Complex Event Processing Beyond Active Databases: Streams and Uncertainties

Shariq Rizvi

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2005-26
http://www.eecs.berkeley.edu/Pubs/TechRpts/2005/EECS-2005-26.html

December 16, 2005

Acknowledgement

The author is thankful to his research advisor, Prof. Michael Franklin, for his guidance throughout this work. Dr. Kurt Brown and Dr. Minos Garofalakis provided valuable insights during the work on Probabilistic Complex Event Processing. Members of the Telegraph and HiFi project groups at Berkeley were consistent sources of help during the work on implementing event processing in the TelegraphCQ system. Finally, the author is thankful to all the members of the Berkeley Database Group.

Complex Event Processing Beyond Active Databases: Streams and Uncertainties

Shariq Rizvi

Master's Research Project

Abstract

Complex Event Processing deals with aggregating simple events, which are defined as "occurrences of significance in a system" [20], to get semantically-richer events usable by an end application. They have been studied earlier in multiple disparate contexts, for example, in the Active Database community, under ECA rules, that are used to build triggers for a variety of purposes.

A resurgence of interest in Complex Event Processing research has taken place because of recent advances in sensing technologies like sensornets and RFID. This technology generates events out of real-world inputs such as the movement of people in a home or items in a supply chain. Our work focuses on Complex Event Processing in the context of such real-world event sources, and on two challenging dimensions that arise: streaming data and fuzzy or probabilistic data. We present the notion of *semantic windows*, which go beyond time-based or tuple-based windows proposed for streaming data processing. Probabilistic Complex Event Processing (PCEP) allows applications to reason about and respond to events in scenarios where simple events cannot be monitored in a crisp fashion.

Ideas from this work have been implemented in the TelegraphCQ streaming data processor, and used to drive the core functionality of an event-driven library scenario in a recent system demonstration.

1 Introduction

An *event* is defined as an "occurrence of significance" [20] in a system. This may be the insertion of a tuple into a table in a database, a connection request through a network socket, or the displacement of an item in a store.

Multiple research initiatives have looked at event processing for different reasons. The Active Database community focused on the use of events in "triggers" that take an action based on a situation of interest [32]. For example, the insertion of a new tuple into a table may be seen as an event of interest when a materialized view based on table needs to be maintained accurately and automatically. At the same time, the insertion of this new tuple should not violate the integrity constraints (say, key-foreign key constraints) that are defined on the table. Finally, accurate query optimization may require up-to-date statistics about the table in the catalogs, which needs some work when the insertion takes place. All these practical scenarios require a sense-and-respond functionality inside the database to react to important events, and have been formalized as the extensively-studied ECA rules [31]. It is fair to say that the notion of events in the ECA world has been limited to in-database events, like insertion of tuples, beginning of transactions, and calls to an object's method in an OODBMS.

Recent interest in Complex Event Processing has been triggered by the increasing complexity of real-world processes and distributed systems. For example, The CEP initiative [20] from a group at Stanford tries to reason about the events in a complex system by building the abstraction of a hierarchy of events. The goal is to help the debugging and analysis of these otherwise complex systems.

Resurgence of Complex Event Processing research in the database community [27, 16] has taken place in the context of streaming data from real-world sources like sensornets and RFID. The data from such sources can capture physical events, such as the movement of people and objects in scenarios like a home or a supply chain. This allows Complex Event Processing to detect high-level events, like *shoplifting* and *under-stocking*, which are not available directly in low-level streams coming from sensor sources.

Our work is a fresh take on Complex Event Processing, as required under two new dimensions added to data management - streaming data and fuzzy or probabilistic data. As mentioned earlier, the need to handle streams comes up because of the emergence of data sources like sensornets. We introduce the notion of *semantic windows* over streams, that go beyond the current notion of time-based or tuple-based windows. The need for *Probabilistic Complex Event Processing* comes up because of the nature of the data that is reported by real-world sources. Such data is often *dirty* and *unreliable* [14, 15]. Processing of complex events over such streams requires a new model of (simple and complex) events that can accommodate their uncertain nature. Further, the need for probabilistic processing also comes up when simple events are reported at different levels of semantic detail. For example, in a smart home, simple motion sensors may detect the movement of *persons*, while more sophisticated video-based vision techniques may report events in terms of *specific persons*.

The contributions of our work can be listed as follows.

1. A comprehensive survey of Complex Event Processing work in multiple contexts,

including Active Databases.

2. Design, implementation, and evaluation of a system for Complex Event Processing over streaming data.

3. An architecture for Probabilistic Complex Event Processing (PCEP).

This report is organized as follows. Section 2 surveys past work in the closely related areas of Active Databases and CEP. Section 3 presents new work on event processing over streaming data, including the concept of semantic windows. This work has been done in the context of the HiFi project at Berkeley [11, 9], and implemented as extensions to the TelegraphCQ [7] streaming data processor. The resulting system was used as the core processing engine in a recent demonstration at SIGMOD 2005 [27]. Section 4 goes into Probabilistic Complex Event Processing. Section 5 looks at some other previous work that is related to Complex Event Processing, but not directly applicable - query processing for sequence data. Finally, Section 6 concludes the report and presents directions for future work.

2 Previous Work on Event Processing

Many research initiatives in the past have looked at aggregating simple events into more complex and semantically higher-level events. The work from the Active Database community has focused on *in-database* events, and triggers built on them. The more recent CEP initiative has approached the problem as a means of understanding and debugging complex distributed systems, like enterprise systems. We survey both in this section.

2.1 Complex Events in Active Databases

Unlike *passive* databases, which store, manage and process data, *active* databases support applications that want to automatically respond to certain events and changes inside the database. These in-database events may range from simple events like insertion of a tuple into a relation or committing of a transaction, to more complex events that are specified over *event histories* [13] using a complex event algebra.

In our survey, we first present some basic terminology, followed by two representative pieces of work that have proposed *composite* event processing algebras. These are COMPOSE [13], and SNOOP [6]. This is followed by an overview of the work presented by Zimmer and Unland [32], which is a framework for complex event processing algebras.

2.1.1 Basic Terminology

Primitive Event: A primitive event is the smallest, atomic occurrence in a system that may require a response. By *atomic*, we mean that either the event happens completely or it does not happen at all.

Event Type and Event Instance: Similar to the schema of a relation, an event type gives the metadata for events that belong to the same *class*. This includes the attributes of these events. An *event instance* is a single occurrence of an event of a particular type. This instance *instantiates* the attributes of the event type.

Complex Event Type: A complex event type is the result of applying operators from an algebra to the simple (or recursively, complex) event types.

Event History: An event history is a partially ordered set of event instances, where the order reflects their occurrences in time.

While this set of definitions suffices for the discussion in this paper, a more elaborate list for the active database work can be found in [32]. Note that, although the timestamp for simple events is reported by the underlying system, complex events also need to be given timestamps, to obtain the closure property for operators.

Most previous work attaches a point timestamp to every event (including complex events). Under such an approach, complex events are usually given the timestamp of their terminating component event. This approach can lead to unexpected behavior. For example, suppose we are looking for a complex event that is defined as "E_1 followed by E_2" (as we will see later, sequencing is a fundamental operation on events). Using the point timestamp approach, the complex event will be detected positively as long as the last component event in E_1 occurred before the last component event in E_2. However, if E_2 has multiple component events, it may be possible that the first component event in E_2 occurred before some component events in E_1. Hence, E_1 did not occur *fully before* E_2. This may not be the semantics that the application intended. This problem has been recently addressed by having interval-based timestamps for complex events [3]. However, the active database work surveyed in this report, and also our work on Complex Event Processing over streams, is based on point timestamps for complex events.

2.1.2 The COMPOSE Algebra

The COMPOSE composite event processing language comes from the Ode system [13]. The event history is assumed to be a finite set of *event occurrences*. An event occurrence is a tuple of the form *(primitive event, event identifier).* Event identifiers are used to define a total ordering on event occurrences (a timestamp is one example of an event identifier). In a history, no two events can have the same identifier.

An event expression E is a mapping from one history to another. When an event expression is applied to a history h, we get a history which contains the events in h at which the event specified by E takes place. $E[h]$ denotes the application of E to the history h. By definition, $E[h] \subseteq h$. Event expressions are formed by using the null expression $NULL$, any primitive event a, and the operators described below.

1. $E[null] = null$, $\forall E$, where *null* is the empty history.

2. $NULL[h] = null$

3. $a[h]$ is the maximal subset of h composed of event occurrences of the form (a, eid).

4. $(E \wedge F)[h] = E[h] \cap F[h]$.

   This captures the "simultaneous" occurrences of E and F (having the same event identifier). Note that although two simple events cannot have the same identifier, two complex events may (under point semantics).

5. $(\neg E)[h] = (h - E[h])$.

   This captures those simple event instances which are not of type E. Note however, that this expression is not equivalent to looking for "absence of E" but rather looking for "an event occurrence that is not E".

6. $relative(E, F)[h]$ are the event occurrences in h at which F is satisfied assuming that the history started immediately following some event occurrence in h at which E takes place. Formally, if $E^i[h]$ is the i^{th} event occurrence in $E[h]$ and h_i is obtained from h by deleting all event occurrences whose *eids* are less than or equal to the *eid* of $E^i[h]$, then:

$$relative(E, F)[h] = \cup_{i=1}^{|E[h]|} F[h_i]$$

7. $relative^+(E)[h] = \cup_{i=1}^{\infty} relative^i(E)[h]$

   where,

   $relative^1(E) = E$; and

   $relative^i(E) = relative(relative^{i-1}(E), E)$

   This operator can be interpreted as trying to find a pattern of unlimited length over an event history.

The authors claim that this event expression language has the same expressive power as regular expressions. Other useful operators can be expressed using these basic operators. For example, sequencing and repetition operators:

- $\mathit{prior}(E, F)[h] = (\mathit{relative}(E, \mathit{any}) \wedge F)[h]$

  which captures occurrences of F following those of E.

- $\mathit{sequence}(E, F)[h] = (\mathit{relative}(E, \neg \mathit{relative}(\mathit{any}, \mathit{any})) \wedge F)[h]$

  which captures occurrences of F immediately following some occurrence of E in the event history.

A complete list of the additional operators, and algorithms for constructing a finite state automaton for detecting events expressed in this language, can be found in [13].

2.1.3 The SNOOP Algebra

The SNOOP [6] composite event language consists of conjunction ($\triangle$), disjunction (∇), sequence (;), negation, and some other operators described below. An event expression E is regarded as a function from the underlying time domain onto boolean values. For a given point in time t, E computes to *true*, if an event of type E occurred at t. Otherwise, it evaluates to *false*. $\neg E$ denotes the negation function of E, and expresses the non-occurrence of E at a given point in time.

The event operators are as follows:

1. $(E_1 \nabla E_2)(t) = E_1(t) \vee E_2(t)$

   This captures those points in time, where at least one of E_1 and E_2 occurs.

2. $(E_1 \triangle E_2)(t) = ((\exists t_1)(E_1(t_1) \wedge E_2(t)) \vee (E_2(t_1) \wedge E_1(t)) \wedge t_1 \leq t)$

   This captures those points in time where an instance of E_1 occurs, E_2 having occurred earlier (or at the same instant in time), or vice versa. Note the difference between SNOOP's $\triangle$ operator and the $\wedge$ operator from COMPOSE. The latter looks for instants in time where E_1 and E_2 occur together, while the current operator looks for those instants where both E_1 and E_2 have already (just) occurred.

