RFID Data Processing in Supply Chain Management using a Path Encoding Scheme

Chun-Hee Lee, Chin-Wan Chung

Abstract

RFID technology can be applied to a broad range of areas. In particular, RFID is very useful in the area of business, such as supply chain management. However, the amount of RFID data in such an environment is huge. Therefore, much time is needed to extract valuable information from RFID data for supply chain management. In this paper, we present an efficient method to process a massive amount of RFID data for supply chain management. We first define query templates to analyze the supply chain. We then propose an effective path encoding scheme that encodes the flows of products. However, if the flows are long, the numbers in the path encoding scheme that correspond to the flows will be very large. We solve this by providing a method that divides flows. To retrieve the time information for products efficiently, we utilize a numbering scheme for the XML area. Based on the path encoding scheme and the numbering scheme, we devise a storage scheme that can process tracking queries and path oriented queries efficiently on an RDBMS. Finally, we propose a method that translates the queries to SQL queries. Experimental results show that our approach can process the queries efficiently.

Index Terms

RFID, Supply Chain Management, Path Encoding Scheme, Prime Number

I. INTRODUCTION

As the size of the RFID tag becomes smaller and the price of the RFID tag gets lower, RFID technology has been applied to many areas. It is different from existing technologies, such as barcode systems and magnetic card systems, in that barcode systems and magnetic card systems require contact between a detector and an object. RFID readers in RFID systems, on the other hand, can detect RFID tags without contact.

A typical example of RFID technology being used is supply chain management. In supply chain management, in order to know the movements of products easily, an RFID tag is attached to a product. If the product with an RFID tag moves or stays near the detection region, RFID readers will detect RFID tags, and the detected information will be generated in the form of (*tag identifier* [1], *location*, *time*). As the flow of the product is detected easily by RFID technology, it is observed that RFID can be used to revolutionize supply chain management.

[1]As a tag identifier, RFID systems use EPC (Electronic Product Code) [2], which is a coding scheme of RFID tags, to identify the tags uniquely.

The RFID data generated in each region (i.e., (*tag identifier, location, time*)) is sent to the central server. Then, the data is transformed into stay records in the form of (*tag identifier, location, start time, end time*). While raw RFID data has many duplicates, the transformed data (i.e., stay records) does not have duplicates. We can represent how long a tag stays at a location by the $start\ time$ and $end\ time$ of stay records. The stay records for each tag compose a trace record that gives us movement history with time information for the tag. In this paper, we will use trace records instead of stay records for storing RFID data in the central server.

We can store RFID data in a relational table BASIC_TABLE(TAG_ID, LOC, START_TIME, END_TIME) as a straightforward method, where TAG_ID represents the tag identifier, LOC the location, START_TIME the time when the tag enters the location, and END_TIME the time when the tag leaves the location. The queries that analyze the supply chain are related to the product transition. For example, a manager may ask the query "Find the number of laptops that go through locations $Factory1$, $Distribution\ Center1$, and $Store1$." To evaluate the query with BASIC_TABLE, we must perform self-joins of BASIC_TABLE many times. Also, the size of BASIC_TABLE is big. Therefore, it requires a lot of time to execute the query with BASIC_TABLE.

To support efficient path dependent aggregates for RFID data, Gonzalez et al. propose a new warehousing model [14]. In the model, STAY_TABLE(GID, LOC, START_TIME, END_TIME, COUNT) is used to store RFID data efficiently. In many RFID applications, products usually move together in large groups during the early stages, and move in small groups during the later stages. Therefore, they represent stay records with the same location and time by one record such as (*tag identifier list, location, start time, end time, the number of tags with the same location and time*). To link locations efficiently (i.e., to perform self-joins of STAY_TABLE efficiently), the tag identifier list is encoded by the prefix encoding scheme. The value of the prefix encoding scheme corresponds to GID in STAY_TABLE. To know whether two locations (A and B) are linked, the prefix encoding scheme checks whether the GID that corresponds to A is the prefix of the GID that corresponds to B.

Therefore, the approach by Gonzalez et al. [14] reduces the size of the table significantly and improves the join cost. However, if products do not move together in large groups, the size of the table will not be largely reduced. In this case, the performance of the approach by Gonzalez et al. does not have a great benefit compared to that of the approach with BASIC_TABLE. Also,

since the prefix encoding scheme uses a string comparison in joining tables, it needs much more time than a number comparison. Thus, we propose a new approach to store RFID data and process queries for supply chain management. Since the queries related to the object transition in supply chain management was not defined formally in [14], we first define query templates for tracking queries and path oriented queries to analyze the supply chain. Then, to solve the above drawbacks, we propose a new approach.

While Gonzalez et al. [14] focus on groups in which products move together, we focus on the movement for each tag. The movement of one tag makes a path in supply chain management. Therefore, we devise a path encoding scheme to process tracking queries and path oriented queries efficiently. The proposed path encoding scheme can encode a path with only two numbers, which is motivated by Wu et al. [39]. The numbers are called Element List Encoding Number and Order Encoding Number, and will be explained in detail subsequently. In [39], in order to determine whether a relationship exists between two elements in an XML tree, a property of prime numbers is used. In addition, to preserve the global order for elements, simultaneous congruence values are used. In this paper, we use the property of prime numbers to encode nodes in a path, and simultaneous congruence values to encode the ordering among nodes in the path. Our encoding scheme is based on the Fundamental Theorem of Arithmetic and the Chinese Remainder Theorem. Using the proposed path encoding scheme, we can efficiently retrieve paths that satisfy the path condition in a query. To store the time information related to the movement, we separate the time information from trace records. We use the region numbering scheme [40], which is widely used in the XML area, in order to retrieve the time information efficiently. Based on the path encoding scheme and the region numbering scheme, we devise a new relational schema to store the path information and the time information for tags.

If a path is long, its Element List Encoding Number and Order Encoding Number are very large and an RDBMS may not support the data type for storing very large numbers. Although we can implement our encoding scheme on a special purpose database engine, it has many disadvantages. Therefore, we propose a method to divide a path. If we cannot store a path by using one attribute in an RDBMS, we divide the path into multiple path segments, each of which can be stored by using one attribute.

Based on the encoding schemes described above, we translate query templates into SQL queries. However, in the case of dividing paths, the SQL translation is not straightforward due

to the computation overflow. Therefore, we propose a method to translate query templates into SQL queries, which do not cause the computation overflow, by using mathematical analyses.

A. Contributions

The contributions of the paper are as follows:

- **Encoding Scheme to Encode Flows of Products** To support tracking queries and path oriented queries efficiently, we propose a path encoding scheme for flows of products. Element List Encoding Number is computed by the product of the prime numbers that correspond to the nodes in a path. Based on the Chinese Remainder Theorem, Order Encoding Number is computed. By using the two numbers, we can easily find paths that satisfy the path condition.
- **Efficient Relational Schema and Query Translation** We propose an efficient relational schema with the path encoding scheme and the region numbering scheme. Based on the schema, we propose a method that translates tracking queries and path oriented queries into SQL queries.
- **Effective Method to Store Long Paths** A long path is common in typical supply chain management. The product of prime numbers for a long path can become excessively large, which can cause a limit to the path length. To avoid overflows in storing large numbers, we provide a method that divides the long path into several path segments.
- **Extended Query Translation for Handling Long Paths** When we store a long path by dividing the path, the computation overflow may occur during the execution of the translated SQL query. To overcome the computation overflow, we extend the method to translate query templates into SQL queries by using mathematical formulas.
- **Defining of Query Templates to Analyze the Supply Chain** We define query templates to analyze the supply chain. We consider query templates for tracking queries and path oriented queries. For path oriented queries, we provide a grammar to effectively express the path condition for products like XPath [3].
- **Experimentation to Validate Our Proposed Approach** Through extensive experiments, we show that our approach is efficient. Experimental results show that the query performance of our approach is better than the recent approach in most queries.

