespan.
UNLESS [E1, E2]	Designates the occurrence of the first operand and non-occurrence of the second within the detection lifespan.

Constraints and Event Cancellation – Event patterns normally do not "pend" indefinitely; conditions or constraints may be used to cancel the accumulation of state for a pattern (which would otherwise remain to aggregate with future events to generate a composite event). Such temporal constraints limit event generation to within a particular time window. The CANCEL-WHEN operator is used to describe such constraints. CANCEL-WHEN may be followed either by a temporal expression *time-expr*, and is commonly used with a timer event but may use any other base or pattern event, including patterns that mix timer events with other events (e.g., Approval & T [Approval.t + 20hours]).

Operator	Description
E **CANCEL-WHEN** *EXPR*	Used to cancel or invalidate specific event instances or a complete sub-expression in a pattern query. This effectively removes the event instances being held in the event collection for this pattern.

Use of **CANCEL-WHEN** has a number of interesting applications, which we touch on only briefly. Since multi-term patterns rely upon the asynchronous delivery of multiple events, it is possible that one required event will arrive but the others will not. To avoid the unbounded accumulation of event instances (state) temporal constraints are used to remove them; normally this is thought of as a form of garbage collection. In this case the **CANCEL-WHEN** predicate is used to bound the valid time of an event pattern. As an example: ((E & F) **CANCEL-WHEN** *time-expr*). In this case event E and event F *must* occur before the value in *time-expr* is reached.

The use of **CANCEL-WHEN** can be generalized to include other sorts of pattern cancellation; for example: (E & F **CANCEL-WHEN** G). In this case if event G arrives before both E and F then event pattern detection is aborted.

4 Pattern Detection and Event Processing

In this section, we illustrate the functionality of components in CEDR by walking thru the processing of events and pattern detection. The initial stage is *event filtering*. For each incoming event, the CEDR *filter manager* will determine if it has the potential to impact an active pattern. While at any given time the total number of active pattern instances can be quite large, typically the number affected by an incoming event instance is orders of magnitude lower. We leverage this to construct a restrictive filter on event instances and fine tune filtering by pushing *event instance conditions* into the filter. In the system monitoring application, for example, we filter event instances using event type = **install** with context (instance condition) = **SP2**. Once detected, the filter is updated to include event type = **shutdown** with context = **MachineID**, where the value of **MachineID** is extracted from the **install** event instance. We also use indexes to efficiently identify pattern instances affected by the incoming event.

The decision process in CEDR to determine whether or not an event pattern has actually been detected is divided into three separate stages, where each stage is based on a separate aspect of the pattern language definition.

Stage 1: Event Collection – An *event collection* designates the event instances that are considered for pattern detection, if they occur while the pattern is active. In this stage all event instances that may contribute to an active pattern are collected. An instance *contributes* to a pattern if the event type matches an operand type in the pattern expression, and satisfies all conditions defined for the operand. Each candidate in

the event collection is associated with the operand to which it matches and forms a *candidate list* for the operand. An event instance may contribute to more than one active patterns, so to avoid storing multiple copies CEDR maintains a shared collection of event instances and stores only a reference to the instance in the candidate list; reference counting is used to manage (clean up) this shared collection. To decide if the pattern is satisfied it is sufficient to base detection on these event instances only.

Stage 2: Detection – The decision about whether a pattern occurred is based on a combination of operators and event instance conditions. It is possible that multiple event instances of the same event type satisfy the detection conditions.

Stage 3 Selection – When a pattern is detected, instances that contributed to detection are *selected* from the event collection following the pattern's *selection policy* and used to create the composite event. This new event is then published (broadcast). Finally, the event instances that triggered the detection are removed from the candidate list of that operand. CEDR actually supports a range of consumption policies [2] to update the event instance collection, but this is outside the scope of this paper.

To summarize, filtering permits only select event types to pass into the detection engine. While a standing query is active, all event instances that may contribute to a pattern are collected (step 1). If all conditions that define the standing query have been met (step 2), the event instances in the collection that triggered detection are selected to generate a new complex event and removed from the collection (step 3).

5 Related Work

Throughout the paper we have attempted to point out closely related work, so in this section we briefly discuss other related work in a broader set of areas.

A number of event processing systems have been recently developed. HiFi [7] aggregates events in a tree-structured network on various temporal and geographic scales and offers limited support for complete event processing [9]. Siemens RFID middleware [10] offers a temporal data model and declarative rules for managing RFID data but no solid implementation is described. Overall, these systems lack the expressiveness to support our target applications, which is distributed monitoring.

Conventional publish/subscribe systems [4][6] support predicate-based filtering of individual events. CEDR extends this approach with the ability to handle both temporal and data value correlations among events and transform primitive events into a new composite event. More recent work on enhanced publish/subscribe [5] provides expressive language support to specify subscriptions spanning multiple events, similar to the language in CEDR. However, it supports the absence of events, or negation, in a rather limited way. Moreover, Cayuga does not address issues related to creating composite events as a result of detection and managing event instance state, whereas CEDR exposes commands and techniques to compose complex events after detection.

