THE UNIVERSITY OF QUEENSLAND
AUSTRALIA

Time To Live: Temporal Management of Large-Scale RFID Applications

Xue Li
Jing Liu
Quan Z. Sheng
Weicai Zhong

October 2008

Technical Report SSE-2008-02

Division of Systems and Software Engineering Research
School of Information Technology and Electrical Engineering
The University of Queensland
QLD, 4072, Australia

http://www.itee.uq.edu.au/~sse

Time To Live: Temporal Management of Large-Scale RFID Applications

Xue Li [#], Jing Liu [*1], Quan Z. Sheng [$], Weicai Zhong [*]

[#]*School of Information Technology & Electrical Engineering, The University of Queensland, Australia*
[*]*Institute of Intelligent Information Processing, Xidian University, China*
[$]*School of Computer Science, The University of Adelaide, Australia*
[1]xueli@itee.uq.edu.au

Abstract— **In coming years, there will be billions of RFID tags living in the world tagging almost everything for tracking and identification purposes. This phenomenon will impose a new challenge not only to the network capacity but also to the scalability of event processing of RFID applications. Since most RFID applications are time sensitive, we propose a notion of *Time To Live* (TTL), representing the period of time that an RFID event can legally live in an RFID data management system, to manage various temporal event patterns. TTL is critical in the "Internet of Things" for handling a tremendous amount of partial event-tracking results. Also, TTL can be used to provide prompt responses to time-critical events so that the RFID data streams can be handled timely. We divide TTL into four categories according to the general event-handling patterns. Moreover, to extract event sequence from an unordered event stream correctly and handle TTL constrained event sequence effectively, we design a new data structure, namely Double Level Sequence Instance List (DLSIList), to record intermediate stages of event sequences. On the basis of this, an RFID data management system, namely Temporal Management System over RFID data streams (TMS-RFID), has been developed. This system can be constructed as a stand-alone middleware component to manage temporal event patterns. We demonstrate the effectiveness of TMS-RFID on extracting complex temporal event patterns through a detailed performance study using a range of high-speed data streams and various queries. The results show that TMS-RFID has a very high throughout, namely 170,000 - 870,000 events per second for different highly complex continuous queries. Moreover, the experiments also show that the main structure to record the intermediate stages in TMS-RFID does not increase exponentially with the number of events. These illustrate that TMS-RFID not only has a high processing speed, but also has a good scalability.**

I. Introduction

The size and characteristics of RFID (Radio Frequency Identification) data pose many new challenges to the current data management systems. RFID events have their own characteristics that cannot be supported by traditional event systems [1], [2], [3]. RFID Application Level Event (ALE) standard proposed by EPCglobal[1], a common interface to process raw RFID events, also emphasizes the importance of RFID data stream processing. Since RFID data are time-dependent, dynamically changing, in large volumes, and carrying implicit semantics, a general RFID data processing framework is required to process high volume RFID data streams in real time, and automatically transform the physical RFID observations into the virtual counterparts linked to business applications.

Among various RFID applications, simple and complex event detection plays an important role. Several event processing systems that execute complex event queries over real-time streams of RFID readings have been proposed in recent years [2], [3], [4], [5]. The complex event queries in these systems can filter and correlate events to match with specific patterns, and transform the relevant events into high-level business events for the use of external applications.

However, most RFID applications have time restrictions on target events. An event is valid only if it happens within or after a time limit. For example, in general cases, it is required in an airport that a baggage must arrive at a specified place to wait for being loaded onto a flight 30 minutes before this flight is scheduled to take off. Unfortunately, this kind of time-restriction issues are largely ignored by most of the current event processing systems and left as separate or individual application problems. SASE [3] can extract a target event sequence according to the prescribed sequence order in a query, or check whether a target event finishes within a prescribed period of time. But SASE cannot restrict the temporal distance between any two successive events. Although Wang *et al.* [2] proposed several temporal operators and related constructs that can restrict the temporal distance between two events, these operators can only detect a sequence with two events or the same type of events.

We perceive that it is useful to develop an novel RFID data management system that can handle various temporal event patterns, both simple and complex, with time restrictions. Since temporal event patterns restrict that events can only be valid within a certain period of time, we identify a new time notion, namely *Time To Live* (TTL), to cover all kinds of complex temporal event patterns, including those can be processed in existing systems.

In the Internet, TTL is an important concept that helps the Internet to discard the datagrams that may not be able to reach their destinations. RFID is widely believed as a promising technology to create an "*Internet of Things*" in the near future. As a result, we suggest that TTL in RFID applications should become an important notion in the "Internet of Things", to help the "Internet of Things" to be free from a large amount

[1]http://www.epcglobalinc.org

of partial event-tracking results and make prompt responses to time-critical events.

All available RFID systems have a time notion, i.e., timestamps, which are assigned to RFID data by RFID readers when tags are read. Timestamps indicate the time points that events occurred in the real world, while TTL indicates the time restrictions that target events should satisfy. Thus, in addition to timestamps, we advocate that TTL should become another important time notion for the RFID data management.

When processing complex events, it is general that the order of arriving events may not match with the order of the occurrence of the events in the real world. The problem, named as *Unordered Event Stream* (UnES), can be caused by network routing delay or by an arbitrary selection of records when a single tag is read by multiple readers simultaneously. An analogue problem can be found in the Internet communications when a datagram arrives at the system application level being out of sequence. Existing works address the UnES problem in two ways: one is to simply ignore the problem by assuming that the events enter into the system with the same order of their occurrences in the real world, such as SASE [3]. The other is to sort events according to their timestamps before extracting complex events patterns from them, such as Cayuga [6].

Both approaches, however, have disadvantages. For the first approach, since RFID readers distribute widely and each one may have a delay, it is obvious that events cannot always enter into the system in the order of their occurrences. Moreover, RFID tags can be read by different readers simultaneously, but the readers may send these primitive events to the system in different times. Also, even after the events enter into the system, they may still be subject to the queuing of different processes. Clearly, ignoring the UnES problem is impractical to most RFID applications. For the second approach, they requires the time point that an event enters into the system must be no later than a bound, cope with all applications by the same bound. If there are many applications simultaneously handled by the system, this single bound cannot be sufficient to deal with the combined effect of all incoming data.

To solve the UnES problem, we have developed a data structure, *Double Level Sequence Instance List* (DLSIList), to record the unordered event streams so that the system can extract sequence patterns from an unordered event stream. To avoid the size of the intermediate stages growing exponentially, an algorithm has also been developed to maintain and update DLSIList by using TTL.

Our contributions in this paper can be summarized as follows:

- **A novel notion of TTL**: TTL is a novel concept used to enforce the time restrictions in RFID applications. The main advantage of TTL is that it allows the construction of a *generic* mechanism to handle all kinds of time restrictions. In addition, we design and illustrate a variety of TTL queries that may appear in different RFID applications.
- **An effective solution to the UnES problem based on TTL**: A new data structure, namely DLSIList, is designed based on TTL to mitigate the exponential growth of partial-events tracking results, as well as to extract the complex events effectively.
- **A prototype system for handling TTL queries**: We have developed a prototype system, Temporal Management System over RFID data streams (TMS-RFID) that implements the ideas proposed in this paper. TMS-RFID has been developed as a stand-alone component that is adaptable for different event processing systems. We also demonstrate the effectiveness and efficiency of TMS-RFID in handling large-scale of high-speed RFID data streams.

The remainder of this paper is organized as follows. Section 2 overviews related work. TTL is introduced in Section 3. TTL query language is presented in Section 4. The algorithm based on TTL for solving the unordered event stream problem is described in Section 5. The system architecture of TMS-RFID is presented in Sections 6. The results of a detailed performance study are reported in Section 7. Section 8 concludes this paper.

II. RELATED WORK

Although RFID technology has existed for more than 50 years, it poses many new challenges for data processing and management in recent years when RFID tags are applied in large-scale applications, such as missed and unreliable RFID readings, redundant RFID data, in-flood of RFID data, spatial and temporal management of RFID data. Chawathe *et al.* [1] present a brief introduction to RFID technology and highlight several data management challenges. The importance of event processing for RFID data is pointed out by Palmer [7].

Wu *et al.* [3] propose a stream-based RFID event processing system, namely SASE. SASE can efficiently execute monitoring queries over RFID data streams, and is the first proposed model for RFID event processing. However, SASE cannot control time intervals between two successive events. Since many applications require two successive primitive events in a target complex event satisfying some time intervals, the ability of controlling such intervals is extremely important. Moreover, SASE assumes that all events are totally ordered by their timestamps. Such an assumption is too restrictive and would not be true for most RFID applications.

