Event Query Processing in EPC Information Services.

Tuyen Nguyen, Young-Koo Lee, Byeong-Soo Jeong, Sungyoung Lee
Department of Electronic and Information
KyungHee University, Korea
ntttuyen@oslab.khu.ac.kr, yklee, jeong@khu.ac.kr, sylee@oslab.khu.ac.kr

Abstract

Radio Frequency Identification (RFID) technology is an excellent substitute for barcodes in industry. However, the management of a large amount of RFID data, together with complicated relationships between data, in the context of responding to different kinds of queries is not well supported by traditional databases. Therefore, 1) an event-based model for managing data and 2) an algorithm for responding to event queries are proposed for RFID repositories in this paper. Our approach is based on the specification of the EPC Information Services (EPCIS) to meet the overall data requirements of the RFID community. Specifically, the object-oriented model is built on the top of relational databases so as to make the proposed combined data model highly flexible and semantic comprehensive. Based on this data model, various complicated predicates implied in different parameters of the predefined event query are interpreted into one single request to the database.

1 Introduction

In the last couple of years, the technology maturity and deploying cost decrease have made RFID become more attractive as a prominent solution for wireless identification. The ability to automatically identify and track individual objects with a unique Electronic Product Code (EPC) in RFID technology promises a great automation revolution in many areas, e.g. supply chain management, military, aviation or health care[2]. Nevertheless, RFID data is generated from various dynamic interactions, at a very high frequency, between tagged objects and different static business contexts. Data, thus, is massive and semantic-implicit[14][1][2][15]. The data needs to be classified and represented carefully in such a logical way so as to be easily queried.

Therefore, the data bridge between the RFID physical world and high-level applications, called EPC Information Service (EPCIS), is needed in any RFID system. It captures, manages data and allows accessing applications to query data easily. To achieve these, the EPCIS repository essentially needs a data model that organizes both static and timestamped data with preserving all logical relationships. Such a data model should not only facilitate the processing of EPC data queries, but also meet the basic standards of EPC-related data identified by the RFID community so as to leverage the data sharing between disparate applications. These standards are defined clearly in the EPCIS specification by EPCGlobal [3], a leading organization that develops industry-driven standards for EPC systems. However, the EPCIS specification is difficult for relational databases to support in an efficient and semantic comprehensive way. Two main problems are: 1) to represent different kinds of data together with various relationships including inheritance and 2) to respond to the predefined event query whose predicates, including hierarchical retrieval, are concealed in complicated parameters.

First, the EPCIS repository maintains two categories of data: *event data* and *master data* [5]. Both of them need semantic comprehensive representations that cover all physical storage details so as to be easily looked up. As for event data, its compactness, extensibility, its hierarchical relationships and object associations need to be considered carefully. In many cases, traditional databases do not well support these requirements.

Second, the event exchange between applications is also a big issue in heterogeneous RFID networks. To solve this problem, the Simple Event Query is predefined by the EPCIS specification to support a simple way for an accessing application to request events from an EPCIS that is not under its control. The Simple Event Query is easily requested by providing the query interface with the query name and parameters. Processing this query, on the contrary, is complicated. All the constraints on the result events are implied in the parameters. There are some constraints that typical relational databases do not provide mechanisms to deal with or can handle but in a clumsy way with many database requests. Those are the problems of: database retrieval of events in generalization hierarchy; semantic integrity of re-

Copyright © SITIS

turned events and predicates on metadata.

This paper combines the strength of both object-oriented and relational database concepts to address the above issues. An event-based data model with two flexible layers and algorithms for responding to event queries are proposed. Advantages of our model include: 1) the applicability on existing relational RFID databases without drastic changes; 2) more efficient query processing for the Simple Event Query; 3) the flexibility in the future evolvement; 4) the semantic expressivity and comprehensive of the data model; 5) more succinct query statements. Based on this data model, the processing of the Simple Event Query is simplified. Constraints implied in given parameters are interpreted into one single SQL DML statement placed on the root of the event hierarchy. Thus, each query of this type just needs one database request. The query performance is based on the existing optimization mechanisms of relational databases.

This paper is organized as follows. Section 2 summarizes some related works and backgrounds on EPC-related data standards. Section 3 designs the EER diagram for EPC data and proposes the flexible two-layer data model. Section 4 provides algorithms for processing the Simple Event Query. Section 5 concludes our work. Section 6 summarizes the experimental result and future works.