3. $ANY(m, E_1, E_2, ..., E_n)(t) = (\exists t_1)(\exists t_2)...(\exists t_{m-1})(E_i(t_1) \wedge E_j(t_2) \wedge ... \wedge E_k(t_{m-1}) \wedge E_p(t) \wedge (t_1 \leq t_2 ... \leq t_{m-1} \leq t) \wedge (1 \leq i, j, ..., k, p \leq n) \wedge (i \neq j \neq ... \neq k \neq p))$, with $m \leq n$

   This operator looks for the occurrence of *exactly* m out of n events in time.

4. $(E_1; E_2)(t) = ((\exists t_1)(E_1(t_1) \wedge E_2(t)) \wedge t_1 \leq t)$

   This sequencing operator looks for the occurrence of E_1 followed in time by the occurrence of E_2.

5. $A(E_1, E_2, E_3)(t) = (\exists t_1)(\forall t_2)(E_1(t_1) \wedge E_2(t)) \wedge (t_1 \leq t) \wedge ((t_1 \leq t_2 < t) \rightarrow \neg E_3(t_2))$

   This operator captures all occurrences of E_2 that happen between an occurrence of E_1 and an occurrence of E_3. In the real world, this operator may be used, for example, to capture the occurrence of *every* write operation between the start of a transaction and the end of the transaction.

6. $A^*(E_1, E_2, E_3)(t) = ((\exists t_1)(E_1(t_1) \wedge E_3(t)) \wedge t_1 \leq t)$

   Although this operator looks similar to the sequencing operator described above, the idea here is to *accumulate* all occurrences of E_2 between occurrences of E_1 and E_3 and detects the complex event when E_3 is finally seen (unlike the $A(E_1, E_2, E_3)$ operator described above, which detects the complex event on each individual occurrence of E_2). Note that the above formal expression represents only those points in time when the complex event is detected. It does not explicitly say anything about the accumulation of E_2 events. In the real world, this behavior may be required to collect all the data items touched by a transaction, only when the transaction has ended.

7. $P(E_1, [T], E_2)(t) = (\exists t_1)(\forall t_2)(E_1(t_1) \wedge (t_1 \leq t_2 < t) \rightarrow \neg E_2(t_2) \wedge (t = t_1 + i \times T))$

   This *periodic* event operator detects the complex event periodically, after every interval T, starting from an occurrence of event E_1, and ending the process when an instance of E_2 is seen. In the real world, this may be used, for example, to perform periodic defragmentation inside the database server's memory manager, between the starting of the server, and its shutting down.

8. $P^*(E_1, [T], E_2)(t) = (\exists t_1)(E_1(t_1) \wedge E_2(t)) \wedge (t_1 + T \leq t)$

   This operator detects the complex event, if E_2 is seen at least time T after E_1 is seen.

9. $(\neg E_2)(E_1, E_3)(t) = (\exists t_1)(\forall t_2)(E_1(t_1) \wedge \neg E_2(t) \wedge E_3(t) \wedge ((t_1 \leq t_2 < t) \rightarrow \neg(E_2(t_2) \vee E_3(t_2))))$

   This operator detects the complex event when the first occurrence of E_3 is seen after an occurrence of E_1, provided this interval has not already seen an E_2 event. This behavior may be used by an application that wants to react to read-only transactions - it needs to look for an *end of transaction* that follows its beginning, with no write operation in between.

While COMPOSE uses the equivalence between its algebra and regular expressions to build a finite state automaton for event detection, processing in SNOOP is done in a bottom-up fashion, using a tree of operators. As we will see in Section 3.3.1, the complex event processing operator in TelegraphCQ adopts a similar tree-based strategy.

An interesting aspect of event processing that has been explored in the context of SNOOP, is the use of *parameter contexts*. When a complex event takes place, there may be multiple bindings available for it out of the component simpler events. For example, if we are looking for the complex event $a; b$, the history $a\ b\ b$, may detect the complex event once or twice, depending on what semantics are desired by the application. Details of the different contexts (*unrestricted*, *recent*, *chronicle*, *continuous*, and *cumulative*) explored in this work can be found in [6]. The following discussion on a formal metamodel for event algebras touches on this aspect again.

2.1.4 A Metamodel for Event Algebras

The large number of proposals for complex event processing algebras in active databases sometimes makes the area hard to understand. This issue has been raised by Zimmer and Unland [32], who present a *framework* that is a metamodel for any event processing algebra. The idea is to decompose a complex event processing algebra into three independent dimensions:

1. **Event type pattern:** This aspect covers the *structure* of a complex event, i.e., which component event types' instances are required to detect the complex event. There are multiple aspects to this dimension - *type and order*, *repetition*, *continuity and concurrency*, and *context condition.*

   The *type and order* aspect gives the identity of component event types, the order in which they are required to occur for the complex event to be detected. The following operators are available in the metamodel (all operators have arity n):

   - *sequence operator (;)* for expressing ordering.
   - *simultaneous operator (==)* for expressing the simultaneous occurrence of events.
   - *conjunction operator ($\wedge$)* captures the order-insensitive occurrence of (all of) multiple component events.
   - *disjunction operator ($\vee$)* requires at least one of the specified events to occur.
   - *negation operator ($\neg$)* specifies the events that are not allowed to occur in a given interval. Negation makes sense only in the context of an interval, whose end points are formed by the first and last parameters of the

operator. The interior events are the ones whose non-occurrence is being sought.

The *repetition* aspect captures the number of component event types that have to occur, to detect the complex event. It can be specified by a delimiter that is written against the event type. It can be an exact number or an (open or closed) interval. An example usage is: $((2)E_1, (3)E_3)$, for two occurrences of E_1 followed by three occurrences of E_3.

The *continuity* aspect indicates whether *irrelevant* events are allowed to take place while the sequence of relevant events is being detected towards a complex event. The *concurrency* aspect determines if the component events are allowed to overlap in time.

The *context condition* aspect specifies restrictions on the context in which component events take place. The term *context* refers to the attributes of the component events; for example, the transaction-id of the transaction that generated this event, user-id of the user behind the event, and the data item whose modification caused the event. The metamodel presented by the authors has three context types that can be addressed individually. The *environmental context* allows the specification of the requirement that appropriate event instances must be triggered by the same transaction (*ta*), process (*proc*), or user (*user*). An example usage is: *(same ta)*$(E_1; E_2)$, which is detected when the specified sequence of events takes place, triggered by the same transaction. Similarly, the *data context* concerns the actual data items which are involved in the events specified. For example, *(same data-id)*$(E_1; E_2)$ is detected when the specified sequence of events (which are assumed to represent data operations) happen over the same data item. Finally, the *operation context* tests for the events coming as a result of the same operation.

2. **Event instance selection:** When a complex event is detected, a decision has to be made regarding which instances of the component events will be bound to this event. This aspect is captured by event instance selection.

   The metamodel claims that this dimension is independent of *event type pattern*, as the latter specifies *when* a complex event has to be detected, while event instance selection specifies which instances of the component events are kept for the execution of, say, the underlying ECA rule. To appreciate the need for this dimension, let's look at the following example. Suppose the event type pattern specifies that we are looking for $(E_1; E_2)$. If we see an event history[1] of the form $[e_1^1, e_1^2, e_2^1]$, we clearly need to detect the complex event when we see the

[1] e_j^i denotes the ith occurrence of the jth event type.

simple event e_2. However, when trying to bind E_1 and E_2 with component event instances, we have the option of binding either e_1^1 or e_1^2 to E_1. Note that this choice is arbitrary, and in general, depends on the application semantics. For example, if E_1 denotes the event "person enters the home, through the door" and E_2 captures "alarm rings", a security application may want to correlate the alarm with the person who *just* entered the home. Hence, this application will need only the second occurrence of E_1, i.e., e_1^2, to be returned for further rule processing. On the other hand, if E_1 is an RFID tag reading for a particular object, and E_2 is the event that refers to "some object being taken out of the store", then multiple occurrences of event E_1 may just be due to the fact that RFID antennas continuously sense the tags close to them, and all the readings but the first one are redundant. In this case, an application that wants to find out which objects go out of a warehouse, will want only the first instance of E_1, i.e., e_1^1, to be considered for further processing.

The metamodel provides keywords like *first*, *last* and *cumulative* to select which instances of component events will be selected for processing. These keywords respectively select the first, last and all permissible instances of the component event, from the underlying history. For example, given the complex event expression $;(first : E_1, last : (2)E_2, E_3)$, the event history $[e_1^1 e_2^1 e_1^2 e_2^2 e_2^3 e_1^3 e_3^1]$, would result in the following selected instances: $\{e_1^1, e_2^2, e_2^3, e_3^1\}$. On the other hand, the expression $;(last : E_1, first : (2)E_2, E_3)$ will cause bindings to $\{e_1^2, e_2^2, e_2^3, e_3^1\}$.

3. **Event instance consumption:** After a complex event has been detected using specific component event instances, some (all) of the component events may have to be deleted to ensure they do not trigger any additional instances of the complex event. This aspect is covered by event instance consumption.

   For example, when the event expression $;(last : E_1, first : E_2)$ operates on the history $[e_1^1, e_1^2, e_2^1, e_2^2]$, an instance of the complex event is detected when e_2^1 is seen. Given the specification of event instance selection, the complex event will be bound with $\{e_1^2, e_2^1\}$. If this detection of the complex event does not cause the deletion of these component events, then the occurrence of the simple event instance e_2^2 will trigger the detection of another instance of the complex event, bound to $\{e_1^2, e_2^2\}$. Depending on the application semantics, this *reuse* of the component events may or may not be desired. For example, in a library scenario (which we will revisit in Section 3.2), if E_1 represents a book's barcode being read at the library counter, and E_2 represents a patron's barcode being read at the counter, then $E_2; E_1$ may be used to detect the *checkout* complex event, which is interpreted as the patron checking out that specific book. In general, a patron may check out several books at a time. Hence, when a history of the form $[e_2^1, e_1^1, e_1^2]$ is seen, this application does not want to delete e_2^1 the first time a checkout is detected (with e_1^1). If this happens, the book corresponding to e_1^2 will not be checked out for the same patron. In other words, this application does not permit the expiry of component events when complex events are detected.

The metamodel provides different consumption modes - the *shared* mode does not delete any instance of the component events, while the *exclusive* mode deletes all instances of the component events (that have been used in a binding once).

Based on this metamodel, the authors of [32] describe how several composite event processing algebras, including COMPOSE and SNOOP, fit into this framework.

2.2 CEP: Causality and Hierarchy

The Complex Event Processing (CEP) technology [1] started as a research project at Stanford, between 1993-2000. The roots can be found in the RAPIDE event-based simulation language and simulation analysis toolset. RAPIDE allowed concepts such as causality between events, and event pattern mappings, to be used to model multi-layered system architectures that might involve both hardware and software layers. CEP is being applied to multiple areas: Business activity monitoring, business process management, network and systems security, application servers and middleware.

The concepts that came out of RAPIDE can be applied to any distributed message-based system, from low-level network management to high-level enterprise intelligence gathering. These also form the basic concepts of CEP:

1. Hierarchies of events
2. Causal event modeling
3. Event patterns and pattern matching
4. Event pattern maps
5. Event pattern constraints

We elaborate on the first two aspects next. Details of the RAPIDE language can be found in [22].