Wang and Bai *et al.* [2], [4], [5] address the problem of complex event processing over RFID data stream from the viewpoint of ER model and SQL-based stream query language. Wang *et al.* [5] propose an expressive temporal-based data modeling of RFID data, using a Dynamic Relationship ER Model, and the method on how to use rules to transform RFID data from observations into the data model. Wang *et al.* [2] further formulate a declarative rule based approach to provide support of automatic RFID data transformation between the physical world and the virtual world. Such approach is capable of detecting complex temporal-pattern-based high level events. Among the four proposed temporal complex event constructors, SEQ and TSEQ can only detect sequences of two events, while TSEQ and TSEQ$^+$ can only detect sequences of the same type of events. Although TSEQ and TSEQ$^+$ can control time intervals between two events, TSEQ is only for two events and TSEQ$^+$ is only for the same type of events. Bai *et al.* [4] extend a SQL-based stream query language with temporal operators and related constructors and use several example scenarios to illustrate the power of the proposed language.

Rizvi *et al.* [8] discuss the system architecture and general issues in processing RFID data and propose a system, namely, HiFi. HiFi aggregates events along a tree-structured network on various temporal and geographic scales.

Mansouri-Samani and Liu *et al.* [9], [10] also consider temporal restrictions. However, those temporal restrictions cannot be used to support the special RFID events such as temporal sequence and temporal negation.

Finally, Demers and Brenna *et al.* [6], [11] propose a system, Cayuga, to process complex events in publish/subscribe systems. Cayuga offers a powerful expressive language similar to the one required in RFID data management systems. Cayuga can deal with a wide range of complex event patterns, including sequence and Kleene closure patterns. However, Cayuga cannot control time intervals between two successive events. For the UnES problem, Cayuga sorts events according to their timestamps before extracting complex events patterns from them, using the technique proposed by Srivastava *et al.* [12].

III. TIME TO LIVE

Most RFID applications have time restrictions on target events. However, many partial results would not be able to reach their final stages because of the time restrictions. We therefore need a mechanism to time out RFID events, either simple or complex, to reduce the size of partial results, and thereby to achieve a high scalability in dealing with infinite RFID data streams.

A. RFID Event Model

RFID data are presented to system in terms of *events*. Some events may only need single readings, while the others may need multiple readings over a period of time at different locations [2], [3], [4]. The RFID event model serves as a basis for TTL.

Definition 1 (*Event, Event Type*). An event is an occurrence of interest happening in a particular time and location in the real world and is recorded in RFID data management systems. An event type is a template that prescribes a class of events that consists of a set of attributes and the values of these attributes. Each event has an event type and a set of values corresponding to the attributes of this event type. △

Events can be divided into different types according to applications. We adopt the definitions of events from EPC Information Services (EPCIS) standard[2]. That is, each core event type has the fields that represent four key dimensions of any EPCIS event: (i) the object(s) or other entities that are the subject of the event; (ii) the date and time; (iii) the location at which the event occurred; and (iv) the business context. These four dimensions usually represent "what, when, where, and why", respectively. Here, we encode the information of all four dimensions into the attributes of an event.

[2]http://www.epcglobalinc.org.

Definition 2 (*Primitive Event*). A primitive event is an RFID reader observation. In particular, the time attribute of each primitive event is a *timestamp*, representing the time point that an RFID reader reads an RFID tag. △

Definition 3 (*Complex Event*). A complex event is a pattern of primitive events and happens over a period of time. △

A complex event usually is composed of a few primitive events, and has a start time and an end time.

B. TTL Taxonomy

Definition 4 (*Time To Live*). Time To Live, denoted as TTL, is the period of time that an event can legally live in an RFID data management system. △

In RFID applications, both primitive and complex events are detected by RFID readers in many different situations:

- Single event: A primitive event is detected by a single RFID reading.
- Event sequence: A complex event that involves the same or different event type is read in a certain order within a time period by different readers.
- Repeating events: RFID tags can be detected at different locations and time points with a fixed number of times.
- Periodical events: A number of successive RFID readings, or the period of times that a sequence of events can be detected.

After carefully analyzing various RFID applications, we divide TTL into four categories.

Category 1: Absolute TTL

Definition 5 (*Absolute TTL*). Absolute TTL, denoted as TTL$_a$, is the period of time that a primitive event can enter into RFID data management systems. △

TTL$_a$ specifies the period of time that an RFID tag can live in the physical world. After that period, even if the primitive events generated by that tag are sent to the system again, the system would not recognize them any more or would raise alarms to advice user that the tag is invalid but still used in the physical world.

EXAMPLE 3.1: Suppose each one-time only train ticket is attached with an RFID tag. This kind of RFID tag can only be valid within the period of time that a passenger travels from place A to B, and cannot be reused. After the passenger arrives B, the primitive events corresponding this tag will be labeled as invalid in the system.

EXAMPLE 3.2: In pharmacy, drugs have expiry dates. Suppose bottles of medicines are attached with RFID tags. This kind of RFID tag can also only be valid within the expiry dates, and be used only once. After the expiry date, if the medicine is still on the shelf, the system should raise an alarm.

Although both at train stations and in pharmacy stores, there are many RFID readers, and one RFID tag can generate many primitive events with different timestamps, there is a unique period of time related to each RFID tag. This period of time is TTL_a. In other words, TTL_a of the primitive events in Example 3.1 is the period of time that the train moves from A to B, while that of Example 3.2 is the expiry date. Obviously, TTL_a indicates the *life span* of the tagged objects in the physical world.

Since the number of tags in the physical world will increase dramatically, if a system just tries to process all tags it is reading, it would be worn down quickly. The main function of TTL_a is in fact to help the system to distinguish between the tags that are still *in use* and the tags that *cannot be used* any more. As a result, the workload can be reduced in later processing steps.

One may argue that we can kill the tags if they cannot be used any more, and there is in fact a "kill" feature in the EPCglobal Architecture Framework[3]. However, the "kill" feature in EPCglobal Architecture Framework is only a part of a comprehensive privacy policy, not aimed at reducing the workload of systems. In addition, what would happen if someone physically reproduces the tags with the same EPC values of the tags that have been killed? Since these fake tags have the same EPC values of the killed ones, the system would be easily *cheated*. We therefore strongly advocate that it is important to ensure that dead tags cannot be recreated at a system logic level. TTL_a is designed for meeting this purpose. TTL_a can be assigned when tags are physically generated and recorded in the system when the tags are first read. In some cases, TTL_a can also be opened until users notify the system.

Category 2: Relative TTL

Definition 6 (*Relative TTL*). For primitive events, Relative TTL, denoted as TTL_r, is the period of time that a primitive event is valid in *one* application. For complex events, TTL_r is the period of time that a complex event can last. △

For primitive events, TTL_r specifies the period of time that RFID tags can stay in a particular application. In Examples 3.1 and 3.2, the tags can only be used once. There are still many situations where each tag can be used in many applications. Each application may generate different types of events, and each type of events may have different time restrictions. TTL_r is used to denote the period of time for each application that the tags are involved in. After that period, they can be reassigned to other applications.

EXAMPLE 3.3: In building access control, suppose each visitor is assigned an RFID tagged card when he comes. Visitors can only stay in the building for a prescribed period of time, $\mathcal{T}$, based on their purposes. When visitors leave the building, cards are taken back, and will be reassigned to other visitors in the future.

In this example, an RFID tag can be used many times, and each time its valid usage time (i.e., TTL_r) is different. Clearly, TTL_r in this example is the period of time that a visitor can stay in the building, which helps systems to find out over-staying visitors.

For complex events, TTL_r specifies the time period that this complex event can occupy. That is, TTL_r is the time restriction on the start and end time of this complex event.

[3]http://www.epcglobalinc.org/standards/architecture/.

Since TTL_a stands for the life span in the physical world of the RFID tag, each primitive event can only be associated with one TTL_a. In contrast, each type of event can have several different TTL_r depending on the applications. An event can have both the restriction of TTL_a and TTL_r simultaneously in some applications. Such tags can be assigned to more than one application during their life span, and each application may have a different TTL_r.