2 Related Works and Background

RFID data poses challenges to traditional relational databases and data warehouses due to its characteristics of large data flood, data inaccuracy and temporal data [1, 2]. During the past few years, RFID data management has gained considerable attention. Many solutions have been proposed to address new methods for: warehousing and mining [9, 7, 8]; cleansing [13] and compressing [11] RFID data. These works consider some particular issues, but do not provide a general data model for managing RFID data.

The data models discussed in [10, 15] are perhaps the most related to our work. In [10], Harrison et al. summarizes RFID data in simple events of three dimensions: the timestamp, the tagged object and entity that are in a dynamic interaction. RFID data, thus, is semantics-implicit. And complex queries need to be decomposed into numerous simple queries, which leads to multiple inefficient joins. Later, the authors in [15] introduce state changes of tagged objects in addition to events. The concept of state changes makes the data model more expressive and supports tracking and tracking queries efficiently. They also consider many cases of RFID applications. Yet, state changes are materialized before-hand from events. And events are atomic and semantics-poor. Since these models deal with raw data and do not consider any EPC-related data standard, they are suitable to be encapsulated in the RFID platforms developed by the same IT vendors.

To provide a standard open system for EPC usability, EPCGlobal [3] defines the EPC architecture framework [4] which includes interrelated components and standard interfaces together with core services. EPCIS sits at the highest level in this framework as a primary vehicle for data sharing between RFID partners. Its specification [5] specifies the generic structures for representing EPCIS data, the data exchanged through EPCIS, their abstract structures and service operations. Included in this specification is the Simple Event Query. This query has many complicated parameters. However, there appears to be little in the way of detailed work on the efficient implementation of this query.

Our data model is different from previous models. Based on the EPCIS specification, our data model meets the basic standards of EPC-related data identified by most enterprises. Besides, we do not deal with preliminary sensor readings as previous models, but meaningful events pushed up by RFID middlewares. Especially, a virtual object-relational layer is introduced on the top of relational tables so as to make our data model more comprehensive in terms of semantics and more flexible in terms of applicability on existing databases and future extensibility.

Our work is extended from [12]. The paper proposed an event-based data model for EPCIS with two efficient layers. However, it did not go into the details of how to process the Simple Event Query based on that proposed data model. In this work, we do focus on summarizing the data model and addressing the algorithms to process the Simple Event Query with one database request.

In the EPCIS repository, data falls into two categories: *event data* and *mater data*. Event data is timestamped and collected throughout objects' lifecycles, whereas master data is almost static and related to business entities and product/service groups. RFID data is viewed as events that are classified into a hierarchical tree of various types. More event types can be added, specialized or deleted according to each particular enterprise. However, there are four primary types:

- *Object Event*: any observation or assertion about tagged objects.
- *Aggregation Event*: containment relationship between a group of tagged objects that are contained in another one.
- *Quantity Event*: inventory report about the number of instances of a specific object class counted in a certain business context.
- *Transaction Event*: association or disassociation of tagged objects in business transactions.

All events are temporal with two timestamp fields: *eventtime*, when events occurred and *recordtime*, when they are recorded in repositories. Each core event also has fields for

four key dimensions: the objects, the date and time, the locations and the business context [5]. The values of these fields are of primitive types or vocabulary elements from master data.

Master data is organized into different vocabularies. Typical vocabularies include:

- *Object, Product, Organization*: instant-level and class-level information about EPC-tagged objects.
- *Container*: different types of containers into which objects can be packed (case, pallet, truck, etc.).
- *Readpoint*: places where EPCIS events took place.
- *Business Location*: business locations where objects are assumed to be following events.
- *BizTransaction, BizTransactionType*: information about transactions and their types.
- *BizStep*: steps in business process, e.g. shipping.
- *Disposition*: business states of EPC objects, e.g. available for sale.

Each vocabulary contains a list of identified elements. Each element is interpreted with a distinct attribute set and can have the hierarchical relationship with other elements in the same vocabulary. Conventionally, each vocabulary and event types can be normalized in several tables.

3 EPCIS Data Model

3.1 Extended Entity Relationship (EER) Model

The requirements on EPCIS data analyzed in previous sections are conceptually represented in Figure 1 using object-oriented concepts from the EER model. Master and event data are respectively distinguished in two types of entities: events (shadowed rectangles), and vocabularies (flat rectangles). There exist two kinds of relationships: inheritance among events (connections with U-shaped symbols); and normal relationships between events and vocabularies, and between vocabularies themselves (connections with a diamond-shape in the middle).