B. Organization

The rest of the paper is organized as follows. In Section II, we discuss related work on managing RFID data. In Section III, we deal with data formats and define query templates for tracking queries and path oriented queries in supply chain management. We show the architecture that stores RFID data and process queries in Section IV. We describe our path encoding scheme in Section V and devise a storage scheme based on our path encoding scheme in Section VI. We propose a method to translate tracking queries and path oriented queries into SQL queries in Section VII, and experimentally show the superiority of our approach in Section VIII. We make a conclusion in Section IX.

II. Related work

In contrast to the initial study for RFID, which focuses on the device, many studies have been done recently to manage RFID data as the amount of RFID data has become large.

The system architecture for managing RFID data is discussed in [8], [11], [17]. Bornhövd et al. [8] describe the Auto-ID infrastructure (Device Layer/Deivce Operation Layer/Business Processing Bridging Layer/Enerpise Application Layer). The Auto-ID infrastructure integrates data from smart items such as RFID and sensor with existing business processes. Chawathe et al. [11] suggest a layered architecture for managing RFID data (Tag/Reader/Savant/EPC-IS/ONS server). In the work of Hag et al. [17], two software layers are proposed: the ubiquity agent architecture and the tag centric RFID application architecture. The ubiquity agent architecture is for a general-purpose core architecture. The tag centric RFID application architecture is for the RFID application architecture that specializes the generic agent architecture.

RFID data is generated in the form of streaming data and then it is stored in a database for data analyses. Therefore, there are two types of approaches for managing RFID data. One approach is on-line processing for RFID data and it is related to data stream processing [5], [6], [19], [36]. The other approach is off-line processing and it is related to stored data processing [4], [7], [13], [14], [18], [29], [35].

In the aspect of viewing RFID data as data streams, event processing and data cleaning have been studied. RFID data has a temporal property, which is important in analyzing data. Therefore, a temporal RFID event can be defined. However, this cannot be well supported by traditional ECA (Event-Condition-Action) rule systems. Therefore, Wang et al. [36] formalize the specification

and semantics of RFID events and rules including temporal RFID events. Also, they propose a method to detect RFID complex events efficiently. Bai et al. [6] explore the limitation of SQL queries in supporting temporal event detection and discuss an SQL-based stream query language to provide comprehensive temporal event detection.

Inevitably, RFID data has some errors, such as duplicate readings and missing readings. To rescue missing readings, the first declarative and adaptive smoothing filter for RFID data (SMURF) [19] is proposed. SMURF controls the window size of the smoothing filter adaptively using statistical sampling. Also, Bai et al. [5] propose several methods to filter RFID data.

In the aspect of viewing RFID data as stored data, various approaches exist that can manage RFID data. Since RFID readers can detect RFID tags easily and quickly, object tracking that uses RFID is widely used. To trace tag locations, a new index is proposed by Ban et al. [7]. They point out the problem of representing trajectories in RFID data and propose a new data model to solve it. Also, they devise a new index scheme called the Time Parameterized Interval R-Tree as a variant of the R-Tree. In order to represent a collection of tag identifiers generated by item tracking applications compactly, a bitmap data type is proposed and a set of bitmap access and manipulation routines is provided in [18]. Agrawal et al. [4] deal with the tracing of items in distributed RFID databases. They introduce the concept of traceability networks and propose an architecture for traceability query processing in distributed RFID databases.

Since RFID data has a temporal property, it is difficult to model RFID data by using the traditional ER-model. Therefore, Wang et al. [35] propose a new model called the Dynamic Relationship ER Model (DRER) which simply adds a new relationship (dynamic relationship). They also propose methods to express temporal queries based on DRER by using SQL queries. Although we can use the above techniques, such as indexes and bitmap data types, to process tracking queries and path oriented queries, it is inefficient to process the queries since they do no consider the object transition.

Gonzalez et al. [14] propose a new warehousing model for the object transition and a method to process a path selection query. To get aggregate measures on a path, they join tables many times. Therefore, they use compression in order to reduce the join cost. However, the proposed compression is useless if products do not move together in large groups.

A preliminary version of this work appeared in [22]. While an efficient storage scheme and query processing for RFID data in supply chain management were proposed by Lee and Chung

[22], they cannot be applied when the path length is large since Element List Encoding Number and Order Encoding Number cannot be stored by using the data types that an RDBMS supports. Long paths are common in typical supply chain management because there are many checkpoints in a route. In such a case, the work of Lee and Chung [22] should be implemented on a special-purpose DBMS that can support very large numbers.

To solve the problem, we propose a method to divide a long path. By dividing the path, we can store the long path without modifying an RDBMS or implementing a special purpose database engine. However, in the case of dividing the path, if we use the translation method proposed in [22], computation overflow may occur. Therefore, we provide a method to translate query templates into SQL queries, which do not cause the computation overflow, by mathematical analyses. Also, we redo the entire experiment with data sets generated from a real-life example and additional queries. In addition, managing object transitions has been dealt with in various areas with different views and terminologies.

In the vehicle tracking area, most papers focus on how to detect and track vehicle objects than how to manage the generated tracking data. To track vehicles, video cameras are generally used [16], [20], [21]. To detect and track vehicles from video streams efficiently, Kanhere et al. [20] and Kim et al. [21] propose image processing-based methods while Haidarian-Shahri et al. [16] use a graph-based approach. They do not consider how to manage the tracking data for vehicles. In this paper, we focus on managing the flows of products that are similar to the tracking data.

In XML databases, the path information can be stored in XML data format. In [10], [25], [34], XML data can be represented as a compact data (i.e., compressed data) and direct query processing is available in such a compact representation. Also, to index and query XML data, various approaches are proposed [23], [24], [30], [33], [37], [39], [40]. However, in an environment where so much path data is generated, the existing approaches in the XML area cannot manage and analyze the path information for products efficiently since they do not focus on many ancestor/descendant relationships in a query.

In spatio-temporal databases, the data generated from moving objects (e.g., vehicles) is manipulated. As moving objects go around, the object movement information is generated and its collection composes the trajectories of moving objects. In [12], [15], the data structure and model for moving objects are proposed. Also, in [26]–[28], [32], [38], to index and query the data from moving objects, various index structures and algorithms are used. In the aspect of representing

object movements, the trajectory data of moving objects and the flow information of products in supply chain management using RFID are similar. Both the trajectories of moving objects and the flow information of products can be represented as the set of <ObjectID, Location, Time>. However, the locations of products in RFID applications are limited since they are detected through the fixed RFID readers. In contrast, the locations of moving objects in spatio-temporal databases are not limited. Also, in spatio-temporal databases, most queries are related to the locations of objects (e.g., k-nearest neighbor query, range query) rather than the flow of objects. Therefore, query processings in these two areas are quite different, although the data can be represented in a similar format.

III. Problem Definition

Raw RFID data consists of a set of triples (*TagID, Loc, Time*), where

- *TagID* is the Electronic Product Code (EPC) of the tag and is used for identifying the tag uniquely.
- *Loc* is the location of the RFID reader which detects the tag
- *Time* is the time of detecting the tag

We translate raw RFID data generated in supply chain management into a set of stay records that do not have duplicates. A stay record has the form (*TagID, Loc, StartTime, EndTime*), where

- *TagID* and *Loc* are the same as above
- *StartTime* is the time when the tag enters the location
- *EndTime* is the time when the tag leaves the location

From the stay records of a tag, we can construct the trace record of the tag in the form of *TagID*: $L_1[S_1, E_1] -> \cdots -> L_n[S_n, E_n]$, where $L_1, \cdots, L_n$ are the locations where the tag is detected, S_i is $StartTime$ at the location L_i, E_i is $EndTime$ at the location L_i, and $L_i[S_i, E_i]$ is ordered by S_i. We use a set of trace records instead of raw RFID data in our systems.

Example 1: Figure 1-(a) shows raw RFID data and Figure 1-(b) shows a set of trace records that corresponds to raw RFID data. From (tag1, A, 2) and (tag1, A, 3), we get the stay record (tag1, A, 2, 3). Similarly, we can compute stay records (tag1, B, 5, 7), and (tag1, C, 8, 9) for tag1. Finally, we can compute the trace record, $tag1 : A[2, 3] -> B[5, 7] -> C[8, 9]$.

To analyze the supply chain, we use queries for the object transition. Although Gonzalez et al. [14] use a path selection query, it is insufficient to express the relationship between locations.