2.2.1 Hierarchy of Events

A general introduction to CEP has been presented in [21]. It argues for *multilevel viewing* of events in a complex distributed system. For example, listening to the message bus in a multi-component system may give us only low-level messages that the components are passing between each other. However, for analyzing problems in such a system the administrator may want to aggregate sets of low-level events into the corresponding higher-level events (which signify what the components were trying to do when passing the low-level messages).

CEP's approach is to separate the system's activities and operations into layers. This is called an *abstraction hierarchy*. For example, in a fabrication line control system, a two-level hierarchy may look like:

1. **Layer 1:** Middleware Communication. Activities here include sending/receiving messages.

2. **Layer 2:** FabLine Workflow. Activities here include movement of lots, machine status change, yield measurement etc.

In other words, layer 2 provides semantically higher information about the system's activities than layer 1. An application or analysis tool can now attach to any level it wants, based on its requirements. The hierarchy in CEP is flexible and dynamic.

The CEP mechanism for generating a hierarchy is to use RAPIDE to filter events at each level and then map event patterns at one level to complex events at the next higher level.

2.2.2 Causality in Event Processing

Apart from aggregating sets of events, the relationships between events are also an important aspect in CEP. There may be different kinds of relationships between events in a system. For example, *event A happened after (before) event B*, or *event A caused event B to happen*. Causality is fundamental because it helps analyze a complex system. For example, an administrator would like to know what went wrong at a lower layer of the abstraction hierarchy that caused improper behavior in a higher layer. This is aided if the system can reason about what events in the lower layer *caused* the abnormal events in the higher layer.

3 Complex Event Processing over Streaming Data

This section presents an overview and details of our system for complex event processing over streaming data. Several data stream processing systems [7, 2, 4] have emerged over the last few years, and have been accompanied by a lot of research into issues like query languages, (adaptive and shared) query processing, load management, and archived stream management. The reason for these research initiatives is the increasingly *streaming* nature of data in today's world, in the form of network traffic, receptor (sensor and RFID) streams, and XML news feeds. This trend has triggered a paradigm shift in data processing, which was traditionally about queries streaming (and getting answered) over disk-based data, to data streaming *through* continuous standing queries (CQs). For example, the query "Is my host under a TCP hijacking attack?", which can be processed by detecting specific signatures in the TCP traffic, needs to run continuously, with the TCP-layer traffic data flowing through it.

This model of streaming data processing, unlike traditional querying, resembles active database-like event processing in two ways:

1. Streams are analogous to (time-ordered) event histories.

2. Continuous queries are analogous to event expressions, that continuously monitor the event history for matching patterns.

These similarities raise the following question: *Shouldn't the query language and operators for stream processing be inspired by event processing in addition to traditional SQL-like querying?*

Most of the current data stream processors use SQL, extended with *windows* over streams, for expressing continuous queries. A window defines some portion of an infinite stream, based on temporal proximity ("a 5 second or 6 tuples long window"), over which some computation must take place. For example, we may want to find the average price of Google stock over the last 1 hour of quotes. In this example, the size of the window is *1 hour* and the computation performed over this window is the aggregation for average. Given the continuous nature of such queries, a window has to continuously slide as it sees more data on the stream. This *sliding window* paradigm for querying streams has dominated much of data stream research. A concrete proposal for a streaming query language is CQL [5], which gives stream processing functionality by providing *stream-to-relation* and *relation-to-stream* operators, on top of the normal *relation-to-relation* operators. This effort represents an attempt to solve the streaming query language problem by extending the well-known SQL language.

We describe our system for Complex Event Processing over streams in three parts. First, we look at query language extensions that allow the specification of active database-like complex events on streams. This includes a discussion on *semantic windows*, which go beyond time-based or tuple-based windows used in SQL-extensions for streaming systems. Second, we describe how queries phrased using these language extensions were used to drive the core functionality of an event-driven library scenario, demonstrated recently at SIGMOD 2005 [27]. Finally, we describe some issues in the implementation of these ideas in the context of the TelegraphCQ [7] data stream processor built at Berkeley.

3.1 Complex Events over Streams: Semantic Windows

Computation and operators over streams differ from those over traditional relations because of the fact that streams are continuous and infinite. This limits the operators that can be applied to streams to only the set of *non-blocking* operators, which do not wait for the end of input data to return a result. A recent result [18] shows the equivalence of the computation that can be performed using only non-blocking operators to the computation that is imposed by the set of *monotonic queries*, which informally, is the set of queries that *produce strictly more output as they see more input data*. Not all operations performed on traditional relations are non-blocking. For example, the sorting operation falls into this category, and cannot produce any output until it has seen all the input. Windows help to convert traditionally blocking operators into non-blocking operators by specifying a portion of the infinite stream over which the computation should be performed[2][3].

In current systems, windows are defined *structurally*, based on temporal proximity of the data inside the window. For example, In CQL, the query:

```
SELECT AVG(price)
FROM Quotes [5 min]
WHERE name = ‘GOOG’
```

is interpreted as a time-varying relation. At any time, the content of the relation (a single-column, single row), is defined by taking the average stock price over the ‘Google’ tuples in the Quotes stream which have timestamps in the range $[T - 5min, T]$, where T is the current time.

In the latest version of TelegraphCQ, the query:

```
SELECT AVG(price)
FROM Quotes [RANGE BY ‘5 min’, SLIDE BY ‘1 min’]
WHERE name = ‘GOOG’
```

[2]An alternative is to use punctuations, which capture application semantics [30].

[3]Note that the join operator is not inherently blocking. It can have a non-blocking implementation, like symmetric hash join.

is interpreted as a stream of single-column tuples. One such tuple is reported on the stream every 1 minute. If the tuple is computed at time T, it consists of the average of 'Google' stock tuples with timestamps in the range $[T - 5min, T]$.

Both these examples show how windows are defined structurally. Expressing complex events over streams requires more "windowing flexibility" than what is offered by structural windows. For example, suppose an RFID-enabled library reports books and patrons as they move around the library (shelves, counter, door). To detect booklifting, which is defined as a book going out of the door without being preceded by a checkout, the most natural way to express this query is using a *window which starts backwards in time whenever a book is seen going out of the door, and looks for the presence or absence of the corresponding checkout.* As another example, consider a query that wants to find the average temperature in the home between the time a person comes home and when he goes out. This query is looking at a window over a temperature sensor stream that starts and ends based on "person" events. The concept is illustrated in Figure 1. Figure (a) shows a structurally-defined window of size t. Figures (b) and (c) show "semantic" windows which are triggered by event e and span an interval t over the stream in the future or the past w.r.t. e (in contrast, the structurally-defined window in Figure (a) always has its recent edge on current time). Figure (d) shows a window which extends in time based on two events e_1 and e_2. As we will see shortly, our event operators are based on the semantic windows depicted by Figures (b) and (c). Providing an operator that expresses the semantic window in Figure (d) is a topic of future work. Note that the way a window is defined (structurally or based on one or two events) is orthogonal to *what* computation is done inside the window (average, max, etc.).

We call such windows *semantic windows*, as they are not defined based on a number of tuples or a time interval but rather semantically, based on the occurrence of some (simple or complex) events. Concurrent to our work, a group at University of Texas at Arlington has worked on semantic windows [16]. Their motivation for going beyond structural windows is the same as ours. However, their mechanism for defining the start and end of a window is different from our event-based approach, and involves writing an SQL query that tests some condition on the *next* tuple of the stream.

We shall proceed with the following fundamental binary event operations. Our choice is based on simplicity and usefulness. While generic n-ary operators are strictly more powerful than our binary operators, we wanted to start with a more basic set of operators that could still capture many real-world complex events, such as those in our target library scenario described in Section 3.2.2. In the following, E_1 and E_2 are two event streams, while e_1 and e_2 represent individual event tuples on these streams. These events occur when the corresponding tuple is seen on their stream.

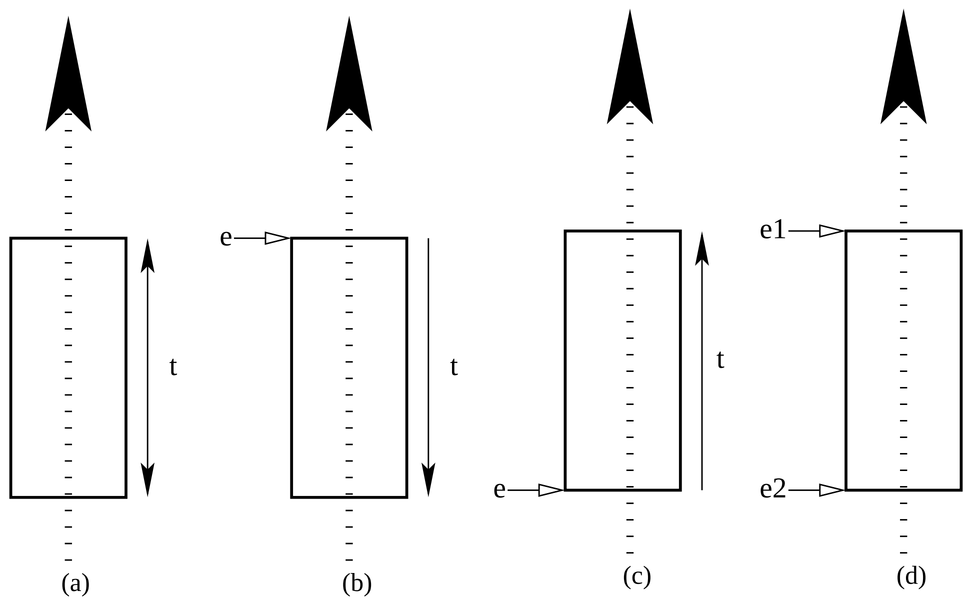

Figure 1: Windowing Options for Streaming Data.

- FOLLOWS(E_1, E_2, t). This detects if an event instance e_2 is seen on stream E_2 in the time interval t following an event e_1 being seen on stream E_1. This defines a semantic window of length t over stream E_2, triggered by events on stream E_1. The timestamp of the resulting event, is that of e_2.

- NOTFOLLOWS(E_1, E_2, t). This detects the absence of any event instance on stream E_2 within a window of length t triggered by events from stream E_1 (following the corresponding E_1 events). The timestamp of the resulting event is t plus that of e_1.

- NOTPRECEDES(E_1, E_2, t). This detects the absence of any event instance on stream E_2 within a (backward) window of length t triggered by events from stream E_1. The timestamp of the resulting event is that of e_1.

- ANYONE(E_1, E_2). This defines a "union" event, which triggers whenever an event is seen on either stream E_1 or E_2. The timestamp of the resulting event is that of the event which triggers it.

Note that the output of these operators is also a stream, which consists of tuples that are generated whenever the operator detects its corresponding event. This approach results in closure of these operators over streams, and hence they can be arbitrarily nested within each other. For example, NOTFOLLOWS(NOTPRECEDES(E_1,

E_2, t_1), E_3, t_2) can be interpreted by recursively applying the semantics of the operators defined above.

These operators constitute a non-redundant operator set for detecting complex event correlations in our system. Additional operators can be defined using them. For example, we found it useful to have the operator PRECEDES(E_1, E_2, t), which is defined as FOLLOWS(E_2, E_1, t). Also, the operator BOTH(E_1, E_2, t) looks for events on the two streams within time t of each other, and is equivalent to the sliding window join operator in current streaming systems. In terms of our algebra, it can be written as ANYONE(FOLLOWS(E_1, E_2, t), FOLLOWS(E_2, E_1, t)). Note that the NOTFOLLOWS and NOTPRECEDES operators cannot be expressed in terms of each other.