Events are classified in a specialization hierarchy. The super-event *EPCISEvent* holds the temporal attributes. Sub-events such as *ObjectEvent, QuantityEvent, AggregationEvent, TransactionEvent*, are specialized from *EPCISEvent* as deeply as necessary. In fact, each sub-event describes a particular observation on EPC objects or an object class in a specific business context. It is interpreted by vocabulary elements and non-vocabulary event fields. These fields include: *action*, the impact of an event on the lifecycle of observed entities, and *quantity*, the number of EPC objects in a specific class.

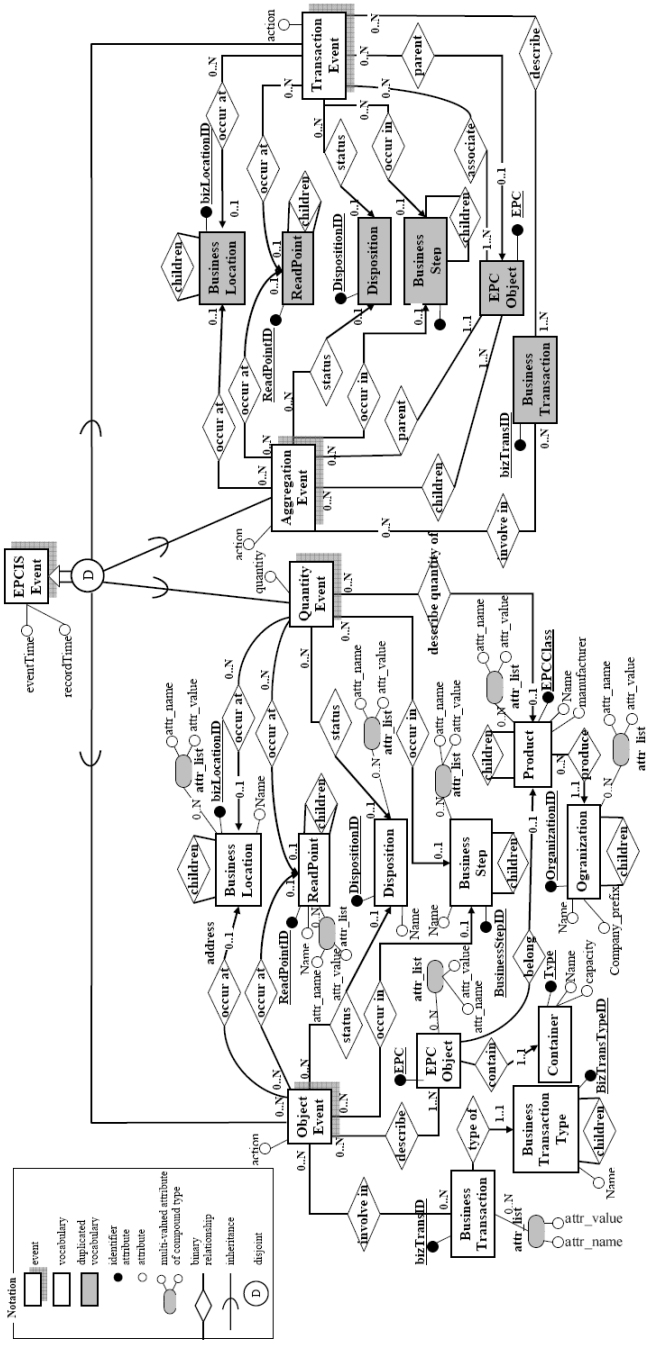

Figure 1. EER diagram for EPCIS data model.

Each vocabulary element is identified by *ID* and named with *Name*. A complex attribute *attr_list*, which is a set of pairs (*attr_name, attr_value*), is needed to keep the distinct attribute set of each element. And, the self-relationship *children* models the parent/child hierarchy inside each vocabulary. Between some vocabularies exist normal static rela-

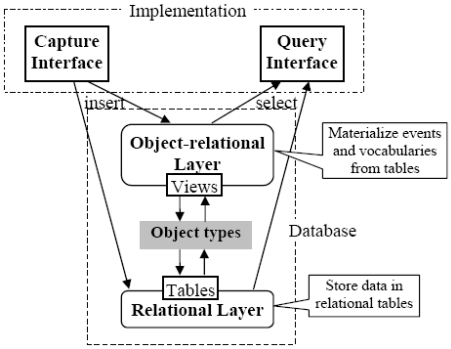

Figure 2. Two-layer framework for EPCIS object-relational database.

tionships. For example, each element in *Business Transaction* belongs to a *Business Transaction Type* and each *EPC object* may be an instance of a particular *Product* manufactured by an *Organization*.

