# RFID Data Processing with a Data Stream Query Language

Yijian Bai *
Department of Computer Science
UCLA
bai@cs.ucla.edu

Fusheng Wang Peiya Liu
Integrated Data Systems Department
Siemens Corporate Research
{ fusheng.wang, peiya.liu}@siemens.com

Carlo Zaniolo Shaorong Liu †
Department of Computer Science
UCLA
{zaniolo, sliu}@cs.ucla.edu

## Abstract

*RFID technology provides significant advantages over traditional object-tracking technology and is increasingly adopted and deployed in real applications. RFID applications generate large volume of streaming data, which have to be automatically filtered, processed, and transformed into semantic data, and integrated into business applications. Indeed, RFID data are highly temporal, and RFID observations form complex temporal event patterns which can be very different for various RFID applications. Thus, it is desirable to have a general RFID data processing framework with a powerful language, for the end users to express a variety of queries on RFID data streams, as well as detecting complex events patterns. While data stream management systems (DSMSs) are emerging for optimized stream data processing, they usually lack the language construct support for temporal event detection. In this paper, we discuss a stream query language to provide comprehensive temporal event detection, through temporal operators and extension of sliding-window constructs. With the integration of temporal event detection, a DSMS has the capability to serve as a powerful system for RFID data processing.*

## 1. Introduction

RFID is an Automatic Identification and Data Capture (AIDC) technology that uses RF waves to transfer data between a reader and an object for the purpose of identifying, categorizing, and tracking the object. RFID is fast, reliable, and does not require line of sight or contact between readers and tagged objects. With such advantages, RFID is gradually being adopted and deployed in various applications, such as supply chain systems, warehouses management, security, hospitals, highway tolls, etc.

An RFID system consists of RFID readers with antennas, host computers, and transponders or RF tags which are recognized by the readers. An RFID tag is uniquely identified by a tag ID stored in its memory and can be attached to almost anything. Such IDs are specified through the EPC (Electronic Product Code) standard [4]. In addition, RFID technology can also be used in conjunction with sensors that measure varieties of physical measurements, such as sensors for temperature, humidity, blood pressure, etc, which provide extra information for the entity uniquely identified by the RFID tag.

RFID technology makes it possible to i) collect large amount of data for tracking and identifying physical objects along their history [22] and ii) real-time monitor physical objects and their environment for monitoring applications [23]. While RFID observations are simple primitive events (consisting of reader's EPC code, observed tag ID and the observation timestamp), RFID observation streams from multiple readers form complex event patterns—mostly temporal in nature [22, 23]—to represent business application logic. This poses a significant challenge for RFID data processing for the following requirements of: i) automatically filtering, interpreting and transforming raw RFID observation data into semantic business logic data; ii) real-time monitoring and querying physical objects and their environment; iii) the capability to process high volume RFID data streams; and iv) minimal effort to integrate RFID data into existing business applications and convenient interfaces for end users. The importance of RFID data stream processing is also emphasized by the newly released RFID Application Level Event (ALE) standard [1]: a common interface to process raw RFID events, including data filtering, windows-based aggregation, and reporting.

In this paper, we try to address the problem of effective processing of RFID data streams. The problem of RFID event processing is first tackled in [23], where a declarative rule based language and a standalone event engine was proposed and developed to automatically process RFID events. Such approach is capable of detecting complex temporal-pattern-based high level events, however it has several diffi-

*Work partially done while visiting Siemens Corporate Research
†Work done while visiting Siemens Corporate Research

culties in terms of the expressive power and generality.

The first shortcoming for a standalone event engine as proposed in [23] is the limited expressive power. In [23], a few temporal RFID event constructors are defined and RFID ECA rules are defined by combining such constructs. This highly specialized language cannot handle a large class of general data inquiries, thus requiring other separate systems and increasing the cost of integration. For example, if a manager needs to find out the real-time location of all product-carrying carts labelled by RFID tags in a warehouse, this will require a query language and system that can request live data from RFID readers. An event processing model such as proposed in [23] is not built for such inquiries. Second, the event processing model cannot support EPC-code pattern based grouping and aggregation queries. This is one of the major requirements of ALE standard. Consider the following example from the ALE standard specification—we need to aggregate readings on all EPC tags conforming to the following pattern: *20.*.[5000-9999]*. Here in the EPC tag pattern *20* is the ID of the company, * matches any product of the company, and *[5000-9999]* matches a valid range of serial numbers between 5000-9000. (Thus we are interested in everything from this company with serial numbers between 5000 and 9000.) This aggregation is difficult for the event system in [23] to support. Third, for high volume RFID events, sliding windows are essential in many applications. However, windows are not natural constructs in traditional event processing systems and could require complex condition-checking. Fourth, there is no established standard for event-based declarative languages, thus increasing the learning curve for the end-users (who might otherwise have a good chance to be familiar with the SQL language, which is widely used due to the popularity of relational databases). Last, the event processing engine RCEDA in [23] takes a simple graph-based processing model and lacks optimization techniques for large volume RFID event data processing.