3.2 CEP in a Real-World Scenario

In this section, we describe how we extended the TelegraphCQ data stream processor to execute complex event continuous queries, along with traditional SQL-like continuous queries. This was achieved in two ways.

1. Extending the TelegraphCQ query language with an EVENT clause that expresses event correlations sought between the streams specified in the FROM clause. We also needed a CQL-like PARTITION BY clause, for *parameterization* - correlating the attributes inside the events that constitute the complex event being detected. The details are presented in this section.

2. Modifications to the query execution engine of TelegraphCQ. This consisted of adding a new operator for event aggregation, that implements the Fjord [23] interface. The internals of this operator involve a tree-based bottom-up event detection mechanism, similar to SNOOP [6], a base event cache manager, and a timeout service. We describe this in Section 3.3.

3.2.1 Query Language Extensions

After our query language extensions, any previously valid query is still supported by TelegraphCQ. Such a query is of the form:

```
SELECT [attributes | (sliding-window) aggregates]+
FROM [stream-name [RANGE BY t1 SLIDE BY t2] | relation-name]+
WHERE <selection and join predicates>
GROUP BY <grouping attributes>
HAVING <group selection predicates>
```

We will not go further into the details of the query semantics for this class of queries. After our extensions, an alternate form of the TelegraphCQ query looks like:

```
SELECT <attribute list>
FROM [stream-name alias]+
PARTITION BY <partitioning attributes>
EVENT <complex event expression over aliases>
WHERE <selection predicates>
```

A quick example of this kind of a query is the data cleaning query:

```
SELECT R1.*
FROM RFIDStream R1 R2
PARTITION BY tag_id
EVENT NOTPRECEDES(R1, R2, '1 sec')
```

This query is a filter. It produces a stream of RFID tag readings, that have not been preceded by another reading for the same tag, within the last one second. Note that the PARTITION BY clause makes sure that two readings corresponding to different tags do not "interact" with each other. The FROM clause has two tuple variables, R_1 and R_2, that reference two *distinct* tuples from the same stream RFIDStream. This is similar in spirit to sequence query languages, and we visit that literature in Section 5. The NOTPRECEDES event operator looks for tuples that bind to tuple variable R_1, such that *there exists no other tuple that can be bound to* R_2 *and was seen in the 1 second period before* R_1 *was seen.* We extensively used such queries for cleaning RFID data in our live demonstration, given that RFID readers continuously produce readings for tags in their vicinity, and most applications want to be notified about a tag only the first time in appears in a certain location.

3.2.2 A Live Demonstration

We now discuss how our system was used as the core processing engine in a real-world scenario. This scenario extensively used sensing technology (RFID, sensors) as the source of event streams. With the rapid adoption of RFID and other sensing technology, a new class of applications is emerging. These applications use receptor technology at their lowest-level and then build complex functionality on top. An example of such an application is an RFID-enabled supply chain, which monitors items as they move (supplier, warehouse, retail store) using RFID technology and also monitors warehouses with other sensing technology (temperature sensors). The HiFi project at Berkeley [12] is exploring the infrastructural support for such systems, and our work is in the context of HiFi.

The extended TelegraphCQ system was used as the core engine in a live demonstration at SIGMOD 2005. We mocked up a real-world library with the following components:

1. Books on shelves. Each book was RFID-tagged, and also equipped with the usual ISBN barcodes. The shelf was monitored using RFID antennas, for tracking books as they move in and out of the shelf.

2. Patron tags. Each patron of the library was given an RFID-tagged ID card. They were also given the usual barcode-equipped ID cards.

3. The counter. The usual librarian counter was setup. This was equipped with a barcode reader.

4. RFID-enabled door. The library door was equipped with RFID antennas. This was used for tracking the movement of books and patrons in and out of the library.

5. Motion sensors on doors. These simple sensors indicate the presence or absence of motion in their (orientation-specific) vicinity. They were used in conjunction with RFID antennas at the door, for directed motion sensing.

The high-level view of the demonstration setup is shown in Figure 2.

Complementary to our work, the work described in [15] makes up the *Virtual Device (VICE)*. The goal of the VICE is to abstract away all the details of the underlying sensing technology from the system layers (and applications) above it, like the Complex Event Processing engine. Although the VICE forms an important part of the demonstration, it is not the topic of this report.

What is relevant to this discussion is that in this arrangement the VICE exports an interface to the CEP engine that consists of the following streams:

```
Person(person_id, loc_id, timestamp)
Book(book_id, loc_id, timestamp)
```

The schema of these streams is self-explanatory. It captures the identity of entities and the locations they are observed at inside the library (shelf, counter, door) at a given time. All the CEP queries were written on top of this abstraction. Some more streams were defined in the system:

```
Checkout(person_id, book_id, loc_id, timestamp)
```

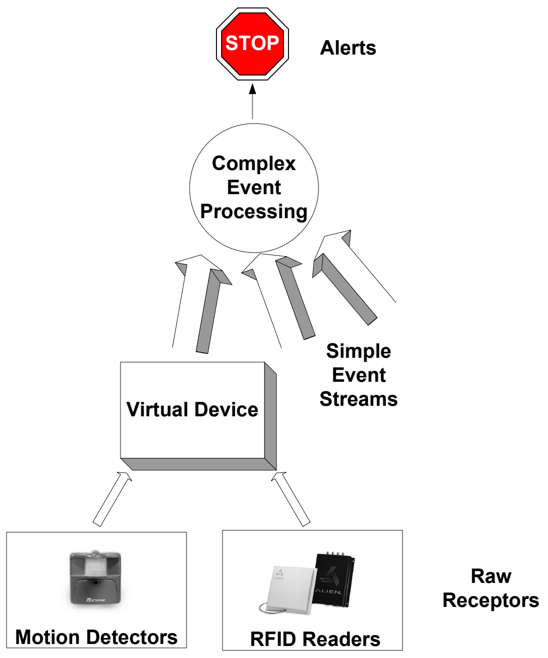

Figure 2: Complex Event Processing over Receptor Streams.

This captures the checkout of a specific book by a specific person[4]

```
BookliftingAlert(book_id, loc_id, timestamp)
```

This captures an alert stream, which indicates a book being taken out of the library, from the location given by the *loc_id*, without being previously checked out.

```
ConfusionAlert(loc_id, person1_id, person2_id, book_id, timestamp)
```

This captures another alert stream, which indicates the failure of the CEP engine to unambiguously checkout a book for a particular person. This happens when, for example, two people walk out of the door within a small period of time, with one of them carrying a book. In this case, it becomes very hard for an automated system to disambiguate the checkout (uniquely identify the patron checking out the book). Hence, we detect this as another complex event, and rely on a human to attend to it.

[4]Several details have been removed from this discussion. For example, in the actual demonstration, catalog lookups were made to transform IDs to meaningful names of books and patrons. Several cleaning queries were also used, similar to the one in Section 3.2.1.

```
OverlimitAlert(person_id, loc_id, timestamp)
```

Our final alert stream detects an *otherwise valid checkout* that is invalid because of some pre-defined business rule. In our library, no patron should be allowed to checkout more books than her limit.

Our demonstration team designed an extensive visualization scheme, and this was used on top of these streams to give an insightful view to our live audience, into the working of the library. We will not go into the details of our visualization.

We will now discuss some of the complex event queries that were used to feed data into the above streams. The following query detects booklifting alerts:

```
SELECT B.*
FROM Book B, Checkout C
PARTITION BY book_id
BOTH(NOTPRECEDES(B, C, '1 min'), NOTFOLLOWS(B, C, '2 sec'), '3 sec')
```

Every time a book is seen going out of the door, this query looks at two semantic windows on the checkout stream - one into the past, and the second into the future. The first one checks for the absence of checkouts for this book at the counter, and the second one looks for the absence of door checkouts in the immediate future (note that, in our demo, a turbo-checkout scheme let the RFID-tagged user checkout a book by just walking out of the library with the book). The PARTITION BY clause makes sure that each book is tested for booklifting independently.

The confusion alert was detected by:

```
SELECT P1.loc_id, P1.person_id, P2.person_id, ....
FROM Person P1 P2, Book B
PARTITION BY loc_id
EVENT ANYONE(
   FOLLOWS(B, BOTH(P1, P2, '3 sec'), '3 sec'),
      FOLLOWS(BOTH(P1, P2, '3 sec'), B, '3 sec')
         )
```

This detects the presence of two persons and a book in very close proximity in time. The PARTITION BY clause makes sure that events at different locations are not mixed up, to raise false alerts.

3.3 Implementation Issues

In this section, we describe the systems details of our prototype. TelegraphCQ [8, 17] processes continuous queries (CQs) over streams in a *shared* fashion. Some key systems features are:

- There is a single *backend process* that runs a query plan that is the result of merging all the currently executing queries. The merging algorithms take care of exploiting commonality between queries, so that redundant work is minimized while processing.
- In addition, each client is attended by its own *frontend process* that takes care of parsing, planning and optimization of queries specific to that client. Note that, in a streaming environment, the frontend cannot perform much optimization, as statistics about data is not available. Much of this work is pushed into the backend, in the form of adaptive query processing [24].
- In addition to these key processes, a *Wrapper Clearing House* process takes care of reading raw bytes from network connections that come from remote data sources. It then converts this raw data into structured tuples, that are sent to the backend for processing.

The different processes communicate with each other through shared memory queues. Other details of the TelegraphCQ architecture are omitted from here. They can be found in [8].

Our extensions to TelegraphCQ go into the frontend and the backend. The frontend changes take care of parsing and planning complex event queries. An arbitrarily complex event query is converted during the planning phase to a tree similar to the one shown in Figure 3. The tree in this example corresponds to the booklifting query discussed earlier. There is one node in the tree for every event operator in the query. The structure of the tree is derived from the nested structure of the event operators in the query. The PARTITION BY clause is compiled into a special data structure. Partitioning is detailed in Section 3.3.3.

The extensions to the TelegraphCQ backend are in the form of a new Fjord-based [23] operator for event aggregation. The Fjord model goes beyond the traditional iterator-based interfaces for database operators. It connects operators to each other using *smart queues*, which allow both *push* and *pull* behavior within the query plan. Next, we go into the details of some key features of this operator.

3.3.1 Bottom-up Event Propagation

The run-time event aggregation operator consists of the tree generated at compile time, and several other data structures (Figure 4). Complex events are detected

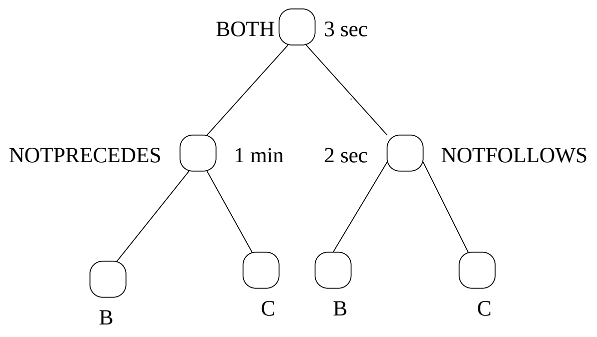

Figure 3: Frontend-compiled tree data structure for an event query.

using the tree in a bottom-up fashion, as was the case with SNOOP [6]. Every simple (base) event tuple is "published" to the corresponding leaves of the tree. For example, a tuple on the book event stream will be published to the leaf nodes marked with B in the tree of Figure 3. Nodes in the event tree have SteM-like [26] data structures called IESteM (Intermediate Event SteM) for storing the intermediate events generated by the subtree rooted at them. Not all nodes have to *materialize* their output events in an IESteM. Whether or not a node materializes its output events depends on its parent node type. This will be explained shortly.