Category 3: Periodical TTL

Definition 7 (*Periodical TTL*). Periodical TTL, denoted as TTL_p, is the time restrictions on the interval between two successive primitive events in a periodically occurred events sequence. The primitive events in the sequence have the same event type. △

In some applications, systems need to check event sequences. These events belong to either the same event type or different event types. The intervals between any two successive events can be identical, or can be different. TTL_p is designed for the cases that the events belong to the same event type and the intervals between any two successive events are identical. In other words, TTL_p controls the period that the same type of events occurs.

EXAMPLE 3.4: Suppose a kind of machine, e.g. an airplane, can be used for many years, but some parts have a shorter life span, and must be replaced in a regular basis. If an RFID tag is attached to each of such parts, the system should determine when a part needs to be replaced according to the readings of RFID readers.

In this example, each time a reader reads an RFID tag attached to such parts, an event will be sent to the system. However, since these parts only need to be replaced periodically, the system does not need to process the events generated within that period. In this case, TTL_p is the interval between two replacements of the same part. Usually, TTL_p requires that the events in the sequence belong to the same event type. The events in the sequence can be primitive or complex.

It should be noted that the applications that TTL_a and TTL_r are applied are different from those of TTL_p. The former only involves a single event, while the latter involves an event sequence. In fact, Example 3.4 is a kind of regular events detection. Such an application involves more than one occurrences of the same type of events, and any two consecutive occurrences of the events must be within some time interval.

Category 4: Sequential TTL

Definition 8 (*Sequential TTL*). Sequential TTL, denoted as TTL_s, is the time restrictions on the intervals between any two successive primitive events in an event sequence. △

Similar to TTL_p, TTL_s is designed for an event sequence, but has no limitation on the event types and the intervals between two successive events. In essence, TTL_p can be viewed as a special case of TTL_s, since TTL_s has fewer restrictions on the event types and the intervals. It can have or have not restrictions on the intervals.

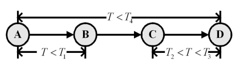


Fig. 1. Time restrictions in Example 3.5

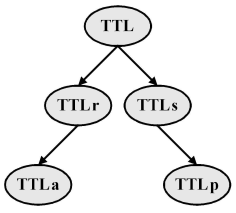


Fig. 2. Internal relationship among different categories of TTL

EXAMPLE 3.5: Suppose many students do an experiment in a laboratory. There are four steps, namely, A, B, C, and D, to finish this experiment, and these four steps must be performed in the correct order and satisfy the time restriction in Figure 1. That is, the interval between A and B must be smaller than a time interval, $\mathcal{T}_1$, the interval between C and D must be larger than a time interval, $\mathcal{T}_2$, and smaller than $\mathcal{T}_3$ ($\mathcal{T}_2 < \mathcal{T}_3$), there is no limitation on the interval between B and C, and the whole experiment must be finished within a given time period, $\mathcal{T}_4$. It is difficult for a teacher to watch whether each student do the experiment in correct order and intervals. If each student takes an RFID reader on his wrist, and the reagents or equipments in each step are attached with RFID tags, then the sequence of steps can be detected as a sequence of RFID readings, which can be generated when students' RFID readers are very close to the reagents or equipments. Thus, the system can watch the experimental process of each student according to these RFID readings.

Example 3.5 is a kind of event sequence detection with intervals restrictions. Such an application involves different types of events which occurred in a prescribed order, and there is or is no limitation on the intervals between two successive events. And there can also have a time restriction on the whole sequence, such as $\mathcal{T}_4$ in Example 3.5. In this example, different types of events will be sent to the system in different steps, and the system needs to check whether the interval between two successive events is valid. Each interval is a kind of TTL$_s$. Each event in the sequence can also have TTL$_a$ and TTL$_r$, and TTL$_r$ can also act on the whole event sequence. Both TTL$_p$ and TTL$_s$ are attached to each occurrence of the event in the sequence, which prescribes when the next event will happen.

To summarize, on the basis of analysing wide range of RFID applications, we have identified four categories of TTL: (i) TTL$_a$: the life span an RFID tag can have; (ii) TTL$_r$: the period of time in applications related only to one event, the event can be primitive, such as $\mathcal{T}$ in Example 3.3, and can also be complex, such as $\mathcal{T}_4$ in Example 3.5; (iii) TTL$_p$: the interval between two successive events within a periodically occurred event sequence; (iv) TTL$_s$: the interval between two successive events within an event sequence. The objects that each category of TTL can exert to are shown in Table 1. From an implementation viewpoint, although we design four categories of TTL, there are some internal relations among different categories, as shown in Figure 2. That is, TTL$_a$ can be viewed as a special case of TTL$_r$, while TTL$_p$ can be viewed as a special case of TTL$_s$.

Generally speaking, no matter which category of TTL is, all TTL represents the time that the system should take an action in an application. The actions can be an alert, a warning, or something else that is related to applications. Such kinds of actions are usually caused by TTL$_a$ or TTL$_r$, which are not related to event sequences. The actions can also be determining whether an expected event occurred. Such kinds of actions are usually caused by TTL$_p$ and TTL$_s$, which are related to event sequences.

TABLE I
OBJECTS THAT TTL CAN EXERT TO

Objects	**Event**		**Event Sequence**	
	Simple	Complex	Same type	Different type
TTL$_a$	√	×	×	×
TTL$_r$	√	√	×	×
TTL$_p$	×	×	√	×
TTL$_s$	×	×	√	√

IV. TTL QUERY LANGUAGE

In this section, we present the TTL query language and illustrate how this language can be used to support a range of emerging RFID applications. Since our purpose is not designing a new language, but enhancing the temporal management ability of the available languages, we leverage the available complex event query languages developed for active databases [13], [14], [15] and RFID data management systems [2], [3], [4].

Our TTL query language is a declarative language, which can be used to specify how individual events are filtered, and how multiple events are correlated via time-based and value-based conditions. Its overall structure is as follows:

[DEFINE ⟨ event_type$_1$=specification,
event_type$_2$=specification,
......
event_type$_n$=specification⟩]
EVENT ⟨event pattern⟩
[WHERE ⟨conditions for attribute values⟩]
[TTLA ⟨a list of events⟩
[{operations for invalid TTL$_a$}]]
[TTLRP ⟨a list of events⟩
[{operations for invalid TTL$_r$ of primitive events}]]
[TTLRC ⟨a period of time⟩]
[TTLP ⟨a period of time⟩]
[TTLS ⟨a list of period of times⟩]

The latter five clauses are specific for TTL queries. Both TTLRP and TTLRC are TTL$_r$, but for primitive events and complex events, respectively. Among the five TTL clauses, TTLA and TTLRP clauses have restrictions on events, while the three others have restrictions on the period of time. Furthermore, TTLRC clause restricts the time of the whole sequence while TTLP and TTLS clauses restrict the time between two successive events in the sequence. The TTL query language can capture a wide range of RFID applications related to temporal restrictions.

EXAMPLE 4.1: Let us visit again the application in Example 3.1. Suppose there are several RFID readers at the check-in counter, and the event type of the primitive events generated by these readers is `TICKET-CHECKIN`*. When a passenger passes these places, his/her ticket needs to be checked. The query can be expressed as:*

EVENT `TICKET-CHECKIN`
TTLA {*Raise an alarm: cannot check in*}

When a ticket is sold, its corresponding TTL_a *is saved in a central database. This query checks* TTL_a *when a* `TICKET-CHECKIN` *type of event is received. If its* TTL_a *is invalid, an alarm will be raised, and the passenger cannot check in.*

EXAMPLE 4.2: For the application in Example 3.3, suppose many RFID readers are installed in a building, and the event type of the primitive events generated by the tags on cards is `CARD`*. When the card belongs to a visitor,* TTL_r *needs to be checked. The query can be expressed as:*

EVENT `CARD`
WHERE *Type=*`Visitor`
TTLRP {*Raise an alarm: over staying visitor*}

Similarly, when a card is assigned to a visitor, its TTL_r *has been saved in a database. This query checks all visitors'* TTL_r *to prevent from over-staying visitors.*

EXAMPLE 4.3: For the application in Example 3.4, suppose the event type of the primitive events generated by the tags on the special part of this machine is `SPECIAL-PART`*. The query can be expressed as:*

EVENT *SEQ+(*`SPECIAL-PART`*)*
WHERE `[ID]`
TTLP *1 year*

Here the SEQ+ is for event sequence pattern in TTL_p*, and the WHERE clause is used to make sure that it is the same part. Since it is a special part, each time the system receives such an event, the system will check its last repairing record to see whether 1 year has past.*