On the other hand, data stream applications and data stream management systems (DSMSs) [7, 13, 12, 19, 3] are under rapid developments and have unique advantages. Such systems are designed to answer realtime or near-realtime continuous queries on potentially unbounded data streams, and the issues such as resource allocation, query-optimization, QoS are addressed at the system level, allowing application programmers to focus on business logic programming. Some systems use query language largely conforming to the well-established SQL syntax and semantics. Furthermore, RFID tag readings can be seen as continuously-generated relational data streams, which may be treated as append-only, structured data tuples that conform to a specified data schema. Using a DSMS we can apply data transformations/event detection logic on the tag data, producing another data stream with cleaner or enhanced data, or leading to actions such as persistent database updates, alerts, and so on. Indeed, an SQL-based stream query language provides many potential benefits and conveniences for RFID data processing.

However, current DSMSs do not support temporal event processing, which is a major requirement in RFID applications. Therefore, in this paper, we propose a stream query language to provide comprehensive temporal event detection, through temporal operators and extension of sliding-window constructs. By integrating temporal event detection, DSMS becomes a powerful system for RFID data processing.

In this paper we discuss the following:

- We first analyze RFID data processing tasks, and identify the types of tasks that are well supported by a SQL-based stream query language in a DSMS;
- Then we explore the limitation of SQL in supporting temporal event detection;
- We discuss how we extend an SQL-based stream query language with temporal event operators. We also explain how to utilize the SQL:2003 sliding window construct, the *FOLLOWING* window, to support negative temporal events;
- We finally demonstrate the power of our language with RFID application scenarios.

## 2. RFID Data Processing in a SQL-Based Stream Query Language

RFID data are generally timestamped tag readings with unique IDs. Some of the common RFID data processing tasks are very well-suited for an SQL-based stream query language such as the Expressive Stream Language (ESL) [2, 8]). Next, we discuss some of these common tasks.

### 2.1. RFID Data Inquiry and Transformation

Some of the common tasks involved in RFID data processing include *duplicate elimination*, *ad-hoc queries*, *context retrieval* for tag IDs, *database updates* and *data aggregation*. As we will show next, all of these tasks are very suitable for SQL-based query languages.

**Duplicate Elimination** Duplication is common in RFID data, since it can be used to compensate for missed tag readings. Duplications can be caused by duplicated tags, duplicated readers, or simply repeated reads on the same tag. Suppose we have a data stream **readings** with the schema below, and we want to derive another stream **cleaned_readings** with the same schema from which the duplicates are eliminated (for simplicity the data types are omitted in all examples):

STREAM **readings(reader_id, tag_id, read_time)**;

To eliminate the duplicates, we can use the following criteria: identical tag readings that appear within a given time-period threshold (say, 1 second) can be regarded as the same reading. This can be easily coded in a DSMS as a single-stream transducer as shown in Example 1. In this example we use the syntax of the ESL stream query language [2, 8], and use a sliding window construct of 1 second to handle the time threshold. (A single-stream transducer in a DSMS is a continuous query that takes in a tuple, and produces tuples into another data stream.)

**Example 1** *Duplicate Filtering with Join*

```
INSERT INTO cleaned_readings
    SELECT * FROM readings AS r1
    WHERE NOT EXISTS
    (SELECT * FROM TABLE( readings OVER
       (RANGE 1 seconds PRECEDING CURRENT)) AS r2
       WHERE r2.reader_id = r1.reader_id
       AND r2.tag_id = r1.tag_id )
```

**Ad-hoc Queries** Ad-hoc queries on the data streams represent an important category of queries that are essential for RFID applications that often need to provide current status information to the end-user. An SQL-based stream query language in a DSMS system that supports ad-hoc *snapshot* queries provides a well-accepted language syntax to the end-user for such queries. For example, in a patient-tracking application, the current location of the patient represented by tag readings from readers at various locations may need to be queried directly by a physician. This should be done without having to store such location data all the time in a persistent database, which will cause a large storage overhead. The SQL syntax here for such kind of ad-hoc queries is very similar to regular database queries, thus examples are omitted.

**Context Retrieval** While different types of RFID tags have different capacities, in most cases it is not practical, or safe, to carry along on the tag all the information necessary for business interpretation. Therefore, meta-data lookup is almost always needed. For example, when a person carries an item out of a gate, a database lookup might be needed to determine whether this person is authorized for taking out this item, and such information can not be carried on the tag for every possible item. Such information has to be retrieved before the tag readings can be further interpreted and processed. The look up may be conveniently expressed in a SQL in a DSMS that supports stream-DB spanning queries. These context-lookup queries can reside in the system as continuous queries, where incoming tag readings are enhanced to include the lookup result, which again forms a data stream with more complete information that can be further used by other queries.

**Database Update** In a RFID based application, one common scenario is that we may constantly persist information from some of the tag readings to a persistent database. For example, we may determine the product locations travelling in a warehouse and track the movement history of objects by selectively sending some tag readings to a database table based on some predefined criteria. Such updates can be naturally expressed as stream-DB spanning UPDATE queries in the ESL language. As a simple example, suppose we have a data stream **tag_locations** which contains the id and location of incoming readings. Then a persistent database table **object_movement** is used to track history information for an object, where a new row is not added to the DB unless the object location changes from last time reported.

STREAM **tag_locations(readerid, tid, tagtime, loc)**;
TABLE **object_movement(tagid, location, start_time)**

Then the following continuous query can be defined on the data stream Tag_readings, which inserts the reading into the persistent table every time the location changes.