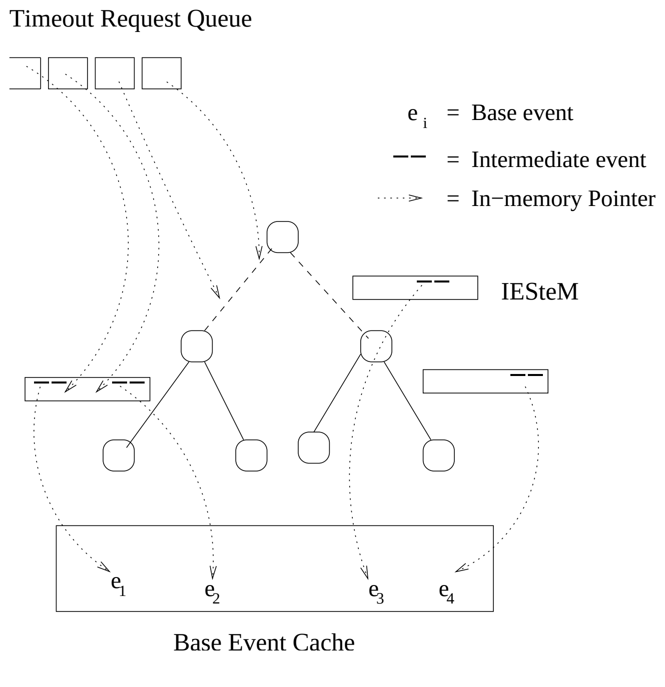

Figure 4: Run-time details of the event aggregation operator.

The operation *GenerateComplexEvent()*, when called on an event node inserts its output events into its output IESteM. If an output event has been produced, the node may call *GenerateComplexEvent()* on its parent recursively. This is how event detection propagates bottom-up along the tree structure.

Different operators implement the *GenerateComplexEvent()* interface differently:

- FOLLOWS(E_1, E_2, t): This operator is scheduled when an event is generated by its right child. It looks for E_1 events in its left child's IESteM that match the E_2 event triggering it. Note that this operator does not need to be *scheduled* when its left child generates an output event but rather only when its right child generates an output event. An event in the left child's IESteM can be expired after time t. The right child does not need to materialize its output in an IESteM.

- NOTFOLLOWS(E_1, E_2, t): This operator is scheduled in three cases. First, if its left child generates an output event e_1. In this case, it inserts a timeout request for time $t_{e_1} + t$, where t_{e_1} represents the timestamp of the event e_1. Second, when the right child generates an output event e_2. In this case, the operator expires the matching e_1's from the left IESteM. This expiry is optional, and can be avoided if a *lazy* strategy is used. Third, by the timeout mechanism. In this case, it looks for the presence of a matching e_2 event in the right child's IESteM. If no such event is found, it puts the concerned e_1 into its output IESteM. In either case, e_1 is expired from the left IESteM.

- NOTPRECEDES(E_1, E_2, t): This operator is scheduled when its left child produces an output event. It looks for the absence of a matching e_2 in the right child's IESteM. An event in the right child's IESteM can be expired after time t.

- ANYONE(E_1, E_2): This operator is scheduled whenever any of its children generates an output event. It just inserts the triggering child event into its output IESteM, if any.

Figure 4 shows the details of a run-time event aggregation operator. It consists of the following three components:

1. The event tree. IESteMs are allocated wherever they are required (as discussed above).

2. A base event cache. Base events refer to the tuples that enter the event aggregation operator. They correspond to the leaf nodes of the event tree. We describe this component in more detail in Section 3.3.2. It consists of a copy of

every base event that enters this operator. Various operators in the event tree maintain pointers to these base events in their IESteMs. This makes sure that no base event is materialized more than once, as should be the case. A base event is maintained as long as any IESteM has an intermediate event pointing to it.

3. A timeout queue. This queue is used by instances of the NOTFOLLOWS operator in the tree. Each request in the queue has a pointer to the intermediate event which caused this request, and a time when the request will fire. As mentioned above, timeouts can be requested only by intermediate events that go into a NOTFOLLOWS operator's left child IESteM. In our current implementation, the timeout queue is served by the operator itself. However, a more general design principle would be to make a centralized scheduler, like the Eddy [24], serve the queue.

Operator	**IESteM for E_1**	**IESteM for E_2**	**Scheduled On**
FOLLOWS(E_1, E_2, t)	window of t	None	E_2
NOTPRECEDES(E_1, E_2, t)	None	window of t	E_1
NOTFOLLOWS(E_1, E_2, t)	window of t, event-driven expiry	None	E_1, E_2, Timeout
ANYONE(E_1, E_2)	None	None	E_1, E_2

Figure 5: Memory Management for event operators. The events that cause the given operator to be scheduled are shown in the rightmost column.

A summary of how each operator behaves, in terms of scheduling and IESteM state for its children, is given in Figure 5.

3.3.2 Base Event Cache Management

Base events are combined as they move up the event tree (for example, an e_1 and an e_2 event may get combined by a FOLLOWS(E_1, E_2, t) operator, before the result is inserted into an IESteM). In general, any base event may be propagated up the tree along different paths, and inserted in multiple IESteMs (for example, in our booklifting query (Figure 3), a book event gets picked up by two leaf nodes, and goes up to the root along two different paths).

As a space optimization, we do not materialize each copy of a base event inside the operator independently. Rather, a single copy of the event is maintained in a *Base Event Cache* and intermediate events consist of pointers to events in this cache. We note that the same trick is played in traditional database systems, where as (composite) tuples move across operators, they consist of pointers to the base tuples in the buffer pool.

3.3.3 Enforcing Partitioning: Necessary Pushdowns

If a PARTITION BY clause is present in the query, then only a subset of all complex events produced by the event aggregation operator are actually needed. For example, a one-level FOLLOWS(E_1, E_2, t) query should output a complex event of the form e_1e_2 only if the simple events e_1 and e_2 have the same value for all the attributes in the PARTITION BY clause.

One solution is to enforce partitioning at the *top of the tree*. In this case, a complex event of the form $e_1e_2...e_n$ is deleted if *any* two component events e_i and e_j do not have the same value for *every* partitioning attribute. However, this approach gives incorrect results, due to the presence of *negative* event operators, like NOTPRECEDES and NOTFOLLOWS, in the event tree. To see this, let's look at the event tree node corresponding to a NOTPRECEDES(E_1, E_2, t) operator. Suppose this operator is scheduled due to the insertion of an event e_1 into its left child's output IESteM. Suppose the right child's IESteM currently has an event e_2 whose timestamp is between $t_{e_1} - t$ and t_{e_1}. Then this e_2 will cause e_1 to get deleted immediately, and not be inserted into our operator's output IESteM. However, this should happen only if e_2 and e_1 belong to the same partition (have the same value for each partitioning attribute). Thus, the *top of the tree* approach may lose some complex events that should have been generated. This problem occurs because this operator has *negative* semantics.

In our implementation, we make sure that partitioning is pushed down the event tree, so that the above problem is avoided.

3.4 Experiments

In this section, we compare our system with the vanilla TelegraphCQ relational stream processing engine. TelegraphCQ offers only traditional relational operators over streams, like selections, projections, and (windowed) joins. However, our operators are targeted towards complex event queries, that involve sequencing and negation, in addition to traditional operations.

We compare our system (TCQ*) to TelegraphCQ (TCQ) by designing experiments that exploit the expressive power of the FOLLOWS operator. Note that NOTFOLLOWS and NOTPRECEDES cannot be expressed in TelegraphCQ, as no available relational operator can capture their negative semantics[5]. The FOLLOWS operator is captured in the TelegraphCQ language using the join operation, followed by a selection that imposes the sequencing requirement.

[5]Outer joins are currently not available in TelegraphCQ.

Parameter	Description	Values
numEventTypes	Number of distinct event types	20
numAttrs	Number of attributes in every event	5
ds	Domain size for event attributes	10, 50, 100, 500, 1000
sl	Length of sequential pattern in the query	2, 3
ws	Window size for query	100-10,000

Figure 6: Data and Query Workload Parameters.

3.4.1 Data and Query Workload

Figure 6 illustrates the parameters involved in our workload generation [6]. One base event stream carries *numEventTypes* different event types, which differ in their *type* field. In addition to this field and the usual timestamp, every event has *numAttrs* fields, that are associated with different domain sizes. In our experiments, we set *numAttrs* to 5. An event tuple on the base stream is chosen uniformly from each of the *numEventType* event types. Once the event type has been chosen, each field in the event is chosen uniformly from an integer domain whose size is equal to the chosen domain size.

The query workload is generated as follows. Every query consists of a sequence of *sl* event types. For example, when $sl = 2$, the query is detecting occurrences of E_1 events followed by E_2 events. Further, a (semantic) time window of *ws* events is imposed, such that only the sequences that occur within such a time window are extracted. The five attributes available in events are used, in turn, as the partitioning attributes in the query. Before we discuss our specific experiments and results, we describe how the TelegraphCQ system was configured for measurements.

3.4.2 Experimental Setup

As described earlier, the TelegraphCQ system is a multi-process server. A backend process carries out all the continuous query processing, while other processes are responsible for client and data source connections. In our experiments, we made sure that all measurements capture only the query processing costs, inside the backend. To achieve this, the TCQ code base was engineered to avoid unnecessary inter-process communication, through shared-memory operations. The backend was interfaced with a stub, that included:

- The random event generator, for feeding the base stream into the execution engine.

[6]Yanlei Diao and Eugene Wu were extremely helpful in setting up the experiments.

- Timing code, to measure the time taken for processing batches of tuples that go into the executor.

For every experiment, multiple batches of tuples were streamed into the executor. The experiment terminated when the time taken to process a batch had shown convergence across several batches. The resulting time delay, along with the batch size, was used to calculate the throughput of the system for the running query.

All experiments were run on a 2.8 GHz Pentium 4 machine, with 1 GB of RAM and running Fedora Core 2 (based on Linux kernel 2.6.5). Processes were prevented from hitting the swap device at any time.

3.4.3 Results and Analysis

Our first experiment shows the importance of sequencing-aware operators for Complex Event Processing. Consider the following query, which detects occurrences of E_1 followed by occurrences of E_2, inside an ws-tuple window:

```
WITH
E1 as (SELECT * FROM base WHERE type = 1)
E2 as (SELECT * FROM base WHERE type = 2)
      (SELECT *
      FROM E1 a, E2 b
      PARTITION BY a3
      EVENT FOLLOWS(a, b, ‘ws’)
      )
```

Note that attribute $a3$ is chosen as the partitioning attribute, and its domain size is 100. The same query can be written in vanilla TCQ as:

```
WITH
E1 AS (SELECT * FROM base WHERE type = 1)
E2 AS (SELECT * FROM base WHERE type = 2)
      (SELECT *
      FROM E1[RANGE BY ‘ws’], E2[RANGE BY ‘ws’]
      WHERE E1.a3 = E2.a3 AND E1.tcqtime < E2.tcqtime
      )
```

This query performs sequencing by first joining the two streams E_1 and E_2 in a sequence-agnostic fashion, and then applying a selection on top of the join result, to enforce the sequencing requirement. On the other hand, the Complex Event Query given earlier is sequencing-aware, and does not have to perform the unnecessary join. Specifically, the TCQ query will have to perform work (like probing a hashtable for matching tuples) every time it sees an E_1 tuple or an E_2 tuple. However, given that the FOLLOWS operator is scheduled only when an E_2 event is seen (Figure 5), it has to do only minimal work when an E_1 event occurs (such as inserting it into an IESteM). This reasoning is verified by our experiment (Figure 7), which varies the value of ws in the above queries, keeping all other parameters fixed. Note that as ws increases, the *wasted* work performed by vanilla TCQ for E_1 tuples also increases. This is manifested as an increasing difference between the throughput of TCQ and TCQ*, in Figure 7.