(tag1, A, 2), (tag4, A, 2), (tag2, A, 2), (tag3, A, 2), (tag1, A, 3), (tag2, A, 3), (tag4, A, 3), (tag3, A, 3), (tag3, B, 5), (tag1, B, 5), (tag2, B, 5), (tag4, B, 5), (tag1, B, 6), (tag4, B, 6), (tag3, B, 7), (tag1, B, 7), (tag2, B, 7), (tag4, B, 7), (tag2, C, 8), (tag1, C, 8), (tag3, C, 8), (tag3, C, 9), (tag1, C, 9), (tag2, C, 9), (tag4, C, 13), (tag4, C, 14), (tag4, C, 16)

(a) Raw data

tag1: A[2,3]->B[5,7]->C[8,9]

tag2: A[2,3]->B[5,7]->C[8,9]

tag3: A[2,3]->B[5,7]->C[8,9]

tag4: A[2,3]->B[5,7]->D[13,16]

(b) Trace records

Fig. 1. Raw Data and Trace Records

Therefore, we define query templates to analyze the supply chain. We consider query templates for tracking queries and path oriented queries. The tracking query finds the movement history for the given tag. The path oriented query is classified into the path oriented retrieval query and the path oriented aggregate query. The path oriented retrieval query finds tags that satisfy given conditions (including a path condition) and the path oriented aggregate query computes the aggregate value for tags that satisfy given conditions (including a path condition). In query templates for the path oriented retrieval query and the path oriented aggregate query, we provide a grammar to effectively express path conditions for products like XPath [3].

Figure 2 shows the formal definition for query templates in supply chain management. There are three query templates (tracking query, path oriented retrieval query, path oriented aggregate query). The query template for a tracking query has only a tag identifier to trace the tag. The path oriented retrieval query consists of Path Condition and Info Condition. The path oriented aggregate query needs Aggregate Function as well as Path Condition and Info Condition. Path Condition uses a grammar similar to XPath. Path Condition consists of a Step sequence. Each Step has a parent axis (/) or an ancestor axis (//). Also, each Step may have Time Conditions. Time Condition is the predicate for $StartTime$ and $EndTime$. The argument for Aggregate Function (i.e., Time Selection) allows only the time information. We can express various queries in supply chain management by using the query templates in Figure 2. We show some examples for tracking queries and path oriented queries in Figure 3.

Our problem definition is as follows: For an environment where there is a large amount of RFID data in supply chain management and users issue tracking queries and path oriented queries, devise an efficient storage scheme and processing method for the queries.

[1] Tracking Query= <TagID = id>
[2] Path Oriented Retrieval Query = <PathCondition, InfoCondition>
[3] Path Oriented Aggregate Query = <AggregateFunction, PathCondition, InfoCondition>

PathCondition -> (Step)*
Step-> /Loc[TimeCondition] | //Loc[TimeCondition]
AggregateFunction->count() | sum(TimeSelection) | avg(TimeSelection)
| max(TimeSelection) | min(TimeSelection)
TimeSelection -> Loc.StartTime | Loc.EndTime | Loc.EndTime – Loc.StartTime

---

* Info Table has the information for tags such as product name, manufacturer, and price. Info Condition is the predicate for the attributes of Info Table and may be empty (Ex. Product Name = 'laptop', Price>10000).

** Time Condition is the predicate for start time and end time (Ex. StartTime > 5, EndTime<10).

*** Loc is the location name of a detection region.

Fig. 2. Query Templates for Tracking Queries and Path Oriented Queries

Semantics	Query
Find the movement history for the tag whose identifier is XXX (Tracking Query).	TagID = XXX
Find the tags that go through locations L_1, ..., L_n (Path Oriented Retrieval Query).	<//L_1//...//L_n>
Find the tags that go through locations L_1, ..., L_n where the duration at L_1 is less than T (Path Oriented Retrieval Query).	<//L_1 [(EndTime-StartTime)<T]//..//L_n>
Find the average duration time at L_2 for tags that go from L_1 directly to L_2 (Path Oriented Aggregation Query).	<avg(L_2.EndTime - L_2.StartTime), //L_1/L_2>
Find the minimum start time at L_2 for laptops that go from L_1 to L_2 (Path Oriented Aggregation Query).	<min(L_2.StartTime), //L_1//L_2, ProductName='laptop'>

Fig. 3. Examples for Tracking Queries and Path Oriented Queries

IV. ARCHITECTURE

Figure 4 shows the architecture to store RFID data, and process tracking queries and path oriented queries in supply chain management. The central server receives raw RFID data from various regions whose format is $(TagID, Loc, Time)$. The raw RFID data is transformed into trace records after sorting the RFID data by the tag identifier and the time (i.e., $TagID$: $Loc_1[S_1, E_1] - > \cdots - > Loc_n[S_n, E_n]$). The path information in the trace records is stored by using Element List Encoding Number and Order Encoding Number and the time information in the trace records is stored by using Region Number. Since we use prime numbers instead

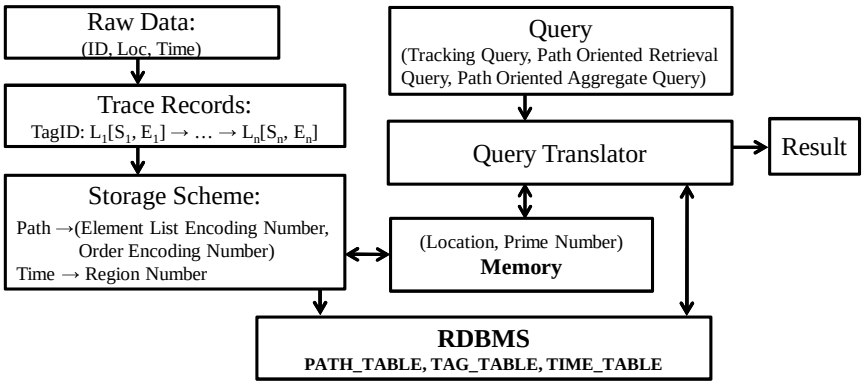


Fig. 4. Architecture

of location names, the (Location, Prime Number) list is kept in memory as a hash structure. Based on the above encoding schemes, we store trace records by using the relational schema (PATH_TABLE, TAG_TABLE, and TIME_TABLE). If a user requests a tracking query, a path oriented retrieval query, or a path oriented aggregate query, Query Translator translates it into an SQL query. Then, the SQL query is processed by an RDBMS and the result is sent to the user.

V. Path Encoding Scheme for Tag Movements

In this section, we devise a new path encoding scheme to represent tag movements compactly and efficiently. A product with an RFID tag goes through many locations. Its movements are represented by trace records in the form of *TagID*: $L_1[S_1, E_1] - > \cdots - > L_n[S_n, E_n]$. In supply chain management, it is important to analyze the object transition. To manage the object transition efficiently, we first extract the flow information from trace records, and it composes the path $L_1 - > \cdots - > L_n$. We then propose a path encoding scheme for encoding the path $L_1 - > \cdots - > L_n$ in order to analyze the object transition efficiently.

tag1: A[2,3]->B[5,7]->C[8,9]	tag6: A[2,3]->E[4,6]->C[7,8]
tag2: A[2,3]->B[5,7]->C[8,9]	tag7: A[2,3]->E[4,6]->C[7,8]
tag3: A[2,3]->B[5,7]->C[8,9]	tag8: A[2,3]->E[4,6]
tag4: A[2,3]->B[5,7]->D[13,16]	tag9: A[2,3]->D[4,5]
tag5: A[2,3]->B[7,8]->D[14,18]	tag10: A[2,3]->D[5,6]

Fig. 5. Example of Trace Records

We can represent different paths for each product by a tree structure. Figure 6 shows the tree structure for trace records in Figure 5. We assume that there is no cycle in a path. However, we will explain how to store and query paths with cycles later. The path of each tag composes the tree by eliminating duplicate nodes. The numbers beside the nodes are prime numbers, which will be explained subsequently. In Figure 6, the dark nodes mean that there are tags whose final location is the node. We store all paths that end at dark nodes, which are $A- > B- > C$, $A- > B- > D$, $A- > E$, $A- > E- > C$, and $A- > D$. Although a huge amount of RFID data is generated, the size of the tree is small.