EXAMPLE 4.4: See the application in Example 3.5. Suppose the event types of the primitive events generated by different steps are A, B, C, and D. The query can be expressed as:

EVENT *SEQ(A a, B b, C c, D d)*
WHERE $(b.ID=a.ID)\wedge(c.ID=a.ID)\wedge(d.ID=a.ID)$
TTLRC T_4
TTLS $(0, T_1)$; ; (T_2, T_3)

Here the SEQ is for event sequence pattern in TTL_s*. a, b, c, and d are the events. The WHERE clause ensures the four steps come from the same student. TTLRC clause is used to restrict the whole process. Although there is no limitation on the interval between B and C, the second ";" in TTLS clause is still required.*

EXAMPLE 4.5: Here, we give an example for negation. Suppose in airport, a baggage must arrive at a specified place to wait for being loaded onto an airplane within 60 minutes after this baggage is checked in. If no such an event, an alarm will be raised accordingly. Suppose the event type of the primitive events generated by RFID readers at check-in counters are `CHECKIN`*, and that at the specified place is* `WAIT-LOADED`*. The query can be expressed as:*

EVENT *SEQ(*`CHECKIN` *x, !*`WAIT_LOADED` *y)*
WHERE $x.ID=y.ID$
TTLS *(0, 60) minutes*

V. ALGORITHM FOR UNORDERED EVENT STREAM PROBLEM BASED ON TTL

For SEQ processing, a system can only send out query results after a complete sequence is received. There is a huge amount of intermediate stages to be stored. The system needs to check the order of the incoming events. The order required by a query is always based on the order that events occurred in the real world recorded as timestamps. However, the order that events enter into the system does not necessarily reflect the order of their occurrences. As a consequence, the system has to extract event sequences from an unordered event stream. The problem is defined as the following:

Definition 9 (*Unordered Event Stream*). An event stream is $\mathcal{E} = (e_{t_1,t'_1}, e_{t_2,t'_2}, \ldots, e_{t_i,t'_i}, \ldots)$. The list $(t_1, t_2, \ldots, t_i, \ldots)$ contains timestamps, and the list $(t'_1, t'_2, \ldots, t'_i, \ldots)$ contains the times that the events are processed by the system. They satisfy $t_1 \le t_2 \le \ldots \le t_i \le \ldots$, $t'_1 \le t'_2 \le \ldots \le t'_i \le \ldots$, $t_1 \le t'_1$, $t_2 \le t'_2$, …, $t_i \le t'_i$, …If $\exists e_{t_i,t'_i}, e_{t_j,t'_j}$, $t_i \le t_j$ and $t'_i > t'_j$, then $\mathcal{E}$ is an *unordered event stream* (UnES). △

In the sequel, we propose a new data structure and an algorithm to deal with the UnES problem, illustrated by comprehensive examples.

A. Double Level Sequence Instance List

Definition 10 (*Sequence Instance, Partial Sequence Instance*). Given a SEQ pattern, SEQ(*EventType*$_1$, *EventType*$_2$, …, *EventType*$_n$), and n events in the input event stream, e_1, e_2, …, e_n. If the event types of e_1, e_2, …, e_n are *EventType*$_1$, *EventType*$_2$, …, *EventType*$_n$, respectively, and

$$e_1.timestamp < e_2.timestamp < \ldots < e_n.timestamp \quad (1)$$

and e_1, e_2, …, e_n are also satisfy WHERE and TTLS clauses, then $(e_1, e_2, \ldots, e_n)$ is a sequence instance, denoted as $\mathcal{SI}$, and $(e_i, e_{i+1}, \ldots, e_j)$ $(1 \le i \le j \le n)$ is a partial sequence instance, denoted as $\mathcal{PSI}$. △

In query processing, the system needs to find out all sequence instances from the input event stream. Any sequential parts of a sequence instance are partial sequence instances, which satisfy the corresponding conditions in the query. From another viewpoint, sequence instances are composed of partial sequence instances. Therefore, we can first record partial sequence instances, and then use them to compose sequence instances. To this purpose, we design the following structure for recording such information.

Definition 11 (*Double Level Sequence Instance List*). Given a SEQ pattern, SEQ($EventType_1$, $EventType_2$, …, $EventType_n$), a double level sequence instance list, namely DLSIList, is defined as follows:

DLSIList = {
  $PSIList$=($PSIList_2$, $PSIList_3$, …, $PSIList_n$);
  $L^P = (L^P_2, L^P_3, \ldots, L^P_n)$;
  $SIList = \{\mathcal{SI}_1, \mathcal{SI}_2, \ldots, \mathcal{SI}_m\}$;
}

$PSIList$ has (n-1) components, and each one is a list of partial sequence instances with single event belonging to the same event type, that is,

$$PSIList_i = \{e_1, e_2, \ldots, e_{L^P_i}\}, \quad 2 \leq i \leq n \qquad (2)$$

Where L^P_i is the number of components in $PSIList_i$, and the event type of each component is $EventType_i$ in the SEQ pattern. Each component of $SIList$ will form a sequence instance, and has an attribute, namely $Changed$, to indicate whether new event has been added into this component. For each event in $\mathcal{SI}_1$, $\mathcal{SI}_2$, …, $\mathcal{SI}_m$, another two attributes are appended when it is added into $SIList$. That is, *systemstamp*, which indicates the time that this event is processed by the system, and *First*, which indicates whether this event is contained in the former components. That is, for $1 \leq j < i$ $\triangle$

$$\textbf{If}\,(e_i \in \mathcal{SI}_i) \wedge (e_i.First = ture), \textbf{then}\, e_i \notin \mathcal{SI}_j \qquad (3)$$

Intuitively, $DLSIList$ has two lists: one for partial sequence instances, and the other for sequence instances. *PSIList* is used to store single event so that the system can handle UnES problems. $SIList$ is used to store intermediate results in different stages, which may form the final results in the future. What should be noted is that events can only be added into a component of $SIList$ one by one in the order defined by the SEQ pattern. Another point should be noted is that $PSIList$ starts from 2, not 1. This is because the first event in the sequence can go into $SIList$ directly. There are three new attributes appended to different components in $DLSIList$, and they are designed for helping the updating process in this module. Before added into $DLSIList$, each event has an attribute, *timestamp*, which is assigned by RFID readers. After events get into $DLSIList$, another attribute, *systemstamp*, will be appended to them. *Timestamp* indicates the time an event occurs in the real world, while *systemstamp* indicates the time an event is processed by the system (after all transmission delays). Both of them help the system to deal with UnES problems. Here, we assume the clock of each reader is synchronized, and clock synchronization of all readers and the system is out of the scope of this paper. In the following, $PSIList_{i,j}$ represents the *j*th event in $PSIList_i$, and $\mathcal{SI}_{i,j}$ represents the *j*th event in $\mathcal{SI}_i$.

B. The Algorithm

When a SEQ pattern is received, a new $DLSIList$ will be created. At the beginning, the $DLSIList$ is empty. As the events come, the $DLSIList$ is updated and query results are sent to user when some sequence instances are obtained. The main technique is the cooperation between $PSIList$ and $SIList$ in the $DLSIList$. Events are first used to update $SIList$, and then stored in $PSIList$ to wait for being checked in the future. After a component in $SIList$ is changed, $PSIList$ is checked to see whether former events can make this component grow further.

Although the cooperation between $PSIList$ and $SIList$ can solve the UnES problem, the sizes of $PSIList$ and $SIList$ may grow dramatically. Their sizes can be reduced by exploiting TTL. When the conditions in TTL clauses indicate that some components in $PSIList$ and $SIList$ are useless, they are deleted. The details are given in Algorithm 1. Some details are explained as follows.