**Example 2** *Location Tracking*

```
INSERT INTO object_movement
    SELECT tid, loc, tagtime
    FROM tag_locations WHERE NOT EXISTS
      (SELECT tagid FROM object_movement
        WHERE tagid = tid AND location = loc);
```

**Data Aggregation** Data Aggregation could be needed for various RFID data processing tasks. For example, we may need to count the number of products passing through the door every hour, or monitor the max/min blood pressure of a patient throughout the day. (The blood pressure itself is not RFID data, but it can be sensor data that are associated with the RFID identifications.) Simple built-in aggregation operators are provided by most SQL-based stream query languages. Furthermore, some languages, such as the ESL language, support User Defined Aggregates (UDAs) and User Defined Functions (UDFs). In particular, ESL also allows users to express UDAs in native SQL. These constructs enable the end-users to perform arbitrarily complex aggregation tasks in the SQL-based query language.

For example, suppose we like to perform EPC-code based data inquiry and aggregation. We want to get the total count of all EPC codes conforming to the pattern *20.*.[5000-9999]*, where *20* is the ID of the company, * matches any product of the company, and *[5000-9999]* matches a valid range of serial numbers between 5000-9000. We could use a combination of SQL's built-in string-matching support and UDFs to perform this aggregation. We use the stream **readings** defined above,

where tid is a formatted epc number in the form of "company.productcode.serialnumber". In the following code, **extract_serial** is a UDF defined separately, which can be used to extract the serial number part of the EPC and return it as an integer.

**Example 3** *EPC Code Pattern Based Aggregation*

```
SELECT count(tid) FROM readings
  WHERE tid LIKE '20.%.%'
  AND extract_serial(tid) ≥ 5000
  AND extract_serial(tid) ≤ 9999
```

As we can see from the above examples, an SQL-based stream query language contains constructs and extension mechanisms that can be naturally used for many purposes in RFID data processing. However, one of the important aspects of RFID data processing—temporal events detection—lacks direct support in SQL, as we discuss next.

## 2.2. Temporal Events Detection in RFID Data

Detecting temporal events is frequently an essential component of an RFID application. Consider the following example, where RFID readers are used to automatically detect the presence of products, packaging boxes, personnel, etc, in a warehouse. The containment relationship between products and packing cases can be arranged to be detected as follows:

**Example 4** *Say that reader r1 scans products to be packed, and reader r2 scans packing cases. If a sequence of readings from reader r1 is followed by a distinct reading from reader r2 within time t0 (e.g., 5 seconds), then we can conclude the products observed by r1 are contained in the case observed by r2, as illustrated in Figure 1(a).*

*Furthermore, the products for the next case may start to be detected before the previous packing case is detected. To distinguish products that belonging to different cases, we mandate that products being packed into the same case are detected immediately following each other—i.e. the gap of detection time between consecutive products for the same case is below a certain time threshold $t_1$ seconds (e.g. 1 second). While, products belong to the next case have a gap of arrival longer than the threshold (Figure 1(b)).*

This temporal pattern is difficult to detect efficiently using SQL. Basically, we are trying to detect a regular expression of $a^+b$ tuples, where $a$ denotes the products and $b$ denotes the case. $a^+$ denotes that the products can occur one or more times. Furthermore, there are timing constraint $t_0$ between the last $a$ tuple and the $b$ tuple, and timing constraint $t_1$ between consecutive $a$ tuples.

In SQL, one join is required for expressing constraint $t_1$ between every pair of consecutive tuples. Since we do not know how many consecutive $a$ tuples there will be, detection of this pattern cannot be expressed using regular join operators. Therefore, SQL lacks the constructs to supports this kind of temporal patterns.

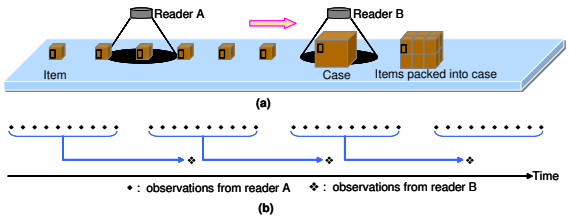


**Figure 1. Use RFID readings to detect containment**

For more examples of different temporal patterns, consider a RFID-enabled clinic laboratory. Suppose a staff member wears a wrist-band RFID reader, and the equipments are labelled with tags. The staff need to perform a sequence of operations on patient's specimen, which is detected as sequences of RFID readings (when the wrist-band RFID reader is put very close to the corresponding equipment). We like to enforce a certain workflow of actions with timing constraints as in the next example, and raise an exception to alert the staff member if the workflow is violated.

**Example 5** *In a lab test, a sequence of operations $A$, $B$ and $C$ has to be performed in the correct order by a medical staff, within some given time period (e.g. 1 hour within the test starts, when $A$ is operated on), otherwise the test may fail. Therefore, if any operation occurs in the wrong order (e.g. $C$ directly follow $A$, or $A$ follow $B$), or if the required time interval passes and the sequence of action did not complete, we raise an exception alert.*

Again, trying to detect this temporal pattern using SQL is difficult. Here we need to perform a 3-way self-join on the stream of tag readings, and apply multiple timestamp-based selection conditions. Furthermore, if we allow repeated operations on one equipment, then we will have to support repeating patterns like $A^+$, which becomes impossible for a join operation.

Temporal pattern detection, as the examples above, is important for many RFID applications. However, since SQL is not well-equipped to handle them, we propose to extend the ESL [2, 8] language, which is a SQL-based general-purpose stream query language, with temporal pattern operators as discussed next.