The EER diagram in Figure 1 illustrates the fundamental entities and relationships in a typical RFID system (Some vocabularies are duplicated in the right part (gray rectangles) to avoid line overlap. The relationships and schemas of vocabularies are fully described in the left part). Events and vocabularies can be added or deleted according to the business logics of each particular application.

The benefit of this conceptual model is that it is as simple as comprehensive with respects to the EPCIS specification. Events are classified as specifically as they should be. Vocabularies are organized in a common schema so as to be easily managed and looked up. Besides, since objects in the same business process are reported in the same event, data are more compact and the associations between objects appearing in the same event are preserved.

3.2 Two-Flexible-Layer architecture for EPCIS repository

Maintaining EPCIS data in object-relational databases is obviously a better choice for EPCIS repository. With object-oriented concepts, EPCIS repositories are semantic expressive and comprehensive. Besides, such repositories will be in good harmony with the well-known object-oriented programming.

However, we do not directly deploy the object-oriented concepts in Figure 1 into physical structures in DBMSs for the following reasons. First, in some RFID systems, master data and event data may have been stored in relational tables. Applying object-oriented features directly requires the reconstruction of data structures in these systems, which may be costly. Second, the future extensions of the data model, for example, if event schemas are extended, also require object types to be redefined, which also leads to changes in physical structures. Finally, relational databases, though weak at semantic modeling, are best at data normalization and query processing with different optimization mechanisms integrated. We should exploit these features.

A two-layer model (Figure 3), therefore, is proposed: to adapt with future evolvement, to integrate with current databases and to benefit from the advantages of pure relational approaches. The first layer normalizes and maps entities and normal relationships in the conceptual model (Figure 1) into relational tables. This job is easily done by using ER mapping rules. Compound attributes such as *attr_list* are maintained in separate tables that refer to main tables via foreign key constraints.

The second layer, then, virtually applies the object-oriented concepts above these relational tables. Events and vocabularies, which are stored in several tables, are synthesized in single rich semantic objects. Constructing this layer consists of two steps. First, object types for all events and vocabularies as well as inheritance relationships between events are defined. These object types conforms to the data structures specified in the specification [5]. There is a root object type for events, called *EPCISEvent_Type*. And each object type created for an event type inherits from *EPCISEvent_Type* and encapsulates all event fields.

Similarly, each vocabulary is represented by an object type which has two special attributes – *attr_list* and *children* – in addition to *ID* and *Name*. *attr_list* is a collection of another object types of two attributes, *attr_name* and *attr_value*. *children* is also a collection of *ID*s that identify all children of the current object.

In the next step, we materialize data from relational tables and convert that data into objects of the above types. Virtual object views are built. Each object view is based on an object type and associates with a query that specifies which data in which relational tables contain the attributes for objects of that type. In other words, it does not store object instances but provides an abstract interface to allow accessing applications to view data from multiple relational tables as single objects. Inheritance relationships are also defined between event views to form the event hierarchy as mentioned before. Queries are placed on these views with succinct syntax instead of writing complex joins with multiple tables. Other data manipulation operations on views, such as update and delete, can be easily handled by using *INSTEAD_OF* triggers. Further detail on building these views can be found in our previous work [12].

Since object views are virtual, they can be easily changed by redefining object types and data-extracting queries for views. Therefore, if we want to redesign the object-relational layer, we do not have to make the drastic changes in relational layer. Besides, the object-oriented features in this layer make our model profitable in the processing of the

Simple Event Query we mention in the next section.

4 Query Event Data in EPCIS

Data exchange between heterogeneous repositories in RFID networks can be done through the Query Interface. This interface covers all the database techniques and allows users to request data by predefined queries

4.1 Introduction to SimpleEventQuery

Simple Event Query is the first query predefined by the EPCIS specification for event exchange. It has a large number of optional parameters. Each one consists of two parts: parameter name and parameter value(s). By choosing appropriate parameters, we can place various constraints to get the desired events. The parameters fall into several categories as follows