Algorithm 1: Algorithm for the UnES problem based on TTL

```
//Initialize a new DLSIList
1:   If (receive a SEQ pattern from Query Analyzer)
2:   {
3:     Create a $DLSIList$;
4:     $PSIList_2 \leftarrow \emptyset$; …; $PSIList_n \leftarrow \emptyset$; $L^P_2 \leftarrow \emptyset$;
5:     $SIList \leftarrow \emptyset$; $m \leftarrow 0$;
6:   }
//Update the DLSIList when a new event comes
7:   While (a new event $e$ comes)
8:   {
9:     $Index_e \leftarrow$ Event type of $e$ in the sequence pattern;
       $e.systemstamp \leftarrow SystemTime$; $e.First \leftarrow True$;
10:    Set Changed to False for all components in $SIList$;
//Add $e$ into $SIList$
11:    If ($Index_e = 1$) $m$++; $\mathcal{SI}_m \leftarrow (e, \emptyset_2, \emptyset_3, \ldots, \emptyset_n)$;
                      $\mathcal{SI}_m.Changed \leftarrow True$;
12:    If ($1 < Index_e \leq n$)
13:      For ($i = 1$; $i \leq m$; $i$++)
14:        If ($\mathcal{SI}_{i,Index_e-1} \neq \emptyset$)$\wedge$
              ($\mathcal{SI}_{i,Index_e-1}.First = True$)$\wedge$
              ($\mathcal{SI}_{i,Index_e-1}$ and $e$ satisfy TTLS clause)
15:        {
16:          If ($\mathcal{SI}_{i,Index_e} = \emptyset$) $\mathcal{SI}_{i,Index_e} \leftarrow e$;
                                $\mathcal{SI}_i.Changed \leftarrow True$;
17:          Else $m$++;
                 $\mathcal{SI}_m \leftarrow (\mathcal{SI}_{i,1}, \ldots, \mathcal{SI}_{i,Index_e-1}, e,$
                        $\emptyset_{i,Index_e+1}, \ldots, \emptyset_{i,n})$;
                 $\mathcal{SI}_{m,1}.First \leftarrow False$; …;
                 $\mathcal{SI}_{m,Index_e-1}.First \leftarrow False$;
                 $\mathcal{SI}_m.Changed \leftarrow True$;
18:        }
//Check $PSIList$
19:    For ($i = Index_e + 1$; $i \leq n$; $i$++)
20:      For($j = 1$; $j \leq L^P_i$; $j$++)
21:        For (each $\mathcal{SI}_k$ in $SIList$ whose Changed is True)
22:          If ($\mathcal{SI}_{k,i-1}$, $PSIList_{i,j}$ satisfy TTLS clause)
23:          {
24:            If ($\mathcal{SI}_{k,i} = \emptyset$) $\mathcal{SI}_{k,i} \leftarrow PSIList_{i,j}$;
                                $\mathcal{SI}_k.Changed \leftarrow True$;
25:            Else $m$++;
                   $\mathcal{SI}_m \leftarrow (\mathcal{SI}_{k,1}, \ldots, \mathcal{SI}_{k,i-1}, PSIList_{i,j},$
                          $\emptyset_{k,i+1}, \ldots, \emptyset_{k,n})$;
                   $\mathcal{SI}_{m,1}.First \leftarrow False$; …;
                   $\mathcal{SI}_{m,Index_e-1}.First \leftarrow False$;
                   $\mathcal{SI}_m.Changed \leftarrow True$;
26:          }
//Add $e$ into $PSIList$
27:    If ($1 < Index_e \leq n$) $L^P_{Index_e} ++$;
                      $PSIList_{Index_e, L^P_{Index_e}} \leftarrow e$;
//Reduce the size of $PSIList$
28:    For ($i = 2$; $i \leq n$; $i++$)
29:      For (each event $PSIList_{i,j}$ in $PSIList_i$)
30:        If (no valid events in $PSIList_{i-1}$)$\wedge$
             (future events cannnot satisfy TTLS clause)
31:          Delete $PSIList_{i,j}$; $L^P_i --$;
//Reduce the size of $SIList$
32:    For (each component $\mathcal{SI}_i$ in $SIList$)
33:      If (current system time cannot satisfy the TTLS
            clause for the last non-empty event $\mathcal{SI}i$)$\vee$
            ($\mathcal{SI}_i$ does not satisfy TTLRC clause)
34:        Delete $\mathcal{SI}_i$ from $SIList$ ; $m --$;
//Check the results in $SIList$
35:    For (each changed component $\mathcal{SI}_i$ in $SIList$)
36:      If ($\mathcal{SI}_{i,n} \neq \emptyset$) Send out this sequence; $\mathcal{SI}_{i,n} \leftarrow \emptyset$;
37:  }
```

The conditions in Line 14 make sure that the available events in a component of $SIList$ satisfy the query. Let the restriction in TTLS clause on the intervals between any two successive events are (L_{Index_e-1}, U_{Index_e-1}), then *e.timestamp* must satisfy:

$$\mathcal{SI}_{i,Index_e-1}.timestamp + L_{Index_e-1} \leq e.timestamp \leq \mathcal{SI}_{i,Index_e-1}.timestamp + U_{Index_e-1} \quad (4)$$

Since the events may come in different orders, the system needs to check $PSIList$ to see previous events for these changed components (Line 21). If a component and an event in $PSIList$ satisfy the condition in Line 22, this event is added into the component. The condition in Line 22 is similar to that of Line 14:

$$\mathcal{SI}_{k,i-1}.timestamp + L_{i-1} \leq PSIList_{i,j}.timestamp \leq \mathcal{SI}_{k,i-1}.timestamp + U_{i-1} \quad (5)$$

Events in $PSIList$ are waiting for events which are in front of them in the SEQ pattern but come later than them. If the system time indicates that no such events will come, the corresponding events in $PSIList$ can be deleted to reduce the size of $PSIList$. There are two conditions need to be checked (Line 30). The first one checks whether there is an event, *PSIList*$_{i-1,k}$ in $PSIList_{i-1}$ which satisfies

$$PSIList_{i-1,k}.timestamp + L_{i-1} \leq PSIList_{i,j}.timestamp \leq PSIList_{i-1,k}.timestamp + U_{i-1} \quad (6)$$

If yes, *PSIList*$_{i,j}$ cannot be deleted. The second one needs to check the system time. Let the current system time is *SystemTime*, and the maximum delay between the time that an RFID tag is read and the time that the corresponding primitive event is processed is *Delay*. If *SystemTime* and *PSIList*$_{i,j}$ satisfy

$$SystemTime > PSIList_{i,j}.timestamp - L_{i-1} + Delay \quad (7)$$

then no former events will come and *PSIList*$_{i,j}$ can be deleted. If *Delay* is unknown or too long, *Systemstamp* can be used as the criterion instead, so that the size of the intermediate results can be reduced.

Since ($Delay \geq PSIList_{i,j}.Delay$) and ($PSIList_{i,j}.timestamp$+$PSIList_{i,j}.Delay$ = $PSIList_{i,j}.Systemstamp$), the condition can be changed to

$$SystemTime > PSIList_{i,j}.Systemstamp - L_{i-1} \quad (8)$$

If a component $\mathcal{SI}_i$ already has j events, it is waiting for the (j+1)th event. Since events in $PSIList$ have been checked, if the system time indicates that no such events will come, $\mathcal{SI}_i$ can be deleted to reduce the size of $SIList$. The condition is

$$SystemTime > \mathcal{SI}_{i,j}.timestamp + U_j + Delay \quad (9)$$

If *Delay* is unknown or too long, since ($Delay \geq \mathcal{SI}_{i,j}.Delay$) and ($\mathcal{SI}_{i,j}.timestamp + \mathcal{SI}_{i,j}.Delay = \mathcal{SI}_{i,j}.Systemstamp$), then condition can be changed to

$$SystemTime > \mathcal{SI}_{i,j}.Systemstamp + U_j \quad (10)$$

If a query has a TTLRC clause, the system still needs to check the interval between the first event and the existing last event in $\mathcal{SI}_i$. If the current interval does not satisfy TTLRC clause, $\mathcal{SI}_i$ can be deleted now.

Generally speaking, Algorithm 1 shows that TTL is useful for reducing the size of intermediate results and increasing the system's scalability.

C. Example Application

An example application is given to illustrate the executive process of Algorithm 1. Suppose a query is:

EVENT	SEQ(A, B, C, D)
TTLRC	60
TTLS	(0, 5); ; (10, 40)

The time unit is second, and the processing for other time units is similar. This query is the event sequences of a, b, c, d, belonging to event types A, B, C, D, and satisfy the time restrictions in TTLRC and TTLS clauses. Given that the event stream is ($a_{1,2}$, $b_{5,7}$, $a_{15,21}$, $a_{16,21}$, $b_{18,20}$, $c_{19,19}$, $b_{21,22}$, $a_{25,31}$, $c_{28,31}$, $d_{30,32}$, $b_{30,30}$, $c_{55,56}$, $d_{62,62}$, $c_{65,66}$, $d_{77,77}$, $d_{78,79}$), where the first number of each event is *timestamp* and the second is *systemstamp*, and *Delay*=6s. The events came in the order of *systemstamp*, and the executive process of Algorithm 1 is shown in Figure 3.