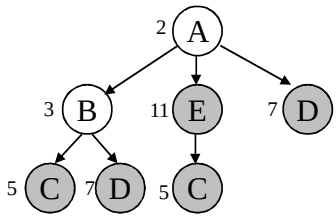


Fig. 6. Tree Structure for the Trace Records

To encode a path, we can consider various techniques in the XML area [23], [25], [30], [33], [37], [39], [40]. However, those techniques are inefficient in processing queries that have many ancestor-descendant relationships, such as "Find the tags that go through locations L_1, L_2, L_3 ($//L_1//L_2//L_3$)." In supply chain management, we need such queries to analyze the flows of tags. Before proposing a new path encoding scheme, we introduce two well known theorems [31].

Theorem 1: The Fundamental Theorem of Arithmetic (The Unique Factorization Theorem): Any natural number greater than 1 is uniquely expressed by the product of prime numbers.

For example, $231 = 3 \times 7 \times 11$ and the product of any other prime number combination for 231 does not exist.

Theorem 2: The Chinese Remainder Theorem: Suppose that $p_1, p_2, \cdots, \ and \ p_n$ are pairwise relatively prime numbers. Then, there exists X with $0 \leq X < p_1 p_2 \cdots p_n$ which solves the system of simultaneous congruences.

$$\begin{aligned} X \ mod \ p_1 &= a_1 \\ X \ mod \ p_2 &= a_2 \\ &\cdots \\ X \ mod \ p_n &= a_n \end{aligned}$$

For example, consider the system of simultaneous congruences, such as $X \ mod 3 = 2,\ X \ mod 7 = 3,\ and\ X \ mod \ 11 = 2$. Then, by Theorem 2, there exists X with $0 \leq X < 3 \times 7 \times 11$. In this example, X is 101. We can compute X by using the method in [1].

Let $L_1 -> L_2 -> ... -> L_n$ be a path to encode. Suppose that each location is associated with a different prime number, the nodes with the same location have the same prime number, and the prime number for location L_i is denoted by $Prime(L_i)$. Then, we define Element List Encoding Number for the path $L_1 -> L_2 -> ... -> L_n$ as $Prime(L_1) \times Prime(L_2) \times \cdots \times Prime(L_n)$. If Element List Encoding Number is given, we can know the locations that compose the path since Element List Encoding Number is uniquely factorized by the product of prime numbers that correspond to locations by Theorem 1. However, although we know the locations in the path by Element List Encoding Number, we cannot know the ordering between the locations. To encode the ordering information compactly and efficiently, we consider the system of simultaneous congruences.

$$\begin{aligned} X \ mod \ Prime(L_1) &= 1 \\ X \ mod \ Prime(L_2) &= 2 \\ &\cdots \\ X \ mod \ Prime(L_n) &= n \end{aligned}$$

$1, 2, \cdots, \ and\ n$ are the levels (i.e., orders) of the nodes that correspond to $L_1, L_2, \cdots, \ and\ L_n$, respectively. Since $Prime(L_1), Prime(L_2), \cdots, \ and\ Prime(L_n)$ are prime numbers, they are pairwise relatively prime numbers. Thus, by Theorem 2, there exists X with $0 \leq X < Prime(L_1) \times$

$Prime(L_2) \times \cdots \times Prime(L_n)$ which solves the system of the above simultaneous congruences. We call X Order Encoding Number. Given Order Encoding Number, we can know the order information for any location L_i in the path by computing $X \ mod \ Prime(L_i)$. Sometimes, $Prime(L_i)$ may be less than i. To prevent it, a prime number that is greater than the maximum path length is assigned to a location.

Our use of prime numbers is similar to that in [39]. However, [39] assigns different prime numbers to each element in an XML tree that can have millions of elements while our approach assigns different prime numbers to different locations in a tree for trace records that has, at most, a few hundred locations. Therefore, in a typical application, [39] generates extremely large prime numbers. Furthermore, [39] orders all the elements of an XML tree, while we order only the locations on a path whose length is much smaller than the tree for trace records.

Therefore, we can encode the path by using Element List Encoding Number and Order Encoding Number. Although a path condition has multiple ancestor-descendant relationships, we can find out whether a path satisfies the path condition in Figure 2 by checking some simple mathematical conditions. We will explain how we process tracking queries and path oriented queries efficiently by using Element List Encoding Number and Order Encoding Number in Section VII.

Example 2: Assume that in Figure 6, the prime numbers that correspond to $A, B, C, D, and \ E$ are 2, 3, 5, 7, and 11. Consider the path $A- > B- > C$. Element List Encoding Number for the path is $2 \times 3 \times 5 = 30$. To compute Order Encoding Number, we must compute X such that $X \ mod \ 2 = 1, X \ mod \ 3 = 2, \ and \ X \ mod \ 5 = 3$. By Theorem 2, there exists X with $0 \leq X < 30$. In this case, $X = 23$. Similarly, we encode the other four paths, and get Element List Encoding Numbers and Order Encoding Numbers for the paths.

Consider a path with cycles. In such a case, we cannot compute Order Encoding Number since the same prime numbers should return different orders. However, we can solve the problem with a simple renaming method that adds the sequence order to the location. For example, consider the path $A- > B- > C- > A- > B- > D- > A$. In the path, the location A appears in the first, fourth, and seventh positions. We can rename A in the first, fourth, and seventh positions to A1, A2, and A3, respectively, by the sequence order of A. In the same way, we can rename the path to $A1- > B1- > C1- > A2- > B2- > D1- > A3$. Then, since the nodes with the same location name are considered differently, we can compute Order Encoding Number. In data

sets that have cycles, a user can write a query with a sequence order such as $//A1//B1//A3$. If a user does not specify the sequence order, we translate the location name with the default sequence order (=1). For example, $//A//D$ is translated into $//A1//D1$.

Even though we can store the flow information for products effectively by using the path encoding scheme, we did not discuss the time information for products yet. To store the time information for products, we construct the time tree from trace records in which the node has the start time and end time as well as the location.

In the time tree, we say that two paths are the same if the flows for locations are the same as well as the time information (start time and end time) for locations is the same.

Figure 7 shows the time tree constructed from the trace records in Figure 5. The construction of the time tree is the same as that of the tree for the trace records except that the stay records with the same location become different nodes if their time information is different. See B[5,7] node and B[7,8] node in Figure 7. Although these two nodes have the same location, they are classified as different nodes since they have different time information.

To retrieve the time information efficiently, we store numbers with nodes in the time tree by using the region numbering scheme [40] which assigns a node two values, Start and End. Start and End are assigned consecutively during the depth-first search. The region numbering has the property that node A is the ancestor of node B if and only if A.Start$<$B.Start and B.End$<$A.End. In order to know the region numbers of the nodes associated with a tag, we attach the tag to the node that corresponds to the final location of the trace record of the tag. Thus, the region number that corresponds to the final node in the trace record of the tag is assigned to the tag. In order to get the time and location information for tag 6 in Figure 7, we see the region number that corresponds to the final node in the trace record of tag 6. For the final node C[7,8] for tag 6, its region number is [13,14]. Therefore, we retrieve nodes that satisfy Start$\leq$13 and End$\geq$14. Such nodes are A[2,3], E[4,6], and C[7,8]. Therefore, we can retrieve the time information for tag 6 efficiently. There are more sophisticated region numbering schemes, however, the incorporation of them is straightforward.

VI. Storing Trace Records in an RDBMS

We devise a relational schema to store RFID data based on the path encoding scheme and the region numbering scheme. The schema is shown in Figure 8. PATH_TABLE, TAG_TABLE,

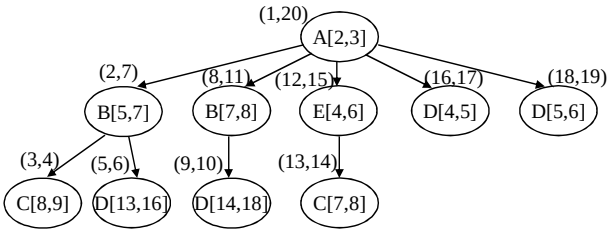


Fig. 7. Time Tree Structure for the Trace Records

and TIME_TABLE are related to the trace records and INFO_TABLE is related to the product information (e.g., product name, manufacturer, price).