- (eventType,*value_list*): names of requested event types.
- (GE/GT/LT/LE/EQ)_*fieldName*,*value*): events whose field named *fieldName* is greater than or equal to (or other comparisons) *value*.
- (EXISTS_*fieldName*, ***void***): events that have a non-empty field named *fieldName*.
- (EQ_*fieldName*, *value_list*): events whose field named *fieldName* matches one of the values in *value_list*.
- (EQ_bizTransaction_*type*,*value_list*): Events contain a transaction list. At least, one transaction is of the type specified by *type* and matches one of the values in *value_list*.
- (MATCH_*impliedEPCField*, *value_list*): events have to have an EPC-related field whose name is implied in *impliedEPCField*. The value(s) of this field have to match one of the *patterns* specified in *value_list*.
- (WD_*fieldName*,*value_list*): events whose field named *fieldName* matches one of the values in *value_list* or is a direct or indirect descendant of them.
- (HASATTR_*fieldName*,*value_list*): events whose field named *fieldName* is a vocabulary element. And master data of this element has attributes whose names match one of the values in *value_list*.
- (EQATTR_*fieldName_attributeName*,*value_list*): events whose field named *fieldName* is a vocabulary element. And master data of this element has an attribute whose name is *attributeName* and whose value matches one of the value in *value_list*.
- (orderBy,*value*): order events by a field whose name is specified in *value*.
- (orderDirection,*value*): order events in descending (DESC) or ascending (ASC) sequence.
- (eventLimitCount,*value*): only fetch *value* top events in the result.
- (maxLimitCount, *value*): if the number of returned rows is larger than *value*, generate an exception instead of the real result.

Let us examine a simple scenario in which the sale agency of the Seoul branch of an enterprise wants to look into *those events that relate to product shipping from Seoul on November 25 at those locations whose functionality is sale*. We can request the Simple Event Query with the following parameters: (EQ_BizStep, $'$shipping$'$), (WD_bizLocation,$'$Seoul$'$), (GE_eventTime, $'$2006-11-25 00:00:00.000 GMT$'$), (LE_eventTime, $'$2006-11-25 23:59:59.999 GMT$'$), (EQATTR_bizLocation_Functionality, $'$sale$'$).

Based on our proposed data model, we propose an algorithm that can process complicated parameters of the Simple Event Query in just one request to the DBMS. First, we convert the query request into a DML statement in SQL and then execute it once against the database. The result is returned to users in an XML document.

How to process various parameters to generate all predicates in just one SQL statement is the hardest part and also our main contribution in this section. Before presenting our algorithm, let us consider some main challenges .

Retrieving events in generalization hierarchy. The query includes not just event types specified in *eventType* but also all their extension types. If *eventType* does not exist, all event types should be considered. In a relational database, we can maintain the event hierarchy in a dictionary, from which the requested event types can be retrieved. Then, we individually retrieve events of each type and union the results together. These retrievals need several requests to the database, which may slow down the performance.

With our data model, we place the query on the root event view *EPCISEvent* in the object-relational layer. And all events in the hierarchy are retrieved at once in one database request. When more event types are introduced, the algorithm still works well.

Semantic integrity of returned events The EPCIS specification requires the returned events to be identical to the originally captured events. With our data model, this semantic integrity has already been ready in the object-relational layer.

Checking the existence of metadata The meta data checking is needed to eliminate unnecessary event types before generate SQL statement

and to process parameters HASATTR_*fieldName*, EQUATTR_*fieldName_attributeName*. Existing data dictionaries in DBMSs can be used but they are large whereas event metadata is small, which makes the checking inefficient. Besides, checking foreign constraints needs several joins between dictionaries. Our solution is to maintain our own metadata dictionary for event types, called *EventDic*. *EventDic* contains information about all event fields, their data types and referred vocabularies (if exist) of event types. In disk, *EventDic* is stored in a table of <*event_type, event_fields, extensions*>, where attribute *event_fields* is a collection of <*event_field, data_type, referred_vocabulary*> and *extensions* is all children types of *event_type*. This table is updated as frequent as there are changes relating to event schemas. When EPCIS is started, *EventDic* is loaded into a hashed list in the main memory. Each entry in this list maintains a set of all event fields and extensions of a particular event type. Because the number of event types is small, we can hold *EventDic* in the main memory and use it when generating an SQL statement to reduce overhead work on checking metadata.

Ordering events of different types Since the result may include events of different schema structures, currently we can only order events by *eventtime* and *recordtime* in the generated SQL statement. We can also load the whole result into main memory, order it and return *eventLimitCount* top events to users. But, this consumes a lot of resources, especially when the result size is much larger than *eventLimitCount*.