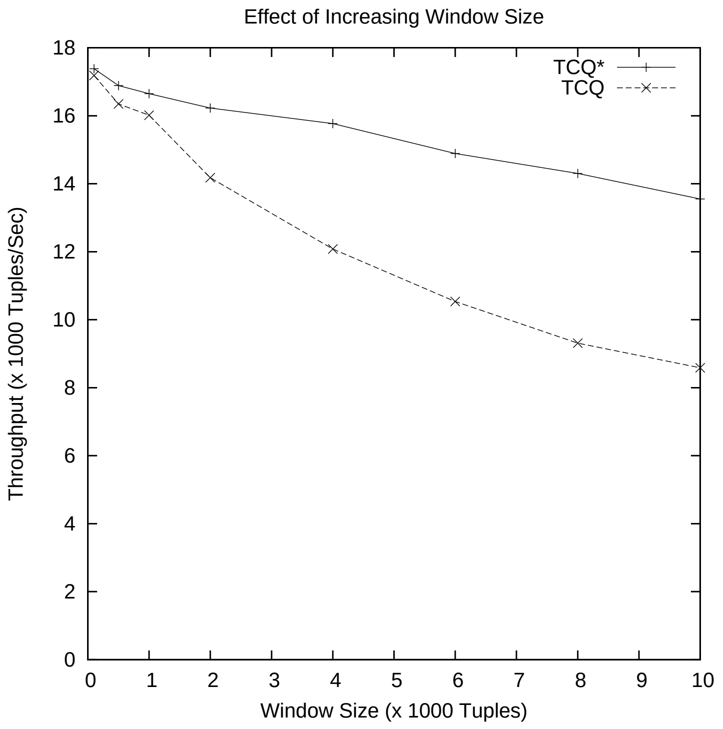

Figure 7: Varying Window Size

The amount of wasted work done by the vanilla approach not only depends on the size of the query window, but also on the domain size of the partitioning attribute. This is because the latter decides the size of the join result. Given that $a3$ has a domain of size 100, a random E_2 tuple will match a given E_1 tuple with probability 0.01. We study the effect of domain size in our next experiment. Attributes $a1, a2, a3, a4, a5$ have domains of sizes $10, 50, 100, 500, 1000$ respectively. Hence, going from $a1$ to $a5$, as the partitioning attribute, represents decreasing work (decreasing

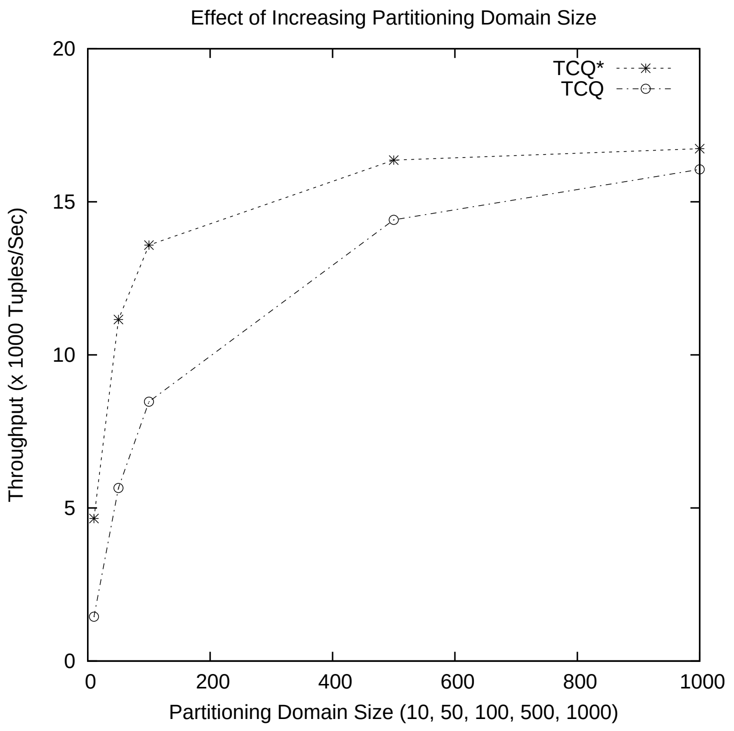

Figure 8: Varying Partitioning Domain Size

result size). Our experiment, illustrated by Figure 8, confirms this. We vary the domain size using these attributes, while the window size is kept constant at 10,000 tuples (the sequence length is still 2). As we go from *right to left* on the x-axis, TCQ performs increasingly more work than TCQ*, and suffers in throughput.

Our final experiment varies the sequence length in the query. We set the window size to 1000 tuples, and choose $a1$ as the partitioning attribute (domain size 10). As Figure 9 shows, TCQ suffers much more than TCQ*, as the sequence length is increased from 2 to 3. This is because a sequence of length 3 requires a 3-way join in TCQ, which is even more work than the previous experiments. On the other hand, TCQ* uses a sequence-aware tree data structure described earlier, and performs only as much work as needed - an E_3 event is used to probe the IESteM which consists of composite E_1E_2 events (which have already occurred), while TCQ will possibly produce intermediate tuples for all subsets of the 3-way join.

3.4.4 Summary

In summary, our experiments confirm that providing first-class support for sequence-aware operators in stream processing engines is important for Complex Event Pro-

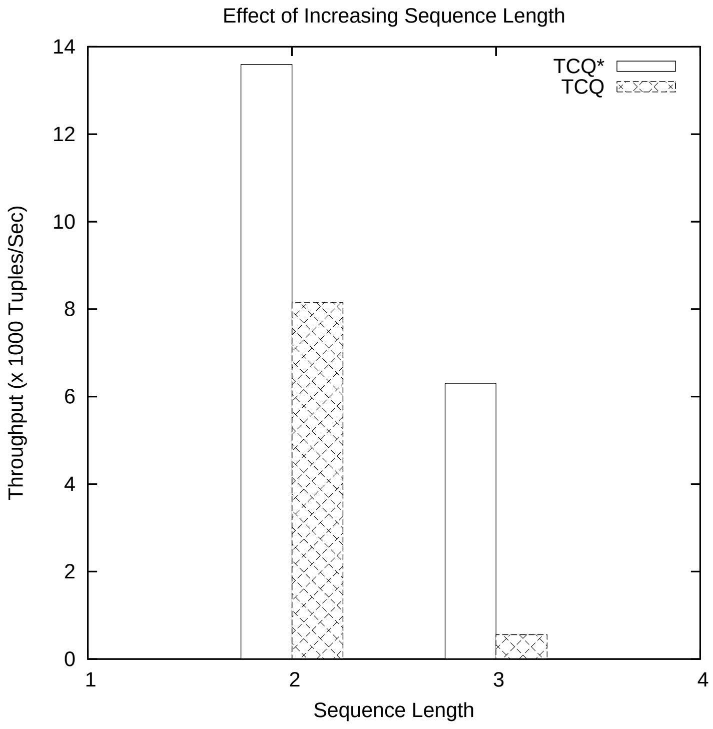

Figure 9: Varying Sequence Length

cessing over streams. While it is possible to write an event query as a complicated join, scalable solutions will require event operators next to traditional relational operators (Note however, that while our added language constructs allow easy expression of event queries, it is possible to use our sequence-aware operator without the new language features. For example, a smart query planner may detect the presence of sequencing in a normal SQL query, and use our operator in place of the normal join operator). Further, support for negation is more naturally provided[7] using event operators, like our NOTFOLLOWS and NOTPRECEDES operators.

3.5 Limitations

While our initial experiments demonstrated the potential for significant performance gains, we would like to point out the following limitations in our approach to complex event processing over streams.

[7]Note that, although TCQ does not currently support outer joins, it is trivial to define and implement windowed outer joins in a stream processor.

1. **Expressiveness:** The binary nature of our event correlation operators represents only a first take. To express correlation among more than two events, more generic forms of sequencing and negation are needed. For example, to define a semantic window of the form given in Figure 1(d), which has two events specifying its ends, our binary operators fall short.

2. **Orthogonality:** Our extensions to streaming SQL are in the form of a new EVENT clause. This makes complex event queries take a different *form* from normal streaming queries. A cleaner alternative, which preserves the orthogonality of the language constructs, is needed[8].

3. **Sharing:** Our event aggregation operator currently does not perform sharing across queries. In general, if two queries lead to event trees having *common* subtrees, then work can be shared at run-time. However, some details, like sharing operators with the same event arguments but different time windows, need to be looked at carefully.

4 Probabilistic Complex Event Processing (PCEP)

Our discussion so far involved processing of *complete* or *precise* data streams. We now move to Probabilistic Complex Event Processing, which is our work on processing *imprecise* or *fuzzy* data streams, such as those produced by current day sensing technology.

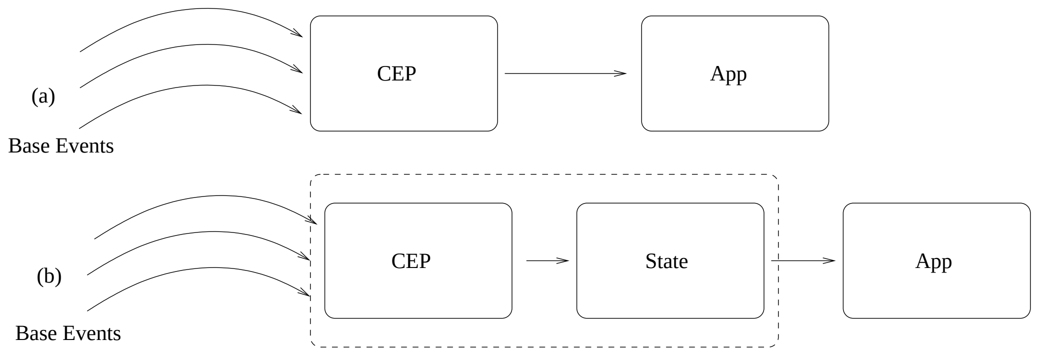

Figure 10: The Role of State in Event-driven Applications.

Figure 10 shows two ways of looking at a Complex Event Processing system. Figure 10 (a) is the view to which all our discussion up to this point conforms. The

[8]To be more accurate, the lack of orthogonality in SQL has been long-known [10]. However, we would still like to have better integration of complex events with SQL.

CEP engine sits between the application and low-level streams, and detects complex patterns over these streams. Once the required pattern is detected, the subscribing application is informed about this, along with a set of bindings (to the component events that make up the complex event), which the application uses. An example of such an application is one that wants to detect a "three day period where Microsoft stock price first goes up by 10% and then goes down by more than 10%" in a stock quote stream.