PATH_TABLE

PATH_ID	ELEMENT_ENC_1	…	ELEMENT_ENC_m	ORDER_ENC_1	…	ORDER_ENC_m

TAG_TABLE

TAG_ID	PATH_ID	START	END	INFO_ID

TIME_TABLE

START	END	LOC	START_TIME	END_TIME

INFO_TABLE

INFO_ID	PRODUCT_NAME	MANUFACTURER	PRICE

Fig. 8. Relational Schema to Store RFID Data

PATH_TABLE stores the path information by using the path encoding scheme in Section V. In PATH_TABLE, the attributes ELEMENT_ENC_1, ELEMENT_ENC_2, $\cdots$, and ELEMENT_ENC_m correspond to Element List Encoding Number , while the attributes ORDER_ENC_1, ORDER_ENC_2, $\cdots$, and ORDER_ENC_m correspond to Order Encoding Number. If an RDBMS supports the data type whose maximum value is greater than Element List Encoding Number and Order Encoding Number of any trace record, PATH_TABLE consists of only ELEMENT_ENC_1 and ORDER_ENC_1 with PATH_ID.

TIME_TABLE stores the time information for trace records by using the region numbering scheme. In TIME_TABLE, START and END are Start and End in the region numbering scheme. LOC is the location. START_TIME and END_TIME correspond to the start time and end time in the time tree. TAG_TABLE has two identifiers for path and time information. PATH_ID is the identifier for the path information and (START, END) is the identifier for the time information. In addition, INFO_ID is the identifier for INFO_TABLE. INFO_TABLE stores the information of products. In this paper, we do not focus on INFO_TABLE.

If a path is long, its Element List Encoding Number and Order Encoding Number will be large and commercial RDBMSs may not support the storing of such large numbers. To solve this problem, we store the path in several attributes by dividing it.

The notations used in this paper are defined as follows:

- $Type$: the data type for Element List Encoding Number and Order Encoding Number which an RDBMS supports. (e.g., BIGINT, DECIMAL)
- Max_{Type}: the maximum value that the data type $Type$ can have
- m: the maximum number of path segments in a path

Consider the long path $L_1 -> L_2 -> \cdots -> L_n$. We can divide the long path into several path segments $s_1, \cdots, s_t$ such that

- $s_1 = L_1 -> L_2 -> \cdots -> L_{i_1}$,
  $s_2 = L_{i_1+1} -> L_{i_1+2} -> \cdots -> L_{i_2}$,
  $\cdots$,
  $s_t = L_{i_{t-1}+1} -> L_{i_{t-1}+2} -> \cdots -> L_{i_t}$
  $(1 \leq i_1 < i_2 < \cdots < i_t = n)$
- For $u \in 1, 2, \cdots, t$,
  $\Pi_{j=i_{u-1}+1}^{j=i_u} Prime(L_j) \leq Max_{Type}$, and $(\Pi_{j=i_{u-1}+1}^{j=i_u} Prime(L_j)) \times Prime(L_{i_u+1}) > Max_{Type}$, with $i_0 = 0$ and $Prime(L_{n+1}) = \infty$.

We can easily find the path segments for the long path $L_1 -> L_2 -> \cdots -> L_n$ which satisfy the above conditions. We do not consider Order Encoding Number when we divide the path since Order Encoding Number X for any path is less than Element Encoding Number E for the path by Theorem 2 (i.e., $0 \leq X < E$). However, elements in each path segment keep the order information in the original path. Example 3 shows how the path is split up.

Example 3: Assume that the path $A -> B -> C -> D -> E -> F$, $p_1 = Prime(A) = 2$, $p_2 = Prime(B) = 3$, $p_3 = Prime(C) = 5$, $p_4 = Prime(D) = 7$, $p_5 = Prime(E) = 11$, $p_6 = Prime(F) = 13$ and Max_{Type} = 250. Since $\Sigma_{j=1}^{j=6} p_j = 30030 > Max_{Type} = 250$, Element List Encoding Number for the path cannot be stored in one attribute with $Type$. We split up the path into path segments $s_1 : A -> B -> C -> D$, $s_2 : E -> F$ which satisfy the following inequalities. Note that the order of E in s_2 is not 1 but 5, and that of F is not 2 but 6.

$\Sigma_{j=1}^{j=4} p_j = 210 < Max_{Type}$ *and* $(\Sigma_{j=1}^{j=4} p_j) \times p_5 = 2310 > Max_{Type}$

$\Sigma_{j=5}^{j=6} p_j = 143 < Max_{Type}$ *and* $(\Sigma_{j=5}^{j=6} p_j) \times p_7 = \infty > Max_{Type}$

If the maximum number of path segments in a path is m, then PATH_TABLE is created, as shown in Figure 8, which has m ELEMENT_ENC attributes and m ORDER_ENC attributes. If some path is divided into t path segments and t is less than m, we fill ELEMENT_ENC_(t+1), ELEMENT_ENC_(t+2), $\cdots$, and ELEMENT_ENC_m with 1 and ORDER_ENC_(t+1), ORDER_ENC_(t+2), $\cdots$ and ORDER_ENC_m with 0.

```
Function store_trace_record(trace records tr)
begin
 1:  // m: the maximum number of path segments in a path
 2:  m := 0

 3:  for i=0; i<the number of trace records; i++
 4:  {
 5:    <path_id, store_flag>:=constructTree(tree, tr[i])
 6:    if store_flag == FALSE
 7:    {
 8:       <segment_list, segment_num> := divide_path(tr[i])
 9:       if segment_num > m
10:          m := segment_num
11:       store  segment_list for trace record tr[i] in PATH_FILE
                        using the path encoding scheme
12:    }
13:    store (tag identifier from tr[i], path_id)  in TEMP_PATH_FILE
14: }

15: for i=0; i<the number of trace records; i++
16:    constructTimeTree(time_tree, tr[i])
17: assign region numbers to nodes in time_tree

18: while traversing nodes in time_tree by the breath-first search
19: {
20:     store nodes of time_tree in TIME_FILE
21:     store (tag identifier, region numbers for node) in
            TEMP_TIME_FILE for all tags attached to node
22: }

23: create the schema according to m
24: perform bulk loading with PATH_FILE, TEMP_PATH_FILE,
                                TIME_FILE, and TEMP_TIME_FILE
25: after joining TEMP_PATH_TABLE and TEMP_TIME_TABLE
                                on TAG_ID, fill TAG_TABLE
end
```

Fig. 9. Algorithm to Store Trace Records using the Relational Schema

Figure 9 shows the algorithm to store trace records using the relational schema. As input for the algorithm, trace records are given. Trace records are translated into relational data. In the algorithm of Figure 9, we pile up the relational data in a text file format and then store it into an RDBMS together with bulk loading in order to load data efficiently. PATH_FILE is the file for PATH_TABLE, TEMP_PATH_FILE for TEMP_PATH_TABLE, TIME_FILE for TIME_TABLE, and TEMP_TIME_FILE for TEMP_TIME_TABLE. In Line 2, we initialize m, the maximum number of path segments in a path. To fill PATH_TABLE, we construct a tree

from the paths of the trace records *tr* by using *constructTree(Tree tree, TraceRecord tr)* (Line 5). In *constructTree(Tree tree, TraceRecord tr)*, if there is no path for *tr* in *tree*, it inserts a new path into *tree*, returns *path_id* of the new path and sets *store_flag* to *FALSE*. Otherwise, it returns *path_id* of the path and sets *store_flag* to *TRUE*.

If *store_flag* is *FALSE*, we insert Element List Encoding Number and Order Encoding Number for the path into PATH_FILE (Line 6-12). However, if the numbers are large and cannot be stored in one attribute, we divide the path using *divide_path(TraceRecord tr)*. *divide_path(TraceRecord tr)* divides one path into several path segments according to Max_{Type} and returns a path segment list (i.e., *segment_list*) and the number of path segments in the path (i.e., *segment_num*). Each path segment is stored in PATH_FILE by using the path encoding scheme with *path_id*. Also, for the creation of the schema, we keep the maximum number of path segments, m (Line 9-10). In Line 13, we insert *tag identifier* and *path_id* into TEMP_PATH_FILE. Note that TEMP_PATH_TABLE is used to fill TAG_TABLE later.