In a broader context, our system is related to sequence databases [17] since raw input streams are a temporally ordered sequence of records. However, the semantics of SQL-style sequence languages includes one-time, but not continuous queries. The chronicle data model [16] provides operators over relations and chronicles, which can be considered as a raw input stream, but focuses on the space complexity of an incremental maintenance of materialized views over chronicles; it does not include continuous queries or aspects of data-driven processing. None of these offer the flexible use of negation. In the context of continuous queries over streams, there has also been considerable research. Tribeca [18] introduces fixed and moving window queries over single network streams. TelegraphCQ [14] defines a declarative language to express a sequence of windows over a stream. Aurora [11, 13] builds a query graph of stream operators parameterized by functions and predicates while abstracting from a certain query language. The Tapestry system [19] transforms a continuous query into an incremental query that is run periodically, which ensures snapshot-reducibility but can not detect patterns (sequences) in the event stream. We refer the interested reader to [12, 15] for a broader overview on data stream processing.

6 Conclusion

Event processing will play an increasingly important role in constructing distributed enterprise applications over web services that can immediately react to critical events. In this paper we have illustrated the need for a common event fabric across web services. This fabric will allow any event source to contribute its events into the "event cloud". Subscribers can register their interest in receiving both the raw events from sources and the result of complex standing queries against events streaming through the cloud. In this paper we introduced CEDR, a runtime service for event correlation and pattern detection. Our presentation outlined CEDR's language for defining event patterns, along with its event processing and detection model. CEDR builds on and extends previous work on event processing in several directions. It introduces a declarative language for specifying patterns that includes high level event operators, support for detection lifetime and flexible instance selection and consumption conditions. Our detection model, currently built using nondeterministic finite automata (NFA) supports a time model to manage temporal correlations across event instances and timeouts, and can efficiently handle predicates in patterns. This paper represents a current snapshot of our design and language development.

References

1. Luckham, David "The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems", Addison Wesley Publishers, 2002.
2. Zimmer D, Unland R On the semantics of complex events in active database management systems. In the Proceedings of ICDE, March 1999, pp 392–399.

3. RS Barga and H. Caituiro-Monge, CEDR – a runtime service for event correlation and pattern detection, ACM Middleware'05 Conference, *demo presentation and short paper*.
4. Aguilera, M.K., Strom, R.E., Sturman, D.C., Astley, M., and Chandra, T.D. Matching events in a content-based subscription sys-tem. In *Proc. of Principles of Distributed Computing*, 1999.
5. Demers, A., Gehrke, J., Hong, M., Riedewald, M., et al. Towards expressive publish/subscribe systems. In *EDBT*, 627-644, 2006.
6. 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.
7. 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.
8. Hinze, A. Efficient filtering of composite events. In *Proc. of the British National Database Conference*, 207-225, 2003.
9. Rizvi, S., Jeffery, S.R., Krishnamurthy, S., Franklin, M.J., Burkhart, N., et al. Events on the edge. In *SIGMOD*, 885-887, 2005.
10. Wang, F. and Liu, Peiya. Temporal management of RFID data. In *VLDB*, 1128-1139, 2005.
11. D. J. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. B. Zdonik. Aurora: A New Model and Architecture for Data Stream Management. VLDB Journal, 12(2):120–139, 2003.
12. A. Arasu, S. Babu, and J. Widom. The CQL Continuous Query Language: Semantic Foundations and Query Execution. Technical report, Stanford University, 2003.
13. D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. B. Zdonik. Monitoring Streams: A New Class of Data Management Applications. In Proc. of the Conf. on Very Large Databases (VLDB), pages 215–226, 2002.
14. S. Chandrasekaran, O. Cooper, and A. D. et al. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In Proc. of the Conference on Innovative Data Systems Research (CIDR), 2003.
15. L. Golab and M. T. Ozsu. Issues in Data Stream Management. SIGMOD Record, 32(2):5–14, 2003.
16. H. V. Jagadish, I. S. Mumick, and A. Silberschatz. View Maintenance Issues for the Chronicle Data Model. In Proc. of the ACM SIGMOD, pages 113–124, 1995.
17. P. Seshadri, M. Livny, and R. Ramakrishnan. The Design and Implementation of a Sequence Database System. In the Proceedings of the Conference on Very Large Databases (VLDB), pages 99–110, 1996.
18. M. Sullivan and A. Heybey. Tribeca: A System for Managing Large Databases of Network Traffic. In In Proc. of the USENIX Annual Technical Conference, pages 13–24, 1998.
19. D. B. Terry, D. Goldberg, D. Nichols and B. M. Oki. Continuous Queries over Append-Only Databases. In Proc. of the ACM SIGMOD, pages 321–330, 1992.
20. Chakravarthy, S., Krishnaprasad, V., Anwar, E. and Kim, S. Composite events for active databases: Semantics, contexts and detection. In *VLDB*, 606-617, 1994.
21. Gehani, N.H., Jagadish, H.V., and Shmueli, O. Composite event specification in active databases: Model and implementation. In *VLDB*, 327-338, 1992.
22. Opher Etzion, Complex Event Processing. In the Proc. of the Intl Conf on Web Services (*ICWS*) 185-197, 2004.