# 3. The ESL-EV Stream Query Language

A SQL-based stream query language provides convenient constructs for handling many essential RFID data processing tasks. However, complex temporal patterns in RFID

data processing present a major challenge. In this section we first add temporal event operators to the ESL language to define temporal patterns spanning multiple data streams. We then discuss using the sliding-window constructs for temporal operators. The resulting language was named ESL-EV (ESL-EVents).

## 3.1. Temporal Event Operators

We define a temporal event operator as a mapping from a sequence of data tuples (ordered on timestamps) to a boolean value of *true* and *false*. Therefore, a temporal event operator can be used as part of the predicate in a SQL query (i.e., in the `WHERE` clause).

**Primitive Events and Composite Events** In traditional event processing systems there could be many types of primitive events (e.g. events associated with database transactions—INSERT, DELETE, UPDATE, etc). Here, we are only concerned with the arrivals of continuous tag readings under the standard model of append-only relations for data streams. Therefore, tag reading arrivals are the only primitive events we consider.

Composite events are temporal patterns formed by other events (which could be both primitive or other composite events). In [17] it was proved that a core set of event operators can express temporal patterns on primitive events with expressive power equivalent to regular expressions on strings. For example, conjunctions, negations, sequence, and star sequences (sequence with varying number of repeats), which are named as operators *andsign, !, relative, and relative+* were proved to constitute such a core set of operator in [17].

The semantics of the conjunction and negation operators are intuitive. For example, if $A$ and $B$ are primitive events, $A$ *andsign* $B$ is a complex event that indicates both $A$ and $B$ has happened. Negation applied on $A$ means that $A$ did not happen. The conjunction event operator *andsign* can be naturally implemented in SQL using the `AND` operator, while the negation operator can be expressed by the `NOT` operator in SQL. Hence we will focus on the discussion of sequence and star-sequence operators. Moreover, when negation is applied on sequences the condition-checking could become very complex. Therefore we also introduce operators to capture common negation conditions for sequences.

**Tuple Pairing Modes** As studied in the Snoop project [10], arbitrarily applying the event operators on primitive events will result in the generations of large amounts of composite events, many of which are not useful. We introduce *Tuple Pairing Modes* constructs to ESL-EV to limit temporal pattern generation to common interesting cases of RFID data processing. Next, we describe our extensions of temporal operators.

### 3.1.1 The $SEQ$ Operator

This operator detects specific sequences of tuples from multiple streams. The basic operator $SEQ(E_1, E_2)$ returns *true* on two tuples in streams $E_1$ and $E_2$, if the tuple from $E_2$ has a timestamp after the tuple from $E_1$ (the tuples may optionally need to satisfy qualifying conditions on attributes). This basic two-arguments operator can be extended to take multiple streams in the parameter list, $SEQ(E_1, E_2, E_3, \cdots)$, which indicates that a tuple from stream $E_1$ is followed by a tuple from stream $E_2$, which is followed by a tuple from stream $E_3$, and so on[1].

For an example, suppose that in a factory plant every product needs to go through a series of four quality checking steps. Each checking step is has an RFID reader that reads the RFID tag on the product. The four RFID readers produce data streams $\mathbf{C}_1$, $\mathbf{C}_2$, $\mathbf{C}_3$ and $\mathbf{C}_4$, all with the same schema **(readerid, tagid, tagtime)**. We monitor the completion of this sequence by the following continuous query that utilizes the $SEQ$ event operator:

**Example 6** *Detecting a Sequence with the $SEQ$ Operator*

```
SELECT C1.tagid, C1.tagtime,
    C2.tagtime, C3.tagtime, C4.tagtime
FROM C1, C2, C3, C4
WHERE SEQ(C1, C2, C3, C4)
    AND C1.tagid=C2.tagid AND C1.tagid=C3.tagid
    AND C1.tagid=C4.tagid
```

There are two major issues with the above simple approach. i) It seems that we need the full tuple history on the streams to match with incoming tuples. We need constructs such as windows to limit the scope of tuple matching. ii) For any incoming C4 tuples, there might be many matching C1, C2 and C3 combinations, which could be a lot more than what actually makes sense for the application. We will need to remove those undesired combinations, which will may requite complex predicate conditions. We like to avoid generating those wrong combinations to begin with, therefore we define *Tuple Pairing Modes.*

**Sliding Windows on $SEQ$:** In some applications, the sequence might need to finish within a certain time frame. This motivates the application of a sliding time window on the event operator. A sliding-window is a common construct used for join operators and aggregate operators in stream query languages. Here we extend them to the event operator with the following syntax:

```
SEQ(E1, E2, E3, ···, En)
    OVER [time-period PRECEDING En]
```

The sliding window is defined relative to a tuple in the sequence. In the following example the sequence has to

[1] As in standard SQL, the streams in the argument list of the operator may in fact be the same data stream with different aliases.

finish within 30 minutes of the first tuple in the sequence, to qualify as a good sequence.