To fill TIME_TABLE, we construct the time tree from the trace records *tr* by using *constructTimeTree(TimeTree time_tree, TraceRecord tr)* (Line 16). In *constructTimeTree(TimeTree time_tree, TraceRecord tr)*, if there is no *tr* in *time_tree*, it inserts the new trace record into *time_tree*. After the construction of *time_tree*, we assign region numbers to the nodes in *time_tree* (Line 17). Then, we store (START, END, LOC, START_TIME, END_TIME) in TIME_FILE (Line 20) by traversing nodes in *time_tree* by using the breath-first search. If there are tags attached to a node, we store (tag identifier, region number for the node) in TEMP_TIME_FILE (Line 21). We create the schema according to the maximum number of path segments m. We then perform bulk loading with PATH_FILE, TEMP_PATH_FILE, TIME_FILE, and TEMP_TIME_FILE. Finally, to fill TAG_TABLE, we join TEMP_PATH_TABLE and TEMP_TIME_TABLE on TAG_ID (Line 25).

Figure 10 shows the status of tables after storing trace records in Figure 5 by the algorithm in Figure 9. In Figure 10, we assume that all Element List Encoding Numbers can be stored in one attribute since Max_{Type} is large enough. Since there are 5 different paths for trace records in Figure 5, Element List Encoding Numbers and Order Encoding Numbers of 5 paths are stored in PATH_TABLE. Note that the path identifiers for $A->B->C$, $A->B->D$, $A->E->C$, $A->E$, and $A->D$ are 1, 2, 3, 4, and 5, respectively. As shown in Figure 7, the time tree for trace records has 10 nodes. The information for 10 nodes is stored

in TIME_TABLE. Also, the path identifier and the time identifier for 10 tags are stored in TAG_TABLE.

PATH_TABLE

PATH_ID	ELEMENT_ ENC	ORDER_ ENC
1	30	23
2	42	17
3	110	13
4	22	13
5	14	9

TAG_TABLE

TAG_I D	PATH_ ID	START	END	INFO_ ID
tag1	1	3	4	
tag2	1	3	4	
tag3	1	3	4	
tag4	2	5	6	
tag5	2	9	10	
tag6	3	13	14	
tag7	3	13	14	
tag8	4	12	15	
tag9	5	16	17	
tag10	5	18	19	

TIME_TABLE

START	END	LOC	START_ TIME	END_ TIME
1	20	A	2	3
2	7	B	5	7
8	11	B	7	8
12	15	E	4	6
16	17	D	4	5
18	19	D	5	6
3	4	C	8	9
5	6	D	13	16
9	10	D	14	18
13	14	C	7	8

Fig. 10. Status of Tables after Storing Trace Records when m is 1

PATH_TABLE

PATH_ID	ELEMENT_ENC_1	ELEMENT_ENC_2	ORDER_ENC_1	ORDER_ENC_2
1	30	1	23	0
2	6	7	5	3
3	22	5	13	3
4	22	1	13	0
5	14	1	9	0

Fig. 11. Status of Tables after Storing Trace Records when m is 2

If we cannot store Element List Encoding Number in one attribute, we should split up the paths as mentioned above. Figure 11 shows the status of PATH_TABLE after storing trace records in Figure 5 when Max_{Type} is 30. In this case, m is 2. Also, since TAG_TABLE and TIME_TABLE are not affected by m, they are the same as those in Figure 10. Thus, we do not show them in Figure 11. In Figure 11, the path $A- > B- > D$ (*path_id* 2) and the path $A- > E- > C$ (*path_id* 3) are divided into two path segments. Therefore, those paths are stored with two ELEMENT_ENC attributes and two ORDER_ENC attributes. For example, $A- > B- > D$ has two path segments, $A- > B$ and D. $A- > B$ is encoded as ELEMENT_ENC_1 = 2×3 and ORDER_ENC_1 = 5 and D as ELEMENT_ENC_2 = 7 and ORDER_ENC_2 = 3. Other paths are stored with one ELEMENT_ENC attribute and one ORDER_ENC attribute and their remaining attributes are filled with 1 or 0.

VII. Query Translation

Based on the relational schema in Section VI, we provide a method to process tracking queries and path oriented queries. Since we store RFID data by using an RDBMS, we translate tracking queries and path oriented queries into SQL queries. Then, we can easily process the queries by executing the SQL queries. We will first deal with the translation when m is 1 in Section VII-A and then deal with the translation when m is greater than 1 in Section VII-B.

A. Basic Query Translation

In this section, we assume that all Element List Encoding Numbers can be stored in one attribute. That is, m is 1 and PATH_TABLE consists of 3 attributes: path_id, ELEMENT_ENC_1 and ORDER_ENC_1. Therefore, for convenience, we use ELEMENT_ENC and ORDER_ENC instead of ELEMENT_ENC_1 and ORDER_ENC_1, respectively.

1) Tracking Query: We can process a tracking query efficiently by using the relational schema in Section VI. To trace a tag, we get Element List Encoding Number and Order Encoding Number that correspond to the tag. To get them, we join PATH_TABLE and TAG_TABLE. Figure 12 shows the SQL query to get Element List Encoding Number and Order Encoding Number that correspond to the tag with TagID = my_tag_id.

```
SELECT P.ELEMENT_ENC, P.ORDER_ENC
FROM PATH_TABLE P, TAG_TABLE T
WHERE T.TAG_ID = my_tag_id AND T.PATH_ID = P.PATH_ID
```

Fig. 12. SQL Query for the Tracking Query

In order to know the locations in the flow of the tag, we factorize Element List Encoding Number. We then order the prime number factors $P_1, \cdots, P_n$ by computing *Order Encoding Number mod* P_i. We finally transform the prime number into the location name that corresponds to it.

Though the SQL query in Figure 12 has the join between PATH_TABLE and TAG_TABLE, it does not have too much time to execute the query since the query has the selection predicate for TAG_TABLE (T.TAG_ID = my_tag_id) and there is only one tuple that satisfies the predicate. Therefore, we can process tracking queries efficiently.

2) Path Oriented Retrieval Query: Although the path condition in a path oriented retrieval query has many ancestor-descendant relationships, we can easily find paths that satisfy the path condition by checking mathematical conditions. Therefore, we can process path oriented retrieval queries efficiently with our relational schema.

By Theorem 1, the path contains locations $L_1, L_2, \cdots, L_k$ if and only if *ELEMENT_ENC mod* $(Prime(L_1) \times Prime(L_2) \times \cdots \times Prime(L_k)) = 0$. Therefore, if the path condition contains locations $L_1, L_2, \cdots, L_k$, we insert the element membership condition *ELEMENT_ENC mod* $(Prime(L_1) \times Prime(L_2) \times \cdots \times Prime(L_k)) = 0$ into the where clause in the SQL query. To determine the ancestor-descendant relationship or the parent-child relationship, we use Order Encoding Number. Consider two adjacent steps in the path condition and let locations of the two steps be L_a and L_b. If L_a and L_b are in the ancestor-descendant relationship (i.e., $L_a//L_b$), we insert *ELEMENT_ENC mod* $Prime(L_a) <$ *ELEMENT_ENC mod* $Prime(L_b)$ into the where clause in the SQL query. If L_a and L_b are in the parent-child relationship (i.e., L_a/L_b), we insert *ELEMENT_ENC mod* $Prime(L_a) + 1$=*ELEMENT_ENC mod* $Prime(L_b)$ into the where clause.