When $a_{1,2}$ arrives, since it is the first event of the SEQ pattern, a new component with $a_{1,2}$ as the first event is added into $SIList$, and its *Check* is assigned to *True*, see Figure 3(b).

When $b_{5,7}$ arrives, first, it matches with the first component of $SIList$, and added into this component, then it is added into $PSIList$ to wait for future events, see Figure 3(c).

When $c_{19,19}$ arrives, it is added into both the first component of $SIList$ and $PSIList$. Then, since the current system time is already 19, according to the second condition in Line 30 and (7), all future a events cannot match with b_5 any more, so b_5 is deleted from $PSIList$, see Figure 3(d).

When $b_{18,20}$ arrives, no component of $SIList$ can match with it, so it is just added into $PSIList$ (Figure 3(e)). When $a_{15,21}$ arrives, a new component is added into $SIList$. b_{18} and c_{19} came before a_{15}, and are stored in $PSIList$, now they are added into the new component (Figure 3(f)).

When $a_{16,21}$ arrives, a new component is added into $SIList$ and b_{18} and c_{19} are added into the new component (Figure 3(g)). When $b_{21,22}$ arrives, the third component of $SIList$ matches with it, but this component already has a b event, so a new component is created, and a_{16} is copied to and b_{21} is added to this new component (Figure 3(h)).

When $b_{30,30}$ arrives, since the current system time is already 30, b_{18} and b_{21} are deleted from $PSIList$. Although there is no restriction on the interval between B and C, the current system time indicates that no b events before c_{19} will come, so c_{19} is also deleted, see Figure 3(i).

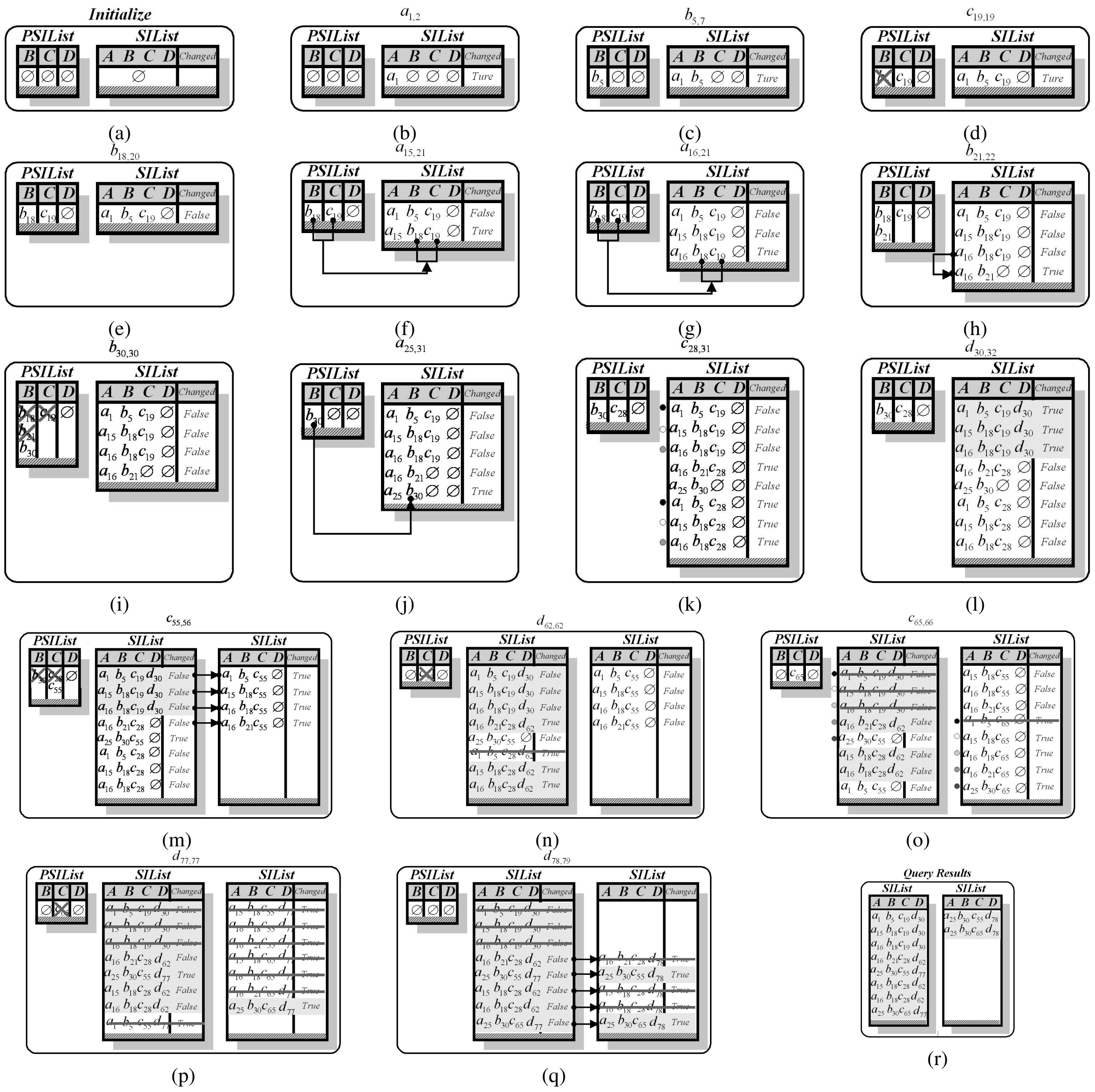


Fig. 3. The executive process of Algorithm 1 for the example in Section V-C

When $a_{25,31}$ arrives, a new component is added into $SIList$ and b_{30} is added into the new component (Figure 3(j)). When $c_{28,31}$ arrives, all the first four components of $SIList$ match with it. But the first three components already have a c event, so three new components are created. Then c_{28} is added into $PSIList$ (Figure 3(k)).

When $d_{30,32}$ arrives, three query results are obtained. Since (7) indicates that no future c events can match with d_{30} any more, so d_{30} is not added into $PSIList$ (Figure 3(l)). When $c_{55,56}$ arrives, although the last three components match with it, their *First* is not *True*. Then c_{55} is added into $PSIList$. The current system time is 56, so b_{30} and c_{28} are deleted from $PSIList$ (Figure 3(m)).

When $d_{62,62}$ arrives, the 6th component of $SIList$ does not satisfy TTLRC clause when the system reduces the size of $SIList$, and is deleted (Figure 3(n)). When $c_{65,66}$ arrives, five new components are created. When the system reduces the size of $SIList$, since the current system time is already 66, the first three components do not satisfy (9), and are deleted. The first new component does not satisfy TTLRC clause, and is also deleted (Figure 3(o)).

When $d_{77,77}$ arrives, nine components of $SIList$ match with it. But only the second and last satisfy the conditions, and the other ones do not satisfy TTLRC clause. Thus, two new query results are obtained (Figure 3(p)). When the last event, $d_{78,79}$, arrives, five new components are created. But only the second and last new components satisfy the conditions, and the three others do not satisfy TTLRC clause (Figure 3(q)). Finally, the obtained ten query results are shown in Figure 3(r).

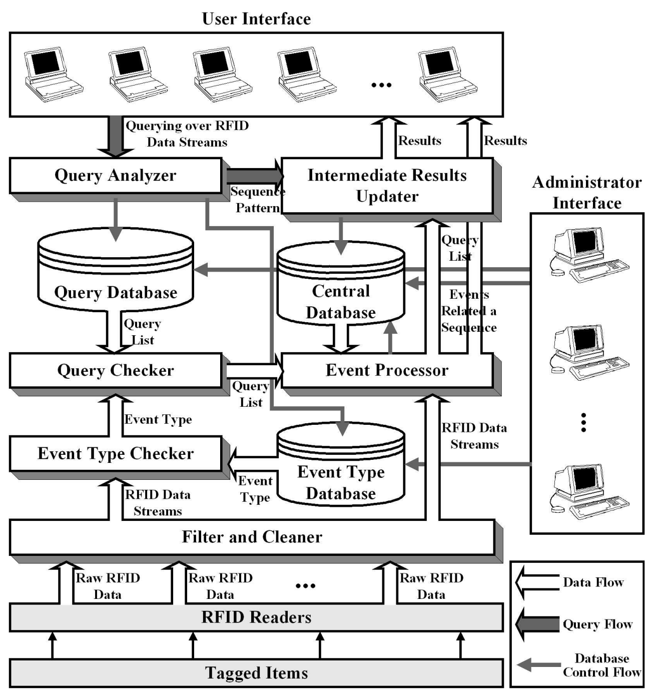


Fig. 4. System architecture of TMS-RFID

VI. System Architecture

An RFID data management system, namely *Temporal Management System over RFID data streams* (TMS-RFID), has been developed based on TTL, and the system architecture is shown in Figure 4.