4.2 Algorithm

In this section, we propose the algorithm ProcessSimpleEventQuery (Algorithm 1) to process the Simple Event Query. The inputs of this algorithm are the parameter list $params$, the system limitation lim and the in-memory structure *EventDic* mentioned before. lim is the number of events that EPCIS can handle based on the current resource. How to specify lim is beyond this paper. The output is the result events formatted in an XML document whose schema is provided in the EPCIS Specification.

First, invalid parameters and parameter collisions are detected to generate necessary exceptions (line 4). Parameter collisions occur in such situations when *eventLimitCount* and *maxEventCount* both occur or when *eventLimitCount* exists but *orderBy* does not. We then get the list of all the event types will be queried, including their extensions (lines 6-9). Based on $EventDic$, all event fields of each event type are retrieved. Thus, we can check and eliminate those types that do not have all the event fields desired in $params$ (line 10-13). Predicates for each remaining event type are then generated (line 13) and joined together with 'OR' operations to union events of different types in the same result set (line 15). The query is then placed on the root view *EPCISEvent* and VALUE function is used to convert each returned tuple into an object instance (line 16). To avoid running out of resources, we allocate a buffer of limited size and fetch the result into it and generate an exception if necessary (lines 18-25). If users want to get only some top events (*eventCountLimit*), lines 20-21 will put one more predicate on the query to limit the number of returned rows. This predicate is the *'SELECT TOP N'* clause in DML statement.

Algorithm GeneratePredicate is provided to generate predicates for each event type based on $params$. We can not place predicates on other event fields except for *eventTime* and *recordTime* because from the root view, events are treated as the root object type. To solve this, each tuple returned from the root view has to be treated as its own object type (line 2) by using TREAT function. For example, if we want to treat a tuple as an instance of *ObjectEvent_Type*, we use this clause `(TREAT VALUE(p) AS ObjectEvent_Type).bizStep='shipping'`

Algorithm 1: ProcessSimpleEventQuery

Input: $params \leftarrow$ list of parameter pairs $(name, value)$,
$lim \leftarrow$ limited size of query,
$EventDic \leftarrow$ a medata dictionary for event types
Output: An XML document of returned events

```
1  begin
2      P ← ∅;  // predicates generated from params
3      T ← ∅;          // all requested event types
4      if (exist invalid parameters or parameter collisions in
       params) then return QueryParameterException;
5      F ← retrieve requested event fields in params;
6      if ∃p ∈ params and p.name ='eventType' then
7          T ← event types in p.value;
8          T ← T∪ all extensions of T in EventDic;
9      else T ← get all event types in EventDic;
10     for t ∈ T do
11         if F∩ (all event fields of t in EventDic) ≠ F then
12             remove t from T;
13         else P ← P∪ generatePredicate(t,params);
14     if T = ∅ then return empty document;
15     predicates ← join all predicates in P with 'OR';
16     sql ← 'select value(e) from EPCISEvent e where ' +
       predicates;
17     if orderBy∈ params then add ORDER clause to sql;
18     if ∃p ∈ params and p.name ='maxEventCount' and
       p.value < lim then
19         lim ← p.value;
20     if ∃p ∈ params and p.name ='eventCountLimit' and
       p.value < lim then
21         restrict sql to select the first p.value rows;
22     else Restrict sql to select top (lim + 1) events;
23     Allocate buffer B of size lim for holding result;
24     Execute sql and fetch the result into B;
25     if B overflows then return QueryTooLargeException;
26     Format data in B in XML Document rDoc;
27     return rDoc;
28 end
```