After finding the paths that satisfy the path condition, we join PATH_TABLE and TAG_TABLE on TAG_ID to get tags. Figure 13 shows the SQL query that corresponds to the path oriented retrieval query <//A//B/C>. In Figures 13 through 18, pA and pB and pC denote $Prime(A)$, $Prime(B)$, and $Prime(C)$, respectively.

```
SELECT T.TAG_ID
FROM PATH_TABLE P, TAG_TABLE T
WHERE MOD(P. ELEMENT_ENC, pA*pB*pC) = 0 AND
MOD(P.ORDER_ENC, pA) < MOD(P.ORDER_ENC, pB) AND
MOD(P.ORDER_ENC, pB) + 1 = MOD(P.ORDER_ENC, pC)
AND P.PATH_ID = T.PATH_ID
```

Fig. 13. SQL Query for <//A//B/C>

Path oriented retrieval queries may have time conditions. If the queries have time conditions, we join TAG_TABLE and TIME_TABLE. We can retrieve the time information efficiently by using the property of the region numbering scheme. We insert the following statement into the where clause in the SQL query for time conditions.

TIME_TABLE.LOC='location name' AND TIME_TABLE.START≤TAG_TABLE.START AND

TAG_TABLE.END≤TIME_TABLE.END AND Time Conditions in the Step

Figure 14 shows the SQL query that corresponds to the path oriented retrieval query <//A//B[(EndTime-StartTime)<10]/C>.

```
SELECT T.TAG_ID
FROM PATH_TABLE P, TAG_TABLE T , TIME_TABLE S
WHERE MOD(P. ELEMENT_ENC, pA*pB*pC) = 0 AND
MOD(P.ORDER_ENC, pA) < MOD(P.ORDER_ENC, pB)  AND
MOD(P.ORDER_ENC, pB) + 1 = MOD(P.ORDER_ENC, pC)
AND P.PATH_ID = T.PATH_ID AND S.LOC = 'B' AND
S.START <= T.START AND T.END <= S.END AND
(S.END_TIME – S.START_TIME) < 10
```

Fig. 14. SQL Query for <//A//B[(EndTime-StartTime)<10]/C>

Path oriented retrieval queries may also have product information conditions, such as PRODUCT_NAME = 'laptop'. To process such queries, we first perform the selection of INFO_TABLE for product information conditions. We then join INFO_TABLE and TAG_TABLE on INFO_ID. Figure 15 shows the SQL query that corresponds to the path oriented retrieval query <//A//B/C, PRODUCT_NAME = 'laptop'>

```
SELECT T.TAG_ID
FROM PATH_TABLE P, TAG_TABLE T, INFO_TABLE I
WHERE MOD(P.ELEMENT_ENC, pA*pB*pC) = 0 AND
MOD(P.ORDER_ENC, pA) < MOD(P.ORDER_ENC, pB) AND
MOD(P.ORDER_ENC, pB) + 1 = MOD(P.ORDER_ENC, pC)
AND P.PATH_ID = T.PATH_ID AND
I.PRODUCT_NAME = 'laptop' AND I.INFO_ID = T.INFO_ID
```

Fig. 15. SQL Query for <//A//B/C, PRODUCT_NAME = 'laptop'>

3) Path Oriented Aggregate Query: Since path oriented aggregate queries have aggregate functions, we add an aggregate function in the select clause of the SQL query. Figure 16 shows the SQL query that corresponds to the path oriented aggregate query <COUNT(), //A//B/C>.

```
SELECT COUNT(*)
FROM PATH_TABLE P, TAG_TABLE T
WHERE MOD(P.ELEMENT_ENC, pA*pB*pC) = 0 AND
MOD(P.ORDER_ENC, pA) < MOD(P.ORDER_ENC, pB) AND
MOD(P.ORDER_ENC, pB) + 1 = MOD(P.ORDER_ENC, pC)
AND P.PATH_ID = T.PATH_ID
```

Fig. 16. SQL Query for <COUNT(), //A//B/C>

In the case of aggregate functions that need time attributes as arguments, we join TAG_TABLE and TIME_TABLE to get the time attributes since PATH_TABLE does not have the time information. Figure 17 shows the SQL query that corresponds to the path oriented aggregate query <AVG(B.StartTime),//A//B/C>.

```
SELECT AVG(S.START_TIME)
FROM PATH_TABLE P, TAG_TABLE T , TIME_TABLE S
WHERE MOD(P.ELEMENT_ENC, pA*pB*pC) = 0 AND
MOD(P.ORDER_ENC, pA) < MOD(P.ORDER_ENC, pB) AND
MOD(P.ORDER_ENC, pB) + 1 = MOD(P.ORDER_ENC, pC)
AND P.PATH_ID = T.PATH_ID AND S.LOC = 'B' AND
S.START <= T.START AND T.END <= S.END
```

Fig. 17. SQL Query for <AVG(B.StartTime), //A//B/C>

Consider the query <AVG(C.EndTime-A.StartTime),//A//B/C>. The query has the aggregate function that has two time attributes as the argument. In this case, we join one TAG_TABLE and two TIME_TABLEs. Figure 18 shows the SQL query that corresponds to the path oriented aggregate query <AVG(C.EndTime-A.StartTime), //A//B/C>.

```
SELECT AVG(S2.END_TIME-S1.START_TIME)
FROM PATH_TABLE P, TAG_TABLE T, TIME_TABLE S1, TIME_TABLE S2
WHERE MOD(P.ELEMENT_ENC, pA*pB*pC) = 0 AND
MOD(P.ORDER_ENC, pA) < MOD(P.ORDER_ENC, pB) AND
MOD(P.ORDER_ENC, pB) + 1 = MOD(P.ORDER_ENC, pC) AND
P.PATH_ID = T.PATH_ID AND S1.LOC = 'A' AND
S1.START <= T.START AND T.END <= S1.END AND S2.LOC = 'C' AND
S2.START <= T.START AND T.END <= S2.END
```

Fig. 18. SQL Query for <AVG(C.EndTime - A.StartTime), //A//B/C>

B. Advanced Query Translation

In this section, we assume that Element List Encoding Number cannot be stored in one attribute. Although we can store Element List Encoding Number for a long path in an RDBMS by dividing the path, it is difficult to find the paths that satisfy the path condition using the SQL query due to the computation overflow. For example, to find some path from PATH_TABLE in Figure 11, we should multiply ELEMENT_ENC_1 by ELEMENT_ENC_2. However, since Max_{Type} is 30, the overflow will occur in an RDBMS during the multiplication. The SQL query may not run or the result may be wrong. Therefore, we need to extend the translation for path conditions when m is greater than 1.

When a path condition contains locations $L_1, L_2, \cdots, L_k$, we should insert the element membership condition *(ELEMENT_ENC_1* $\times$ *ELEMENT_ENC_2* $\times \cdots \times$ *ELEMENT_ENC_*m*) mod* $(Prime(L_1) \times Prime(L_2) \times \cdots \times Prime(L_k)) = 0$ into the where clause in the SQL query in order to check whether the path contains locations $L_1, L_2, \cdots, L_k$. To avoid the computation overflow during the execution of the condition, we change the condition into another equivalent condition by using Lemma 1 and 2.

Lemma 1: Let E_1, E_2, and P be any natural numbers. Then,

$$(E_1 E_2)\ mod\ P = (E_1\ mod\ P)(E_2\ mod\ P)\ mod\ P$$

Proof: This is easily derived from the basic number theory. ■

Lemma 2: Let E, P_1, and P_2 be any natural numbers and P_1 and P_2 be relatively prime numbers. Then,

$E\ mod\ (P_1 P_2) = 0$ if and only if $E\ mod\ P_1 = 0$ and $E\ mod\ P_2 = 0$

Proof: This is easily derived from the basic number theory. ■

Suppose that the path condition in the query contains $L_1, L_2, \cdots, L_k$. Let p_i be $Prime(L_i)$, P be $p_1 p_2 \cdots p_k$ and E_i be ELEMENT_ENC_i. To know whether the path contains locations $L_1, L_2, \cdots, L_k$, we should check the condition $(E_1 E_2 \cdots E_m)\ mod\ P = 0$. To avoid the overflow during the computation of the condition, we change it into another equivalent condition using Lemma 1, which can be computed without an overflow. Consider the simple case that m is 2. By Lemma 1, $(E_1 E_2)\ mod\ P$ is equal to $(E_1\ mod\ P)(E_2\ mod\ P)\ mod\ P$. Therefore, we check $(E_1\ mod\ P)(E_2\ mod\ P)\ mod\ P = 0$ instead of $(E_1 E_2)\ mod\ P = 0$. If $P^2 \leq Max_{Type}$, the computation of $(E_1\ mod\ P)(E_2\ mod\ P)\ mod\ P = 0$ is performed without an overflow since $A\ mod\ P < P$, for any natural number A.