TMS-RFID consists of i) three databases: the *Event Type Database*, the *Query Database*, and the *Central Database*, and ii) six modules: the *Filter and Cleaner*, the *Event Type Checker*, the *Query Checker*, the *Query Analyzer*, the *Event Processor*, and the *Intermediate Results Updater*. TMS-RFID takes the input of queries and an infinite RFID data stream and outputs the query results of events that match with the queries.

In general, there are three independent information flows in TMS-RFID, namely the *Database Control Flow*, the *Query Flow*, and the *Data Flow*. The first one is for administrators to manage the three databases. The second one represents the process where TMS-RFID handles queries submitted by users. Queries are sent to the Query Analyzer, which updates the Query Database, Event Type Database, and Intermediate Results Updater.

Data Flow in TMS-RFID flows from receiving primitive events captured by RFID readers to publishing query results to users. The process is shown in Figure 5. The raw RFID data are first filtered and cleaned, and then the events are sent to the Event Processor, waiting for being processed. When receiving the list of queries related to the events, the Event Processor starts to process the queries. First, queries that are not related to event sequences are processed, followed by the queries that are related to event sequences. Both query results will be sent to users, and the Central Database will be updated if the query results require updating the information of TTL$_a$. The function of each database and module is introduced in detail as follows.

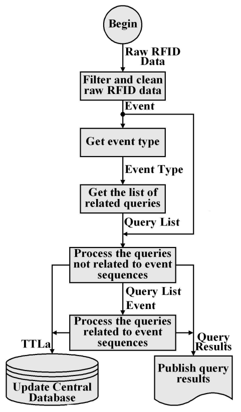

Begin
Raw RFID Data
Filter and clean raw RFID data
Event
Get event type
Event Type
Get the list of related queries
Query List
Process the queries not related to event sequences
Query List
Event
Process the queries related to event sequences
TTLa
Query Results
Update Central Database
Publish query results

Fig. 5. Data flow

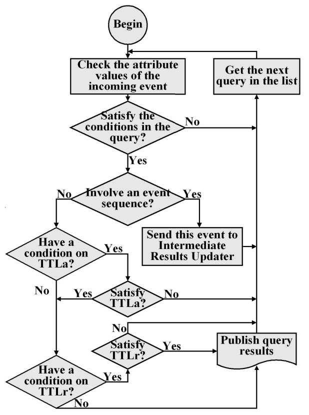

Begin
Check the attribute values of the incoming event
Get the next query in the list
Satisfy the conditions in the query?
No
Yes
No
Involve an event sequence?
Yes
Send this event to Intermediate Results Updater
Have a condition on TTLa?
Yes
No
Yes
Satisfy TTLa?
No
No
Satisfy TTLr?
Yes
Publish query results
Have a condition on TTLr?
Yes
No

Fig. 6. Event Processor

Event Type Database: This database is used to record the event types in TMS-RFID, and the main function is to provide the information of event type for the Event Type Checker when it needs to determine the event type of an incoming event. Some event types are defined by administrators, while some are defined by queries. The Event Type Database is controlled and updated by both administrators and the Query Analyzer.

Query Database: This database is used to record the queries lodged from users, and the main function is to provide the available queries for the Query Checker when it needs to determine the queries that an event type is involved. Since what the Query Checker required is the list of queries related to an event type, the Query Database saves and organizes the submitted queries according to the event types. This database is controlled and updated by administrators and the Query Analyzer.

Central Database: This database serves as the traditional database, recording the information extracted from event streams. On the other hand, it is used to record the information of TTL$_a$ and TTL$_r$ for all primitive events that enter into the system. Since TTL$_p$ and TTL$_s$ are for complex events and only used when the system checks the intermediate results, there is no need to save them in database. The Central Database is controlled and updated by administrators and the Intermediate Results Updater.

Query Analyzer: The main function of this module is to analyze the input queries. Since queries are expressed in a query language, the Query Analyzer needs to compile the query language. According to the analyzing results, the Query Analyzer needs to update the Query Database and the Event Type Database. If the new query involves a sequence, the Query Analyzer needs to notify the Intermediate Results Updater about the checking of the new sequence.

Filter and Cleaner: The main function of this module is to filter and clean the raw RFID readings, which focuses on dealing with missed readings, unreliable readings, and data redundancy. After RFID readings are processed, they are sent to the Event Type Checker and the Event Processor.

Event Type Checker: The main function of this module is to get the event type of the incoming event, and send the event type to the Query Checker for further operations.

Query Checker: This module matches a list of queries with the incoming events, and sends the list to the Event Processor so that it can check whether the incoming events meet the requirements of the available queries.

Event Processor: This module processes the queries in the list sent by the Query Checker one by one according to the incoming event. Usually, there are two kinds of conditions in a query that need to be checked for a single event: (i) attribute values, and (ii) TTL_a or TTL_r. If a query needs to extract an event sequence, the third kind of conditions, TTL_p or TTL_s, are needed to be checked. However, in this case, the conditions will be checked in the Intermediate Results Updater, and the Event Processor only needs to send the incoming event to the Intermediate Results Updater. The flowchart that the Event Processor processes a query is shown in Figure 6. For an incoming event, the related queries that do not involve an event sequence will be processed in the Event Processor, and the results will be sent to users. Additionally, if the results require the system to update TTL_a of some events, the Event Processor will update the Central Database accordingly. The queries that involve an event sequence will be processed further in Intermediate Results Updater.

Intermediate Results Updater: This module is especially designed for queries that need to extract event sequences. For SEQ+, all events belong to the same event type, and each time the system receives a valid event, query results must be sent out. Thus, for each instance of a sequence, the Intermediate Results Updater only needs to record the last event, and to check whether the next event is valid. When the Intermediate Results Updater receives a SEQ pattern, it will create a new double level sequence instance list ($DLSIList$). When new events come, it will update the $DLSILists$. When some intermediate stages reach the final stages, the Intermediate Results Updater will send the results to user. If the results require the system to update TTL_a for some events, this module will update the Central Database correspondingly.

VII. SYSTEM EVALUATION

A. Experimental Setup

We implemented the techniques presented in the previous sections in Visual C++. All experiments were performed on a PC with a Pentium IV 2.8GHz processor and 512MB memory. An event generator is developed to create a stream of events to be input into TMS-RFID. The number of events generated per second ranges from 1000 to 5000, and *Delay*=5s. There are 20 event types and 5 attributes (A_1, A_2, A_3, A_4, A_5) for each event type in addition to the timestamp. The occurrences of event types accord with Zipf distribution, with the key characteristic of Zipf distribution being set to 0. For each attribute, the number of possible values this attribute can take is chosen from the range of [10, 10000]. The time for event generation is not counted in the performance metric. Each time, after the event generator has generated events for 10 seconds, these events are sent to the system according to their delay. The stop criterion is whenever more than 10 million events have been processed.

To generate a query, the event types in EVENT clause are chosen from 20 event types according again to Zipf distribution, with the key characteristic is set to 0. The WHERE clause requires that all events in the same sequence must have the same attribute values on attribute A_1. In TTLS clause, the lower bound L_i is chosen from the range of [0, 5] in second, and the upper bound U_i is chosen from the range of [L_i, L_i+10], where $1 \leq i < n$.

The above setting ought to be reasonable for a large-scale application that would have continuous TTL queries to events detected by a few thousands of RFID readers with a quarter of million tags moving around in the real world.

B. Experimental Results

We put emphasis on testing TMS-RFID's performance on handling queries with SEQ pattern because such queries are the most complicated, which need to check not only the time restrictions but also the order of sequence.