SELECT **C4.tagid, C1.tagtime,** $\cdots$
FROM **C1, C2, C3, C4**
WHERE **SEQ(C1, C2, C3, C4)**
OVER **[30 MINUTES** PRECEDING **C4]**
AND **C1.tagid=C2.tagid** AND **C1.tagid=C3.tagid**
AND **C1.tagid=C4.tagid**

**Tuple Pairing Modes:** Sliding windows provides a simple method to both maintain less tuple history (the expired tuples can be removed) and reduce the generation of unwanted combinations (thus only tuples within the window will produce candidates). However, there are many situations where advanced rules of tuple pairing are needed to avoid complex predicate conditions. In the previous example, suppose by the application semantics we also know that, for any incoming C4 tuple we are only interested in pairing it with the most recent qualifying tuple on each of the other streams (see the RECENT mode below). This scenario gives us a much better way to optimize tuple history storage and tuple pairing.

Therefore, we define *Tuple Pairing Modes* as event operator modifiers, which dictate how the tuple history is kept on the streams, and how the matching tuples are generated, to support common application situations. Next we discuss four *Tuple Pairing Modes*[2].

- UNRESTRICTED: all possible pairing of tuples generate valid events. This is the default semantics when no modifier is applied, as in Example 6.
- RECENT: An incoming tuple is only matched with the most recent qualifying tuple on the other data streams. This mode allows aggressive purge of tuple history, as earlier tuples are constantly replaced by later tuples as the candidate to match with future incoming tuples.
- CHRONICLE: An incoming tuple is matched with the earliest qualifying tuples on other streams. Furthermore, a tuple can only participate in forming temporal events once. Therefore once a matching occurs and an event is generated, the participating tuples can be removed from the tuple history.
- CONSECUTIVE: This mode is the most useful when we are interested in temporal pattern formed by tuples on a single data stream. To extend the concept of *consecutiveness* to multiple data streams, we define a timestamp-based union of all tuples from different streams as the *joint tuple history* of all the streams. Then under the CONSECUTIVE mode we consider the *joint tuple history* formed by all participating data streams—A temporal pattern is of interest if and only if tuples in the pattern are *consecutive* tuples on this unioned joint history. Under this mode, a tuple can only form a pattern with its immediate adjacent tuples on the *joint tuple history*. Therefore, tuple history can be safely purged each time a sequence is finished or interrupted.

[2]The first three modes are based on *Event Consumption Modes* specified in Snoop[10].

The *Tuple Pairing Modes* are used to directly modify the temporal event operators as in the following example. UNRESTRICTED is assumed when the MODE clause is omitted.

**SEQ(**$\mathbf{E}_1$**,** $\mathbf{E}_2$**,** $\mathbf{E}_3$**,** $\cdots$**,** $\mathbf{E}_n$**)**
OVER **[WINDOW_SPEC]**
MODE **CONSECUTIVE**

Next we use an example to illustrate how the above four modes work. We use a scenario of Example 6. Suppose the *joint tuple history* of the four participating data streams is the following (the tuples are presented as a sequence, in the format of [arrival time: stream source]):

**[ t1:C1, t2:C1, t3:C2, t4:C3, t5:C3, t6:C2, t7:C4** $\cdots$**]**

Basically, the above says that a $C1$ tuple arrives first at t1, and another C1 tuple arrives at t2, and so on. By time t7, a C4 tuple arrives and causes the temporal operator $SEQ(C1, C2, C3, C4)$ to return.

1. Under the UNRESTRICTED mode, the operator will return *true* on all possible sequences of the correct time order, they include

   (t1:C1, t3:C2, t4:C3, t7:C4)

   (t1:C1, t3:C2, t5:C3, t7:C4)

   (t2:C1, t3:C2, t4:C3, t7:C4)

   (t2:C1, t3:C2, t5:C3, t7:C4)

   If we only want to select a subset of these possible sequences, we will use extra join conditions to remove unwanted sequences.

2. Under the RECENT mode, the operator will only use the most recent qualifying tuple of each stream to return *true*. For example, relative to the C4:t7 the most recent qualifying C3 tuple is C3:t5. The one at time t4 is not used as it is not the most recent qualifying tuple. Then, the most recent qualifying C2 tuple is C2:t3. The C2:t6 tuple is not a qualifying tuple as it is after the C3:t5, thus does not form a correct sequence. Finally we should pick C1:t2 instead of C1:t1. Therefore we only return one event:

   (t2:C1, t3:C2, t5:C3, t7:C4)

3. Under the CHRONICLE mode, the operator uses the earliest available tuple of each stream. It returns *true* only on the following:

(t1:C1, t3:C2, t4:C3, t7:C4)

After the successful matching, all 4 tuples will not be used for future operator matching again.

4. Under the CONSECUTIVE mode, the operator looks for consecutive tuples on the *joint tuple history* of the 4 streams to form the correct sequence. It will not return *true* for any sequence in this case.

### 3.1.2 The Star Sequence

The $SEQ$ operator above is equivalent to composing a 4-way join query on the streams and explicitly applying the timing conditions[3]. However, as discussed in Example 4 of Section 2, join operations can not handle repeating tuples. Thus we add star sequence to the SEQ operator—the '*' symbol can be applied on any of the operator arguments to indicate that this event can repeat many times. The sliding window constructs and the *Tuple Pairing Modes* can also be applied on the star sequences.

Take as example $SEQ(E_1^*, E_2)$, it returns *true* when a tuple from stream $E_2$ follows some tuples from $E_1$. $SEQ(A^*, B, C^*, D)$ says that the operator returns *true* if some A tuples are followed by exactly one B tuple, and followed by some C tuples, and finally followed by one D tuple.