Algorithm 2: GeneratePredicate

Input: $params \leftarrow$ the parameter list, $eventType \leftarrow$ an event type, $EventDic \leftarrow$ lookup dictionary for metadata
Output: Predicates placed on $eventType$

```
1  begin
2    treat each tuple as an instance of eventType;
3    P ← ∅ ;                                   /* predicates */
4    for each param ∈ params do
5      op ← extract constraint type in param.name;
6      fn ← extract the event field in param.name;
7      if op ∈ {'LT', 'LE', 'GT', 'GE'} then
8        map op into >, ≥, < or ≤;
9        p ← fn + op + p.value;
10     else if op ='EQ' then
11       if param is 'EQ_bizTransaction_type' then
12         map event field bizTransactionList to a table
           of T(type, bizTransaction);
13         p ← exists(select * from T where
           T.type =type and T.bizTransaction in
           param.value;
14       if param.value is of primitive type then
15         similar to 7;
16       else if param.value is a list of string then
17         p ← check if fn in param.value;
18
19     else if op ='MATCH' then
20       if fn is single-valued then
21         p ← match(fn, param.value))
22       else
23         map fn in to table T(epc);
24         p ← check if exists(select * from T where
           match(epc,param.value);
25
26     else if op ='WD' then
27       T ← parent/child table of the vocabulary referred by
         fn;
28       href ← select child from T start with parent in
         param.value connect by prior child = parent;
29       p ← check if fn in href;
30     else if op ='EXIST' then
31       p ← check if fn is not null;
32     else if op ='HASATTR' or op ='EQATTR' then
33       look up fn in EventDic and fetch its properties
         into d;
34       if d is null then return ∅;
35       if d.isVocabulary()=false then return ∅;
36       v ← the view of the vocabulary referred by fn;
37       look up v to get the master data for fn map attribute
         attr_list of master data into table
         T(attr_name, attr_value);
38       if op ='HASATTR' then
39         p ← check if exists(select * from T where
           attr_name in param.value);
40       if op ='EQATTR' then
41         attrname ← attribute name from
           param.name;
42         p ← check if exists(select * from T where
           attr_name = attr_name and attr_value in
           param.value);
43
44     add p to P;
45   result ← join all predicates in P by 'AND';
46   return result
47 end
```

Each parameter name contains at least two parts separated from each other by an underscore. The first part, capitalized, states the constraint type. The second part implies the event field on which the constraint is placed. Based on the constraint type and event field in each parameter (lines 5-6), we generate the predicates as follows.

Constraint types GE, GT, LT, LE, EQ are the simplest cases (lines 7-9). Predicates have the form of '*fieldname* $\geq$ *value*' or similar, e.g $bizStep = '\ shipping'$.

If EQ associates with a list of values (line 16), we use IN expression to check if the value of an event field matches one of the specified values, e.g *bizStep IN ('shipping', 'receiving')*. For EQ_bizTransaction_*Type* (lines 11-13), the TABLE expression is used to treat *bizTransactionList* like a table in the FROM clause. In effect, we join the *bizTransactionList* with the row that contains it. Predicates are then created on this join. For example, the corresponding predicate of (EQ_bizTransaction_*purchase*, $\{'123','456'\}$) is `EXISTS(SELECT * FROM TABLE(TREAT (VALUE(p) AS ObjectEvent_Type).bizTransactionList) l WHERE l.type='purchase' and l.bizTransaction IN ('123','456').`

MATCH (line 19-24) relates to EPC fields such as epcList, childEPCs, parentID and EPCClass. Each element of this parameter list may be a pure identify pattern or an URI [5]. The pure identify patterns are specified in Section 6 of [6]. As a simple example, an event of ObjectEvent (or TransactionEvent or AggregationEvent) that has 'urn:epc:idpat:sgtin:1233.1.1' as one value of its *epcList* (or *childEPCs*) is selected for (MATCH_*EPC*, 'urn:epc:idpat:sgtin:[1230-1234].*.*'). In this algorithm, a user-defined function *match* is used to check if a particular field matches one of the patterns or URIs specified in the parameter list. This function returns true if there is a match, false otherwise. Due to the limited space and the simplicity of this function, we do not specify the algorithm here.

With WD, we have to refer to the data in other vocabulary to check not only the specified values but also their descendants. (WD_bizLocation, $\{'Seoul'\}$) requires us to recursively retrieve all descendants of location 'Seoul' and check if one of them matches the value of bizLocation (line 26-28). `bizLocation IN (SELECT child FROM BizLocation_Href START WITH parent in ('Seoul') CONNECT BY PRIOR child = parent).`

With the support of $EventDic$ in main memory, handling HASATTR and EQATTR (lines 32-42), e.g (EQATTR_bizLocation_Functionality,$\{'sale'\}$), is easier and more efficient. We look up $EventDic$ to check if the field *bizLocation* exists and refers to another vocabulary. Then we easily find out the view of that vocabulary, *bizLocation_view*. In *bizLocation_view*, there is a special column called *attr_list* that holds all the attributes of each element. This *attr_list* provides information about the existence of

the attribute 'Functionality' and its value 'sale'.

```
EXISTS( SELECT * FROM bizLocation_view
loc, TABLE(loc.attr_list) l WHERE
ObjectEvent.bizLocation = loc.bizLocationID AND
l.attr_name ='Functionality' AND l.attr_value =
'Sale');
```

without $EventDic$, we have to do all the metadata checking by querying system dictionaries, which takes more time and makes the SQL statement more complicated.