1) Experiment on domain size: Since the WHERE clause requires that all events in the same sequence must have the same attribute values on A_1, the domain size of A_1 will affect the performance. Thus, this set of experiments is executed to test the effect of the domain size of A_1 on the performance of TMS-RFID. The sequence length is set to 2, 3, 4, 5, and 6, respectively. The domain size of A_1 increases from 500 to 10000 in step of 500, and the results are shown in Figure 7.

For different sequence lengthes, the throughput of TMS-RFID always achieves to a stable level when the domain size of A_1 increases. When the sequence length is 2, the throughput always stabilizes at about 870,000 events/sec. When the sequence length is 3, the throughput increases from about 230,000 to 470,000 events/sec, and when the domain size of A_1 is larger than 4000, the throughput stabilizes at about 450,000 events/sec. Similar performances are obtained for the sequence length being 4, 5, and 6, which stabilize at 300,000 events/sec, 210,000 events/sec, and 170,000 events/sec, respectively. To summarize, Figure 7 shows that when the domain size of A_1 is larger than about 2000, TMS-RFID always has a high throughput.

Figure 8 further shows the throughput changing with the number of events. The domain sizes are set to 500, 1000, 5000, , and 10000, and the sequence lengthes are set to three longer ones, namely 4, 5, and 6. The results show that whatever the domain size and sequence length are, the performance of the system can reach a stable level quickly.

2) Experiment on sequence length: The longer the sequence length is, the more intermediate stages are. Thus, this set of experiments is executed to test the effect of the sequence length on the performance of TMS-RFID. The domain size of A_1 is set to 500, 1000, 5000, 10000, and the sequence length increases from 2 to 6, and the results are shown in Figure 9.

Figure 9 shows that TMS-RFID scales very well with the sequence length when the domain size of A_1 is 1000, 5000,

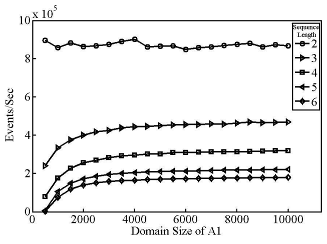
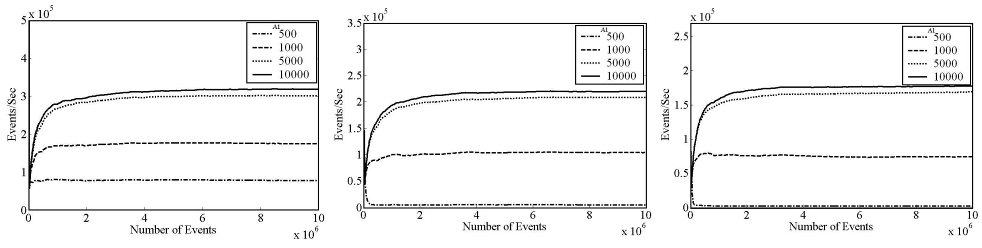


Fig. 7. The effect of the domain size of A_1



(a) (b) (c)

Fig. 8. Throughput of TMS-RFID, and the sequence length is (a) 4, (b) 5, (c) 6

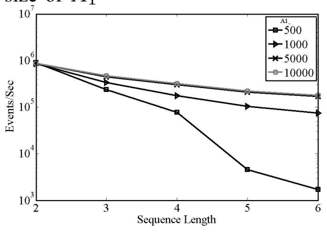
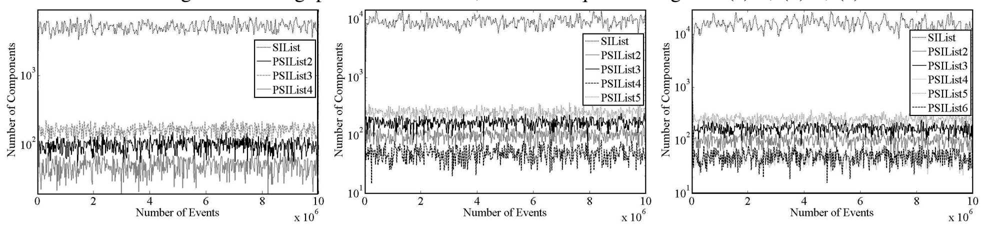


Fig. 9. The effect of the sequence length



(a) (b) (c)

Fig. 10. The size of DLSIList, and the sequence length is (a) 4, (b) 5, (c) 6

and 10000, respectively. When the domain size of A_1 is 500, the throughput of TMS-RFID drops a large level for the sequence length being 5 and 6, but still scales well for the sequence length being 2, 3, 4.

3) Scalability of TMS-RFID: The main intermediate results of TMS-RFID are stored in DLSIList, and the size of DLSIList has a great effect on the performance of TMS-RFID. If its size increases exponentially with the number of events, the system could not handle RFID data streams with huge amount of events. This set of experiments is therefore performed to check the size DLSIList. Since DLSIList includes two lists: $PSIList$ and $SIList$, the number of components in these two lists is shown in Figure 10. Here, the sequence length is set to three longer ones, namely 4, 5, and 6. The domain size of A_1 is set to 500 since the above experiments on domain size shows that this is the most difficult case. The results show that whatever the sequence length is, although the sizes of both $PSIList$ and $SIList$ fluctuate with the number of events, they stabilize at a fixed level, and do not increase exponentially. This demonstrates that TMS-RFID is quite scalable.

VIII. CONCLUSIONS

The ability of RFID technology for precisely identifying objects at low-cost and non-line-of-sight creates many new and exciting application areas. This wide range of applications will make RFID an integral part of our daily lives. Despite the obvious potential, RFID also presents a new challenge on efficiently processing large-scale and time sensitive RFID events. In this paper, we presented TMS-RFID, a system for temporal management of RFID applications over high-speed RFID data streams. The key idea behind TMS-RFID is a novel notion of TTL, which is used to control the time restrictions in various RFID applications. To solve the problem of unordered event stream (UnES), we developed a new data structure, namely $DLSIList$, to mitigate the exponential growth of the intermediate results when processing RFID queries. We demonstrated the effectiveness of TMS-RFID in a detailed performance study. The results illustrate that TMS-RFID is capable of processing high-speed RFID data streams with a good scalability.

Compared with other systems such as SASE [3], HiFi [8], and Cayuga [11], TMS-RFID provides a unique add-on middleware component to deal with the TTL queries with a solution to the UnES problem, which is one step further towards deployment of large-scale RFID applications.

IX. ACKNOWLEDGMENTS

This work is partially funded by the Australian ARC Discovery Project Grant DP0558879.

REFERENCES

[1] S. S. Chawathe et al., "Managing RFID Data," in *VLDB'04*, Canada, 2004.
[2] F. Wang et al., "Bridging Physical and Virtual Worlds: Compex Event Processing for RFID Data Streams," in *EDBT'06*, Germany, 2006.
[3] E. Wu et al., "High-Performance Complex Event Processing over Streams," in *SIGMOD'06*, USA, 2006.
[4] Y. Bai et al., "RFID Data Processing with a Data Stream Query Language," in *ICDE'07*, Turkey, 2007.
[5] F. Wang et al., "Temporal Management of RFID Data," in *VLDB'05*, Norway, 2005.
[6] A. Demers et al., "Towards Expressive Publish/ Subscribe Systems," in *EDBT'06*, Germany, 2006.
[7] M. Palmer, "Seven principles of effective rfid data management." [Online]. Available: http://www.objectstore.com/docs/articles/7principles_rfid_mgmnt.pdf
[8] S. Rizvi et al., "Events on the Edge," in *SIGMOD'05*, USA, 2005.
[9] G. Liu et al., "A Unified Approach for Specifying Timing Constraints and Composite Events in Active Real-time Database Sysems," in *RTAS'98*, USA, 1998.
[10] M. Mansouri-Samani et al., "GEM: A Generalized Event Monitoring Language for Distributed Systems," *Distributed Systems Engineering*, vol. 4, no. 2, 1997.
[11] L. Brenna et al., "Cayuga: A High-Performance Event Processing Engine," in *SIGMOD'07*, China, 2006.
[12] U. Srivastava et al., "Flexible Time Management in Data Stream Systems," in *PODS'04*, France, 2004.
[13] S. Chakravarthy et al., "Composite Events for Active Databases: Semantics, Contexts and Detection," in *VLDB'94*, Chile, 1994.
[14] N. H. Gehani et al., "Composite Events Specification in Active Databases: Model and Implementation," in *VLDB'92*, Canada, 1992.
[15] D. Zimmer et al., "On the Semantics of Complex Events in Active Database Management Systems," in *ICDE'99*, Australia, 1999.