In our semantics, we only generate event on the longest possible star sequences. For example, in $SEQ(E_1^*, E_2)$ if there are three $E_1$ tuples followed by $E_2$, then we will only generate event for ($E_1$, $E_1$, $E_1$, $E_2$). We will not generate event for the combinations where there are less than three $E_1$ tuples, although they still match the pattern. The special case is the last event in the sequence, we return event in an online fashion for each tuple arrival and do not wait for the *longest* sequence (as there might be no valid indicator to tell us to stop matching). E.g., in $SEQ(E_1^*, E_2^*)$, if there are three $E_2$ tuples coming in after the $E_1$ tuples, we generate one event for each $E_2$ tuple.

As an example for the star sequence, the containment relationship in Example 4 can be represented as $SEQ(R1^*, R2)$. Here R1 is a data stream generated by product tag readings, and R2 is a data stream generated by case tag readings. The CHRONICLE *Tuple Pairing Mode* is appropriate here, since a group of products matching $R1^*$ will be packed into only one case matching $R2$ and will not be packed again. Therefore, in ESL-EV, we detect this containment relationship with the following continuous query, which will returns the case tagid together with the number of items packed into the case and the timing information.

**Example 7** *The SEQ Operator with Star Sequence for Containment*

```
SELECT FIRST(R1*).tagtime, COUNT(R1*),
       R2.tagid, R2.tagtime
FROM R1, R2
WHERE SEQ(R1*, R2) MODE CHRONICLE
    AND R2.tagtime - LAST(R1*).tagtime ≤ 5 SECONDS
    AND R1.tagtime - R1.previous.tagtime ≤ 1 SECONDS
```

There are some special properties about the star sequence:

1. A few special aggregate functions are defined for the star sequences. The FIRST and LAST functions used in Example 7 are aggregate functions that return the first tuple and the last tuple, respectively, in the (repeated) star sequence. The COUNT function returns the number of tuples in the star sequence.
2. We use the *previous* operator to indicate the tuple preceding the current tuple in the star sequence. In Example 7 we used this syntax to define inter-arrival timing constraints on the star sequence of products.
3. If we only return aggregated information for the star sequence, as in Example 7, only one tuple is returned each time the SEQ operator is evaluated to *true*. However, if we need to know the identities of individual tuples in the star sequence. (I.e. We like to retrieve the RFID tag readings of all the products that are packed into one particular case.) This situation requires more than one tuple to be returned for each positive evaluation of SEQ.

   For Example 7, suppose now we like to return each individual R1 reading that participates in the star sequence event. Then, if K tuples are included in the star sequence R1*, we will have K returned tuples, as in the following query[4].

   ```
   SELECT R1.tagid, R1.tagtime,
          R2.tagid, R2.tagtime
   FROM R1, R2
   WHERE SEQ(R1*, R2) MODE CHRONICLE
   AND R2.tagtime - LAST(R1*).tagtime ≤ 5 SECONDS
   AND R1.tagtime - R1.previous.tagtime ≤ 1 SECONDS
   ```

### 3.1.3 The $EXCEPTION\_SEQ$ Operator

Many applications require that we detect exceptions on sequences. Any violation of the prescribed sequence or the

[3] In fact, the tuple sequences that satisfy $SEQ(C1, C2, C3, C4)$ can be precisely found this way—For each incoming C4 tuple, we join it with all the tuples that have arrived so far in the other 3 streams, apply the join conditions and the timing conditions, and those tuples that satisfy all conditions render the operator to be *true* and thus are returned as the query result. Therefore, $SEQ(C1, C2, C3, C4)$ could be seen as a simplified syntax for the join operation in this case.

[4] Such multiple-return is allowed when there is only one star sequence in the argument. It is not allowed when there are multiple star sequences, as there will be too many combinations.

timing constraints on the sequence will raise an alert. For instance, in Example 5 of Section 2, a medical staff needs to perform a lab test, which consists of a fixed sequence of operations on multiple instruments, and the test has to finish within 1 hour once started.

Suppose tag readings indicating the operations $A$, $B$ and $C$ come in as three different data streams. Now consider the *joint tuple history* formed by the union of the three data streams. Normal operations will lead to tuple history as the following:

$(A, B, C, A, B, C, A, B, C, \cdots)$

The correct sequence corresponds to $SEQ(A, B, C)$ under the CONSECUTIVE mode with a sliding window of 1 hour. To alert the person when something goes wrong, we try to detect all possible violations of this sequence. The exception conditions are i) if next incoming tuple does not match the correct event for the sequence (e.g., we have $A$ but the next incoming tuple is $C$), and ii) if a sequence does not start from the correct initial event (e.g., the first event in our sequence is $B$), or iii) if a sequence is started but not finished when the sliding window expires.

To specify these exception conditions, we introduce the $EXCEPTION\_SEQ$ operator, which is a generalization of $SEQ$, as discussed below. The semantics of this is based on the concept of *Sequence Completion Levels*.

**Sequence Completion Level.** Suppose we have the following sequence that we try to detect exceptions for:

$$SEQ(E_1, E_2, E_3, \cdots, E_n)$$

We define a *partial sequence* of this sequence as a run of tuples $(E_1, E_2, E_3, \cdots, E_k)$ for some $0 < k < n$.