When parameters in $params$ are all processed, we join them together by 'AND' operations (line 44). That means a tuple is selected only if it satisfies all the constraints indicated in $params$.

5 Conclusion

In this paper, we contribute an event-based data model with two flexible layers for the EPCIS. Relational layer assures data normalization, integrity and takes advantage of existing query processing mechanisms. Object-relational layer brings the flexibility in the future evolvement and in the synthesis of existing relational data. It also makes the data model semantic comprehensive by supporting object-oriented features in modeling semantics and hierarchical relationships. As the result, it facilitates the query statement since reducing many join clauses. Besides, objects that experience the same business processes are recorded in the same event so that we can reduce the data volume and preserve the internal relationships between objects. We also provide methods to process the Simple Event Query. Various complicated parameters, including joining, hierarchical retrieving, metadata checking, etc., are successfully converted into SQL predicates in a single SQL statement. That SQL statement needs just one database request.

6 Experiments an Future Work

The proposed data model and algorithm are parts of our project to build an EPCIS for industrial enterprises based on the EPCIS specification. Now, we finished the first prototype of EPCIS. It consists of the repository, which we designed in Oracle 10g Release 1, and the implementation of the two standard interfaces using Java: capture interface and query interface. To populate data for our model, we built an event generator in Java, which randomly creates events in different types and pushes them to the EPCIS. The EPCIS catches these events and stores them in the repository. The Simple Event Query is one predefined query we have ever implemented. It ran well with our algorithm. And when we introduce more event types, it keeps giving the correct answers.

In the future, we consider partitioning the event data by time and handle queries on temporal data based on these partitions. It will increase the query performance and easy to keep track of new data and maintain old data. Besides, exploiting the path history of EPC objects is also an interesting problem since queries on traversal paths are specific to RFID data and not supported by current DBMSs.

7 Acknowledge

This research was supported by the Ministry of Commerce, Industry, and Energy (MOCIE), Korea (10016466).

The authors wish to thank Prof. Brian J. d'Auriol of KyungHee University for his advices and manuscript proofreading.

References

[1] S. S. Chawathe, V. krishnamurthy, S. Ramachandran, and S. Sarma. Mananaging RFID data. In *VLDB'04*, pages 1189–1195, Toronto, Canada, 2004.

[2] R. Derakhshan, M. E. Orlowska, and X. Li. RFID data management: Challenges and opportunities. In *Proceedings of the IEEE International Conference on RFID*, pages 175–182, Grapevine, TX, USA, 2007.

[3] EPCGlobal. http://www.epcglobalinc.org/.

[4] EPCglobal. *The EPCglobal Architecture Framework.* EPCglobal, July 2005.

[5] EPCglobal. *EPC Information Services (EPCIS) Version 1.0 Specification, Last Call Working Draft Version of 24 March 2006.* 2006.

[6] EPCglobal. *EPCglobal Tag Data Standards Version 1.3.* EPCglobal, March 2006.

[7] H. Gonzalez, J. Han, and X. Li. Flowcube: Constructuing RFID flowcubes for multi-dimensional analysis of commodity flows. In *VLDB'06*, pages 834–845, Seoul, Korea, 2006.

[8] H. Gonzalez, J. Han, and X. Li. Mining compressed commodity workflows from massive RFID data sets. In *CIKM'06*, pages 162–171, 2006.

[9] H. Gonzalez, J. Han, X. Li, and D. Klabjan. Warehousing and analyzing massive RFID data sets. In *ICDE'06*, 2006.

[10] M. Harrison. EPC information service - data model and queries. Technical report, February 2003.

[11] Y. Hu, S. Sundara, T. Chorma, and J. Srinivasan. Supporting RFID-based item tracking applications in oracle DBMS using a bitmap datatype. In *VLDB'05*, Trondheim, Norway, 2005.

[12] T. Nguyen, Y.-K. Lee, R. Huq, B.-S. Jeong, and S. Lee. A data model for EPC information services. In *DEWS'2007*, March 2007.

[13] J. Rao, S. Doraiswamy, H. Thakkar, and L. S. Colby. A deferred cleansing method for RFID data analytics. In *VLDB'06*, Seoul, Korea, 2006.

[14] S. Sarma. Integrating RFID. *ACM Queue*, 2(7):50–57, October 2004.

[15] F. Wang and P. Liu. Temporal management of RFID data. In *VLDB'05*, Trondheim, Norway, 2005.