However, this model fails for some applications. For example, consider a context-aware ubiquitous application for the smart home which wants to display content in the living room based on who the current occupants of the room are. Urgent home state notifications, like a leaking water pipe, need to be reported whenever a senior member of the household is present. Individual-specific reminders and calendar information are other examples. This application *senses-and-responds* to the current living room state, rather than to individual events like motion sensor firings. If the current living room state consists of person A and B being present in the living room, the application does not care about *how this state was reached*, i.e., it does not care about *what sensor firing events* lead to this state. As another example, consider a warehouse scenario. An application wants to take action whenever a shelf gets understocked. Depending on how many items are remaining on the shelf, it places an order for more items through the supply chain for delivery within a week, or the next day itself. This application is also based on the current state of the warehouse, and not on events like items coming in or going out of the garage door.

This paradigm is captured in Figure 10 (b), where the CEP engine feeds events into a state abstraction. Applications then respond to state changes. Note that this state abstraction is not equivalent to just *high-level events* but captures the current configuration of the system, based on *all* the events seen in the past. In this section, we work with this state-based Complex Event Processing model.

4.1 The Need for Probabilistic Complex Event Processing

Real-world applications also face another set of issues because of sensor data sources. The data streams from sensor devices are often dirty and fuzzy [14, 15]. Events based on these data streams are therefore probabilistic. For example, a sensor net that monitors the temperature of a room and generates temperature readings with some confidence (based on the number of sensors nodes and link failure rates) can only detect a *fire* event, defined as "the temperature being above 70C with a 95% confidence" with a probability tag. This probability will be a function of the event specification and underlying data stream's dirtiness. If events are probabilistic, the state that results from these events is also, in general, probabilistic or fuzzy. This calls for *Probabilistic Complex Event Processing.*

	No Semantic Hierarchy	**Semantic Hierarchy**
Crisp Data	No	Yes
Uncertain Data	Yes	Yes

Figure 11: The Need for Probabilistic Complex Event Processing.

Another interesting feature of real-world data sources is that they produce events at "different semantic levels". For example, a simple motion sensor can only detect movement of "object". On the other hand, RFID-based sensing can detect the presence of "specific persons" based on ID-based RFID tags. Vision techniques, using machine learning over a camera's video stream, can detect people inside a room, however, they may give only a distribution for who the person is. In summary, not all event or data sources work at the same semantic level. Hence, a Complex Event Processing system needs to incorporate event streams at different semantic levels. Note that events at any semantic level are fuzzy when seen from a higher semantic level. For example, a "person enters home" event is probabilistic with respect to the "person A enters home" level events. Hence, probabilistic reasoning is necessary even when the data is crisp, if we are trying to make inferences at a higher semantic level. This also calls for PCEP.

Figure 11 summarizes our arguments. PCEP is not needed if all event sources report at the same semantic level, and simple events are reported in a crisp fashion. This corresponds to the upper left corner of the table. All other cells in the table argue for PCEP.

4.2 An Architecture for Probabilistic Complex Event Processing

Now we present our preliminary work on an architecture for Complex Event Processing that incorporates probabilistic events and state. Figure 12 illustrates our architecture.

Base events are fed into the system from the bottom. These consist of low-level sensor data streams. The other components are described as follows:

- **Learning-based Event Refinement:** This component refines base event streams based on past information and Machine Learning techniques. For example, if a "person enters home" event is seen in the base stream, this component can make use of archived information and models, like which home occupant is most likely to come home at this time (on this day), to create a distribution for the identity of the person. We expect this component to play an important role as event sources often give incomplete and fuzzy information.

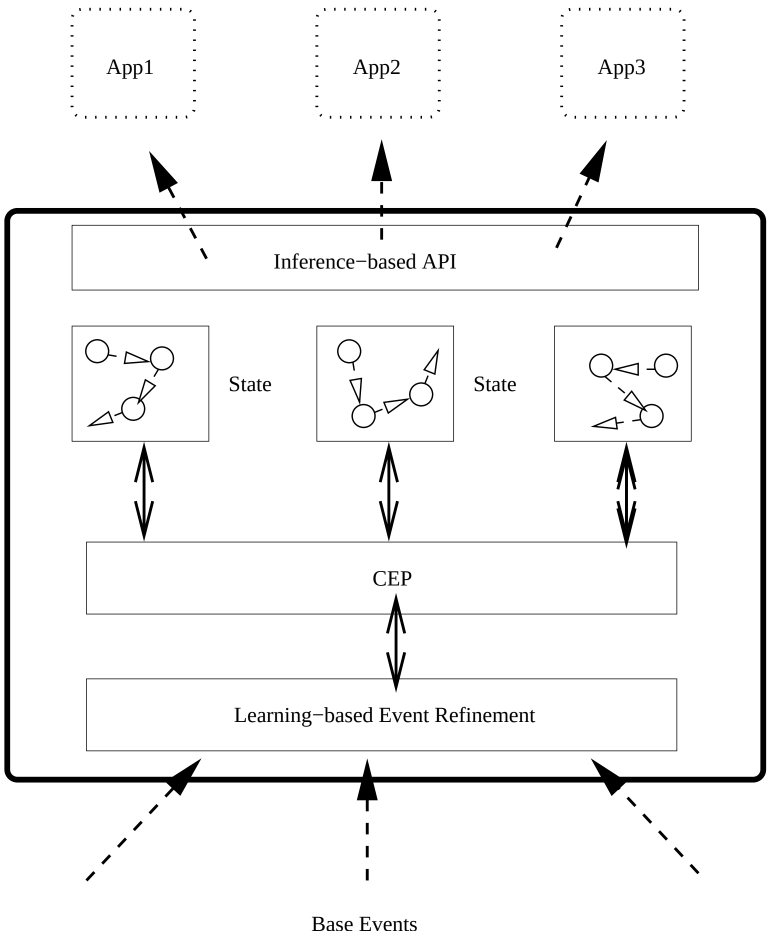

Figure 12: An Architecture for Probabilistic Complex Event Processing.

- **CEP Engine:** This is the core complex event processing engine. It processes multiple event streams to filter, correlate and aggregate them into semantically-richer events. Events from this module go into either the state maintenance module, or the above mentioned event refinement module, for refinement of complex events.
- **Probabilistic State:** This module is responsible for maintaining the "current state of the system". As mentioned earlier, state under fuzzy events is necessarily fuzzy or probabilistic. This module decides a new state based on the old state and new events seen. Some interesting examples are:
  1. If the current state involves "only person A being in the living room" and a new motion event is seen saying "some person has moved from the living

room to the den", then the module's new state reflects "person A being in the den".

2. If the current state involves "either person A or person B being in the bedroom, each with probability 0.5", a high-certainty event that points to "person A being in the bedroom" will lead to a new state with modified probabilities (0.9 for person A, and 0.1 for person B being in the bedroom). This is an example of how new events can reduce the uncertainty caused by past events.

- **Inference-based API to the Probabilistic State:** Applications built on top of this PCEP framework need some kind of API to access the current (probabilistic) state. For example, if the state corresponds to a *people-in-rooms* tracker for a smart home, then, an application may want to ask questions like:

  1. What is the number of people in the home currently?
  2. Who is in the bedroom currently?
  3. Is Person A in the bedroom?

  In general, the answers to these questions are probabilistic. For example, suppose the home has two rooms - living room (LR) and bedroom (BR). Further, suppose we have only two people in the household, person A and person B. Then, let random variables X_A and X_B capture the current state w.r.t. person A and person B. These two variables take values out of the set $\{LR, BR, Outside\}$. The current state is then given by the (discrete) joint probability mass function:

$$p(x_A, x_B)$$

  At any point in time, the probability that "person A is in the living room and person B is outside the home" is given by $p(LR, Outside)$. Along similar lines, the answers to the above three questions are given by:

  1. $0 \times p(Outside, Outside)$
     $+ 1 \times (p(LR, Outside) + p(BR, Outside) + p(Outside, LR) + p(Outside, BR))$
     $+ 2 \times (p(LR, LR) + p(LR, BR) + p(BR, LR) + p(BR, BR))$
  2. "Only person A" with probability $p(BR, Outside) + p(BR, LR)$.
     "Only person B" with probability $p(Outside, BR) + p(LR, BR)$.
     "Both A and B" with probability $p(BR, BR)$.
     "No one" with probability $p(Outside, Outside) + p(Outside, LR) + p(LR, Outside) + p(LR, LR)$.

3. "Yes" with probability $p(BR, Outside) + p(BR, LR) + p(BR, BR)$.
   "No" with probability $p(Outside, Outside)+p(Outside, LR)+p(Outside, BR)+p(LR, Outside) + p(LR, LR) + p(LR, BR)$.

  These answers are generated by the familiar operations of *marginalization* and *expectation calculation* over the joint probability mass function. We expect such operations to be a fundamental part of the API to the probabilistic state.

- **Apps:** These are the actual applications are are written on top of the CEP-generated probabilistic state.

Based on this architecture, we built a small prototype for a smart home scenario. The simple event streams reported the movement of people as they moved from one room to another. The probabilistic state was maintained using a *probabilistic state machine*, which is a state machine that uses a distribution over all its states for the notion of *current state*. The application used was a ubiquitous computing application that displays content on the living room display, based on the person(s) currently in the living room.

5 Other Related Work

In this section, we overview related work that is relevant but not directly applicable to the problem addressed in this paper. These are the proposals for query languages (and operators) for sequence or time-series data [25, 29, 19, 28].

5.1 Query Processing for Sequence Data

While SQL is the inter-galactic standard for querying relational data, the fact that relations are "sets", limits its applicability to other kinds of data. An important class of data is sequence data, like time-series from scientific experiments and a sequence of stock quotes from a stock exchange. As we will see below, using native SQL for querying such data will result in two problems:

1. Expressing SQL queries for sequence data results in unintuitive queries, which are hard to write, and understand.

2. An unintuitive query is hard to optimize, resulting in inefficient query processing.

For example, suppose we have a table with stock quotes, which stores prices (*end of trading day* prices) of stocks, along with the timestamp, with the following schema:
`Quotes(name varchar(10), price integer, date timestamp)`

Note that this data is *inherently* sequential but the underlying data model (relational) is not aware of this. Suppose we want to find out all those stocks that went up by at least 10% in one day, and then another 10% on the next day, then this can be done in SQL as follows:

```
SELECT q3.*
FROM Quotes q1, Quotes q2, Quotes q3
WHERE q1.name = q2.name AND q2.name = q3.name
AND q1.date = q2.date - '1 DAY'
AND q2.date = q3.date - '1 DAY'
AND q2.price >= 1.1 * q1.price
AND q3.price >= 1.1 * q2.price
```

This query has to do a three-way join because the data model is not sequence-aware. On surface, the query is hard to read and understand, as it is written unintuitively. The relational optimizer will also find it hard to optimize.

This has resulted in several proposals for query languages geared towards querying sequence data. Given the similarity between sequence data and (sequence-aware) event histories, we overview this interesting research space in this section.

5.1.1 SQL-TS

Simple Query Language for Time Series (SQL-TS) has been proposed in [28]. The authors present SQL extensions that let users express such queries in a natural way, and also study the optimization of sequence queries written in their language. In SQL-TS, the above query will be phrased as:

```
SELECT Z.*
FROM Quotes
CLUSTER BY name
SEQUENCE BY date AS (X, Y, Z)
WHERE Y.price >= 1.1 * X.price AND Z.price >= 1.1 * Y.price
```

The CLUSTER BY clause specifies that data items corresponding to each *name* should be processed separately (a *group by* like functionality). The SEQUENCE BY clause specifies which attribute determines the ordering of data for the current query. The AS clause is used to specify a sequence of tuple variables from the specified table.