Then, we define the **Sequence Completion Level** of the full sequence $SEQ(E_1, E_2, E_3, \cdots, E_n)$ as equal to $n$. And, if currently there is a partial sequence $(E_1, E_2, E_3, \cdots, E_k)$ for some $0 < k < n$, and it is no longer possible to extend the partial sequence to $(E_1, E_2, E_3, \cdots, E_{k+1})$, then we say that the Sequence Completion Level of $(E_1, E_2, E_3, \cdots, E_k)$ is $k$, and an exception event occurs at level $k+1$. (Note that sequences that do not start with the correct first event has Sequence Completion Level 0.)

Note that the following scenarios can make a partial sequence unable to extend.

1. An existing partial sequence can not longer correctly extend due to an wrong incoming tuple.

   For example, suppose we want $SEQ(A, B, C)$ under the RECENT mode, and currently we have $(A, B)$. An arrival of $B$ will make it impossible for the partial sequence $(A, B)$ to extend (as the second $B$ will replace the first one to match with future $C$ tuples). Therefore an exception event occurs.

2. If an incoming tuple is not the correct event to start a new sequence and can not be matched with existing events to form a correct partial sequence, then we raise an exception on the incoming tuple (i.e., failure to extend a sequence with Sequence Completion Level 0).

   For example, suppose the desire sequence is $SEQ(A, B, C)$ under CONSECUTIVE mode. If we currently have an $(A, B, C)$ and the next tuple is $C$, the incoming tuple can not start a new sequence, an exception event occurs.

3. The expiration of a sliding window on some tuple in a partial sequence will trigger an exception, as we can no longer extend the partial sequence and still satisfy the timing constraint.

Therefore, we define the $EXCEPTION\_SEQ$ operator as the following: The operator $EXCEPTION\_SEQ(E_1, E_2, E_3, \cdot\ \cdot\ \cdot, E_n)$ returns true whenever we have a sequence with Sequence Completion Level less than $n$.

The query below can now be used to solve Example 5, where an exception will be raised whenever the operation sequence of the medical staff violates the correct procedure, or the sequence does not finish within the desired time (1 hour).

```
SELECT A1.tagid, A2.tagid, A3.tagid
    FROM A1, A2, A3
    WHERE EXCEPTION_SEQ(A1, A2, A3)
      OVER [1 HOURS FOLLOWING A1];
```

Alternatively, we may explicitly use Sequence Completion Level in the predicate, by defining an operator $CLEVEL\_SEQ$, which returns the Sequence Completion Level of a tuple sequence. Therefore, the following query using the $CLEVEL\_SEQ$ operator is equivalent to the last query.

```
SELECT A1.tagid, A2.tagid, A3.tagid
    FROM A1, A2, A3
    WHERE (CLEVEL_SEQ(A1, A2, A3)
        OVER [1 HOURS FOLLOWING A1]) < 3;
```

While it is common for stream query language to support PRECEDING windows, it is not common to support FOLLOWING windows, as they are usually specified between two streams and are symmetric. (E.g., **A1** PRECEDING **A2** is the same as **A2** FOLLOWING **A1**.)

Here we use the FOLLOWING window construct to allow the sliding window to start from any of the participating events when the number of arguments in the operator is larger than 2. E.g., we may need to say that the sliding window should start from the second event in the sequence, which can not be specified using an equivalent PRECEDING construct.

```
EXCEPTION_SEQ(A1, A2, A3)
    OVER [1 HOURS FOLLOWING A2]);
```

In a DSMS the $EXCEPTION\_SEQ$ operator may require *Active Expiration* semantics, where window expiration has to be detected without any new tuple arrivals (readers are referred to [8] for more details about this). Similar to the $SEQ$ operator, $EXCEPTION\_SEQ$ can also allow repeating star sequences does. Detailed discussion of this is omitted here.

### 3.2. Extending Sliding Windows

Some sliding window constructs, which are not commonly supported in a stream query language, are needed for RFID applications.

Consider the situation where RFID readers at the door detect products and personnel passing through. Say that only authorized personnel may carry products out of the door, then we need to generate an alert when unauthorized people take out any product, which can be done as follows: If a product is detected at the door and there is no authorized personnel detected within time $\tau$ (e.g. 1 minute) before or after the item exit, raise an alert of a potential theft (e.g., suppose the products could be on carts, which could pushed or pulled by a person).

Suppose we have the following schema of data stream, where both personnel and products are detected by the same RFID reader, but with different 'tagtype' values.

```
tag_readings(tagid, tagtype, tagtime);
```

Under this scenario, the sliding window has to be defined both *before* and *after* the occurrence of a product exit, as in the continuous query in Example 8.

**Example 8** *Sliding Window Across Sub-query Boundary*

```
SELECT person.tagid
FROM tag_readings AS person
WHERE person.tagtype = 'person' AND NOT EXISTS
    (SELECT * FROM tag_readings AS item
        OVER [1 MINUTES
            PRECEDING AND FOLLOWING person]
        WHERE item.tagtype = 'item' )
```

Here we need to specify a sliding windows across the boundary of a correlated sub-statement. I.e., the *item* tuples inside the sub-statement are need to be inside sliding windows defined on a *person* tuple, which is outside the sub-statement. To the best of our knowledge, specifying a sliding window with synchronization across a sub-query boundary is not supported in current SQL-based stream query languages. It represents an extension that is needed for RFID applications. Also, in this example the FOLLOWING window construct is again needed.

**Summary** In summary, we developed the following extensions to better facilitate the detection of temporal events in RFID data processing:

- The $SEQ$ operator for detecting sequence of tuples;
- Star sequences in $SEQ$ for repeating events;
- The $EXCEPTION\_SEQ$ operator for detecting exception conditions on *consecutive* sequences and sliding windows, and
- Applying sliding window constructs on the temporal pattern operators and introducing *Tuple Pairing Modes* as event operator modifiers.
- Extending sliding window constructs to include windows synchronized across sub-query boundary, and the *FOLLOWING* window construct.

These language extensions make it possible to conveniently handle temporal pattern detection in a SQL-based syntax. The many example scenarios used throughout this paper illustrate the power of this language to support RFID applications.

## 4. Related Work

RFID data processing has been topics of recent research. In [18] warehousing RFID data was discussed, [15] discusses system architecture and general issues in processing RFID data. In [22], temporal modeling of RFID data was studied, and in [23] a declarative rule-based system was proposed to detect high-level RFID events. In recent work by Wu et al. [25], a stream-based RFID event processing system is discussed which supports a dataflow paradigm with native sequence operators. None of these works discuss RFID data processing in the context of a SQL-based stream query language, which enables us to perform varieties of RFID data processing tasks within a single DSMS system. There are also many research projects and commercial products that focus on building RFID middle-ware to collect data from readers and possibly detect simple events (such as duplicate readings). They generally can not detect complex temporal events in the RFID data.

Data stream querying and DSMS has been the focus of many research in recent years. Many systems have been designed to query streaming data using an SQL-based language. For example the STREAM system [20], the Telegraph system [14] and the Gigascope system [12]. Other systems which choose varieties of user interfaces[5, 11]. However, these systems do not deal with temporal events.

ECA rule systems and active databases systems have been studied for many years. The readers are referred to existing books and reviews such as [24, 21]. The existing ECA-rule systems are usually optimized for transactional data and application[9, 16], and can not support RFID data processing tasks efficiently.

## 5. Conclusions

A DSMS with a SQL-based query language can support a variety of critical data processing tasks for RFID applications. However, complex temporal events detection, which is critical for RFID applications, can not be well supported by current DSMS. In this paper, we show that by extending a SQL-based stream query language with temporal operators and related constructs, a DSMS can support general RFID data processing for a large variety of RFID applications.

## References


[1] The ALE standard. http://www.epcglobalinc.org.

[2] The ESL language manual. http://wis.cs.ucla.edu/stream-mill/doc/esl-manual.pdf.

[3] Streambase home. http://www.streambase.com/.

[4] EPC Tag Data Standards Version 1.1. Technical report, EPCGlobal Inc, April 2004.

[5] D. Abadi and et al. Aurora: A new model and architecture for data stream management. 12(2):120–139, 2003.

[6] B. Babcock and et al. Models and issues in data stream systems. 2002.

[7] Yijian Bai, Hetal Thakkar, Chang Luo, Haixun Wang, and Carlo Zaniolo. A data stream language and system designed for power and extensibility. In *CIKM*, 2006.

[8] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim. Composite Events for Active Databases: Semantics, Contexts and Detection. In *VLDB*, pages 606–617, 1994.

[9] S. Chakravarthy and D. Mishra. Snoop: an expressive event specification language for active databases. *Data Knowl. Eng.*, 14(1):1–26, 1994.

[10] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for internet databases. pages 379–390, May 2000.

[11] C. Cranor and et al. Gigascope: High performance network monitoring with an sql interface. page 623. ACM Press, 2002.

[12] D. Abadi et al. Aurora: A new model and architecture for data stream management. *VLDB Journal*, 12(2):120–139, 2003.

[13] Sirish Chandrasekaran et al. Telegraphcq: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

[14] M. J. Franklin and etc S. R. Jeffery. Design Considerations for High Fan-In Systems: The HiFi Approach. In *CIDR*, pages 290–304, 2005.

[15] S. Gatziu and et al. Detecting Composite Events in Active Databases Using Petri Nets. In *Workshop on Research Issues in Data Engineering: Active Database Systems*, 1994.

[16] N. H. Gehani, H. V. Jagadish, and O. Shmueli. Composite Event Specification in Active Databases: Model & Implementation. In *VLDB*, 1992.

[17] Hector Gonzalez, Jiawei Han, Xiaolei Li, and Diego Klabjan. Warehousing and Analyzing Massive RFID Data Sets. In *ICDE*, 2006.

[18] Yan-Nei Law, Haixun Wang, and Carlo Zaniolo. Data models and query language for data streams. In *VLDB*, pages 492–503, 2004.

[19] R. Motwani and et al. Query processing, approximation, and resource management in a data stream management system. In *First CIDR 2003 Conference*, Asilomar, CA, 2003.

[20] Norman W. Paton and Oscar Díaz. Active database systems. *ACM Comput. Surv.*, 31(1):63–103, 1999.

[21] F. Wang and P. Liu. Temporal Management of RFID Data. In *VLDB*, 2005.

[22] Fusheng Wang, Shaorong Liu, Peiya Liu, and Yijian Bai. Bridging physical and virtual worlds: Complex event processing for rfid data streams. In *EDBT*, 2006.

[23] Jennifer Widom and Stefano Ceri. *Active Database Systems: Triggers and Rules For Advanced Database Processing.* Morgan Kaufmann, 1996.

[24] E. Wu, Y. Diao, and S. Rizvi. High-Performance Complex Event Processing over Streams. In *SIGMOD*, 2006.