Note that (X, Y, Z) will be bound to three tuples that immediately follow each other. SQL-TS also allows the expression of recurring patterns (patterns involving a variable number of tuples) using the $*$ operator. For example, to find the maximal period of time where the price of the stock fell more than 10%, one can use the query:

```
SELECT X.name, X.date, Z.date
FROM Quotes
CLUSTER BY name
SEQUENCE BY date AS (X, *Y, Z)
WHERE Y.price < Y.previous.price AND Z.previous.price <= 0.9 * X.price
```

Note that two additional fields are provided for every tuple, that allow addressing the immediately previous and next tuples.

5.1.2 SRQL

Sorted Relational Query Language (SRQL) [25] treats sequences as sorted relations. SRQL consists of extensions to SQL, and the work also presents an algebra over sequences that extends relational algebra.

A *simple sequence* is a relation that is (logically, not physically) sorted based on a set of attributes. A *composite sequence* is a relation that is first grouped on a set of attributes, called *grouping attributes* and then (logically) sorted by another set of attributes within each partition. An alternative way to look at this, is to consider a sequence as a relation with a set of grouping and sequencing attributes (which may be empty). Every tuple in a sequence has an implicit attribute, called the *ordinal number*, which defines the sequence number of the tuple in the sequence (or within its specific group). This number can also be queried. Note that multiple tuples (within the same group) may have the same ordinal number, as they may have the same value for all sequencing attributes.

At its core, the SRQL consists of the standard relational operators $(\sigma, \Pi, \times, \cup, -)$, extended to work on sequences (by defining the grouping and sequencing attributes for the output to be empty) and one new sequencing operator (Ψ) that creates a sequence. The sequencing operator takes as input a sequence or relation R and (re)sequences it based on attribute sets g and s. That is, $\Psi_{g,s}(R)$ is R grouped on g and then sequenced on s. Some other sequence-aware operators are also presented in [25] but they can be expressed using the operators mentioned above. An SRQL query built on top of this algebra may look like:

```
SELECT S.price, SHIFTALL(S,-1).price
FROM Quotes GROUP BY name SEQUENCE BY date AS S
```

```
WHERE S.price >= 1.1 * SHIFTALL(S,-1).price
```

This query returns all stocks which went up by 10% within one day. The SHIFTALL(t,n) keyword gives access to the tuple t' that is at an offset of n with respect to the ordinal number of tuple t, within the same partition. The join operation for sequences can be seen in the query:

```
SELECT C.date, Q.name, Q.price
FROM MarketCrash C, Quotes SEQUENCE BY date AS Q
WHERE Q.date <= C.date
AND (SHIFTALL(Q,1).date > C.date OR SHIFTALL(Q,1).date IS NULL)
```

which finds for every stock market crash, the last stock that was reported before the crash. Finally, sliding window aggregation over sequences is demonstrated by the query:

```
SELECT Q.name, Q.date, AVG(price) OVER -1 TO 0
FROM Quotes GROUP BY name SEQUENCE BY date AS Q
```

This query finds, for each stock and each date, the average stock price over two days (the previous day and current day). The OVER keyword in the SELECT clause specifies the range of the sliding window (in this case, from offset -1 to the current tuple).

5.1.3 AQuery

AQuery [19] supports order in data in a ground-up fashion. AQuery's data model consists of a collection of arrays, called *arrables*. A sorted relation is a special case of an arrable where the arrays (columns) have scalar elements. This corresponds to column-oriented semantics rather than row-oriented semantics. Well-known operators, like projection and selection, have been extended to their order-preserving variants. The FROM clause is extended to specify the ordering attributes, and the SELECT clause allows order-aware operations, as demonstrated in this query:

```
SELECT P.name, last(10, Q.price)
FROM Portfolio P, Quotes Q ASSUMING ORDER date
WHERE P.name = Q.name
GROUP BY P.name
```

This query gives the last 10 quotes for each stock in the person's portfolio. Note that the *last*() function pulls out the last 10 elements from the *price* array.

While work on sequence query processing extends the relational data model and operators with support for time-series data, it does not focus on streaming data or event processing. Hence, these efforts do not provide support for essential features like negation.

6 Conclusions and Future Work

In this work, we have looked at the problem of Complex Event Processing, which deals with aggregating simple events into more meaningful and semantically-richer events. This problem has been looked at in the past, for example, in the context of active databases. However, the challenges of streaming and uncertain data add new dimensions to the problem. We presented an overview of past work in Complex Event Processing. We also described our work in Complex Event Processing over streaming data, which has been implemented in the TelegraphCQ data stream processor and used in a recent demonstration. An architecture for Probabilistic Complex Event Processing (PCEP) was presented, to handle scenarios where either simple events are not reported in a crisp fashion, or simple events are reported at different levels of a semantic hierarchy.

Future work includes extending support for Complex Event Processing over streaming data, by having more generic (n-ary) operators and better (orthogonal) language constructs. For applications that want to execute multiple event queries simultaneously, sharing of work between different queries needs research. Our architecture for Probabilistic Complex Event Processing is preliminary, and requires instantiation in multiple scenarios, to be tested widely.

Acknowledgements

The author is thankful to his research advisor, Prof. Michael Franklin, for his guidance throughout this work. Dr. Kurt Brown and Dr. Minos Garofalakis provided valuable insights during the work on Probabilistic Complex Event Processing. Members of the Telegraph and HiFi project groups at Berkeley were consistent sources of help during the work on implementing event processing in the TelegraphCQ system. Finally, the author is thankful to all the members of the Berkeley Database Group.

References

[1] Complex event processing. http://www.complexevents.com.

[2] D. J. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A New Model and Architecture for Data Stream Management. *The VLDB Journal*, 12(2):120–139, 2003.

[3] R. Adaikkalavan and S. Chakravarthy. Snoopib: Interval-based event specification and detection for active databases. In *ADBIS*, pages 190–204, 2003.

[4] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, R. Motwani, I. Nishizawa, U. Srivastava, D. Thomas, R. Varma, and J. Widom. STREAM: The Stanford Stream Data Manager. *IEEE Data Eng. Bull.*, 26(1):19–26, 2003.

[5] A. Arasu, S. Babu, and J. Widom. CQL: A Language for Continuous Queries over Streams and Relations. In *DBPL*, 2003.

[6] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S. K. Kim. Composite events for active databases: Semantics, contexts and detection. In *VLDB '94: Proceedings of the 20th International Conference on Very Large Data Bases*, pages 606–617, 1994.

[7] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. A. Shah. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *CIDR 2003, First Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA*, 2003.

[8] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, V. Raman, F. Reiss, and M. A. Shah. Telegraphcq: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

[9] O. Cooper, A. Edakkunni, M. J. Franklin, W. Hong, S. R. Jeffery, S. Krishnamurthy, F. Reiss, S. Rizvi, and E. Wu. HiFi: A Unified Architecture for High Fan-in Systems. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases (VLDB), Toronto, Canada*, pages 1357–1360, 2004.

[10] C. J. Date. A critique of the sql database language. *SIGMOD Rec.*, 14(3):8–54, 1984.

[11] M. J. Franklin, S. R. Jeffery, S. Krishnamurthy, F. Reiss, S. Rizvi, E. Wu, O. Cooper, A. Edakkunni, and W. Hong. Design Considerations for High Fan-In Systems: The HiFi Approach. In *CIDR 2005, Second Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA*, pages 290–304, 2005.

[12] M. J. Franklin, S. R. Jeffery, S. Krishnamurthy, F. Reiss, S. Rizvi, E. Wu, O. Cooper, A. Edakkunni, and W. Hong. Design Considerations for High Fan-In Systems: The HiFi Approach. In *CIDR*, 2005.

[13] N. H. Gehani, H. V. Jagadish, and O. Shmueli. Composite event specification in active databases: Model & implementation. In *Proceedings of the 18th International Conference on Very Large Databases*, 1992.

[14] S. R. Jeffery, G. Alonso, M. J. Franklin, W. Hong, and J. Widom. Virtual Devices: An Extensible Architecture for Bridging the Physical-Digital Divide (to appear in Pervasive 2006). Technical Report UCB/CSD-05-1375, EECS Department, University of California, Berkeley, 2005.

[15] S. R. Jeffery, G. Alonso, M. J. Franklin, W. Hong, and J. Widom. A Pipelined Framework for Online Cleaning of Sensor Data Streams. In *Proceedings of the 21st International Conference on Data Engineering (ICDE)*, 2006.

[16] Q. Jiang, R. Adaikkalavan, and S. Chakravarthy. Estreams: Towards an Integrated Model for Event and Stream Processing. Technical Report CSE-2004-3, University of Texas at Arlington, July 2004.

[17] S. Krishnamurthy, S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Madden, F. Reiss, and M. A. Shah. Telegraphcq: An architectural status report. *IEEE Data Eng. Bull.*, 26(1):11–18, 2003.

[18] Y.-N. Law, H. Wang, and C. Zaniolo. Query languages and data models for database sequences and data streams. In *VLDB*, pages 492–503, 2004.

[19] A. Lerner and D. Shasha. AQuery: Query Language for Ordered Data, Optimization Techniques, and Experiments. In *VLDB*, pages 345–356, 2003.

[20] D. C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[21] D. C. Luckham and B. Frasca. Complex event processing in distributed systems. Technical Report CSL-TR-98-754, Stanford University, 1998.

[22] D. C. Luckham, J. L. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and analysis of system architecture using rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, 1995.

[23] S. Madden and M. J. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *ICDE*, pages 555–566, 2002.

[24] S. Madden, M. A. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *SIGMOD Conference*, 2002.

[25] R. Ramakrishnan, D. Donjerkovic, A. Ranganathan, K. S. Beyer, and M. Krishnaprasad. SRQL: Sorted Relational Query Language. In *10th International Conference on Scientific and Statistical Database Management, Proceedings, Capri, Italy, July 1-3, 1998*, pages 84–95. IEEE Computer Society, 1998.

[26] V. Raman, A. Deshpande, and J. M. Hellerstein. Using state modules for adaptive query processing. In *ICDE*, 2003.

[27] S. Rizvi, S. R. Jeffery, S. Krishnamurthy, M. J. Franklin, N. Burkhart, A. Edakkunni, and L. Liang. Events on the Edge. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 885–887, 2005.

[28] R. Sadri, C. Zaniolo, A. Zarkesh, and J. Adibi. Optimization of Sequence Queries in Database Systems. In *PODS '01: Proceedings of the twentieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 71–81, 2001.

[29] P. Seshadri, M. Livny, and R. Ramakrishnan. The Design and Implementation of a Sequence Database System. In *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India*, pages 99–110. Morgan Kaufmann, 1996.

[30] P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras. Exploiting punctuation semantics in continuous data streams. *IEEE Transactions on Knowledge and Data Engineering*, 15(3):555–568, 2003.

[31] J. Widom and S. Ceri. *Active Database Systems: Triggers and Rules For Advanced Database Processing.* Morgan Kaufmann, 1996.

[32] D. Zimmer and R. Unland. On the Semantics of Complex Events in Active Database Management Systems. In *Proceedings of the 15th International Conference on Data Engineering (ICDE)*, pages 392–399. IEEE Computer Society Press, 1999.