If m is greater than 2, we apply Lemma 1 recursively. The detailed algorithm is shown in Figure 19. However, the translation using Lemma 1 is limited since $P^2 \leq Max_{Type}$ should be satisfied. If $P^2 \leq Max_{Type}$ is not satisfied, we can apply Lemma 2 before using Lemma 1. To apply Lemma 2, we first factorize P into $P_1, P_2, \cdots, P_t$ such that $P = P_1 P_2 \cdots P_t$ and $P_i^2 \leq Max_{Type}$ for $i = 1, 2, \cdots, t$. If P is factorized into P_1 and P_2, we can change the condition $(E_1 E_2)\ mod\ P = 0$ into two conditions $(E_1 E_2)\ mod\ P_1 = 0$ and $(E_1 E_2)\ mod\ P_2 = 0$ by using Lemma 2. Also, by using Lemma 1, we can change $(E_1 E_2)\ mod\ P_1 = 0$ and $(E_1 E_2)\ mod\ P_2 = 0$ into $(E_1\ mod\ P_1)(E_2\ mod\ P_1)\ mod\ P_1 = 0$ and $(E_1\ mod\ P_2)(E_2\ mod\ P_2)\ mod\ P_2 = 0$. An RDBMS

```
Function element_translation(element list L1, L2, …, Lk in the path condition)
begin
 1:  product := 1
 2:  condition:= null
 3:  for i:=1; i<=k; i++
 4:  {
 5:    if  (product * Prime(Li))2 > MaxType
 6:    {
 7:      if condition == null
 8:        // m: the maximum number of path segments in a path
 9:        condition := sub_element_translation(product, m) + " = 0 "
10:     else
11:     {
12:       // m: the maximum number of path segments in a path
13:       condition :=  condition + " and  " + sub_element_transation(product, m) + " = 0 "
14:     }
15:     product := 1
16:   }
17:   product := product * Prime(Li)
18: }
19: if condition == null
20:   condition := sub_element_translation(product, m) + " = 0 "
21: else
22: {
23:    condition := condition + " and " + sub_element_translation(product, m) + " = 0 "
24: }
25: return condition
end

Function sub_element_transation(product p, index i)
begin
 1:  if  i==1
 2:    return "ELEMENT_ENC_" + i + " mod " + p
 3:  else
 4:    return "(" + sub_element_translation(p, i-1) + ")(ELEMENT_ENC_" + i +
            " mod " + p + ") mod " + p
end
```

Fig. 19. Algorithm to Translate the Element Membership Condition into an SQL Query

can evaluate $(E_1 \ mod \ P_1)(E_2 \ mod \ P_1) \ mod \ P_1 = 0$ and $(E_1 \ mod \ P_2)(E_2 \ mod \ P_2) \ mod \ P_2 = 0$ without a computation overflow.

The algorithm in Figure 19 summarizes the above process for checking whether the path contains locations $L_1, L_2, \cdots, L_k$. As input for the algorithm *element_translation()*, the element list $L_1, L_2, \cdots, L_k$ in the path condition is given. In Line 1-2, *product* and *condition* are initialized. If the square of the product of prime numbers for the elements in the path is greater than Max_{Type}, we call *sub_element_translation(product, m)* and reinitialize *product*. *sub_element_translation(product, m)* translates $(E_1E_2E_3 \cdots E_m) \ mod \ P$ into the equivalent condition, which does not cause the computation overflow.

To determine the ancestor-descendant relationship or the parent-child relationship, we use Order Encoding Number. When m is 1, we compute *ORDER_ENC_1 mod* p_j for the order of location L_j. However, when m is greater than 1, the order information of each location in a path

```
SELECT T.TAG_ID
FROM PATH_TABLE P, TAG_TABLE T
WHERE
MOD(MOD(P.ELEMENT_ENC_1, pA*pB*pC) *
      MOD(P.ELEMENT_ENC_2, pA*pB*pC), pA*pB*pC) = 0 AND

MOD(P.ORDER_ENC_1, pA) * FLOOR(1 – (P.ELEMENT_ENC_1 mod pA)/ pA) +
    MOD(P.ORDER_ENC_2, pA) * FLOOR(1 – (P.ELEMENT_ENC_2 mod pA)/ pA)
    < MOD(P.ORDER_ENC_1, pB) * FLOOR(1 – (P.ELEMENT_ENC_1 mod pB)/ pB) +
    MOD(P.ORDER_ENC_2, pB) * FLOOR(1 – (P.ELEMENT_ENC_2 mod pB)/ pB) AND

MOD(P.ORDER_ENC_1, pB) * FLOOR(1 – (P.ELEMENT_ENC_1 mod pB)/ pB) +
    MOD(P.ORDER_ENC_2, pB) * FLOOR(1 – (P.ELEMENT_ENC_2 mod pB)/ pB) + 1 =
    MOD(P.ORDER_ENC_1, pC) * FLOOR(1 – (P.ELEMENT_ENC_1 mod pC)/ pC) +
    MOD(P.ORDER_ENC_2, pC) * FLOOR(1 – (P.ELEMENT_ENC_2 mod pC)/ pC)
```

Fig. 20. SQL Query for <//A//B/C> when m is 2

is separately stored in attributes ORDER_ENC_1, ORDER_ENC_2, $\cdots$, and ORDER_ENC_m. We use Theorem 3 to compute the order when m is greater than 1.

Theorem 3: Let O be Order Encoding Number in an original path before dividing the path and O_i be ORDER_ENC_i that corresponds to the i path segment after dividing the path. For each location L_j in the path, the following equation is satisfied.

$$O \ mod \ p_j = \Sigma_{i=1}^{m}(O_i \ mod \ p_j)\llcorner 1 - \frac{E_i \ mod \ p_j}{p_j}\lrcorner$$

Proof: If L_j is contained in the i path segment, $(O_i \ mod \ p_j)\llcorner 1 - \frac{E_i \ mod \ p_j}{p_j}\lrcorner = (O_i \ mod \ p_j) = (O \ mod \ p_j)$ since $E_i \ mod \ p_j = 0$. Otherwise, $(O_i \ mod \ p_j)\llcorner 1 - \frac{E_i \ mod \ p_j}{p_j}\lrcorner = 0$ since $E_i \ mod \ p_j \neq 0$ and $E_i \ mod \ p_j < p_j$. Therefore, $\Sigma_{j=1}^{m}(O_i \ mod \ p_j)\llcorner 1 - \frac{E_i \ mod \ p_j}{p_j}\lrcorner$ computes the order information of location L_j in m ORDER_ENC attributes. That is, $O \ mod \ p_j = \Sigma_{i=1}^{m}(O_i \ mod \ p_j)\llcorner 1 - \frac{E_i \ mod \ p_j}{p_j}\lrcorner$ ■

If we use $\Sigma_{i=1}^{m}(O_i \ mod \ p_j)\llcorner 1 - \frac{E_i \ mod \ p_j}{p_j}\lrcorner$ in Theorem 3, we can translate the order condition into an SQL query. We use the *FLOOR* operator in SQL queries for $\llcorner\lrcorner$.

Figure 20 shows the SQL query that corresponds to the path oriented retrieval query <//A//B/C> when m is 2. We omit other translation examples since the translation, except for the path information, is similar to that of Section VII-A.

VIII. EXPERIMENTS

In order to validate our approach, we conduct experimental evaluations for various queries.

A. Experimental Environment

We experiment on a Pentium 2.53GHz with 1GB main memory using Java. Since there is no well known RFID data set, we generate synthetic data based on a real-life example and formulate 15 queries (1 tracking query, 4 path oriented retrieval queries, 10 path oriented aggregate queries). The query performance is measured by processing the queries 3 times and averaging the execution time. As a comparison system, RFID-Cuboid [14] is used. For fairness, we implement RFID-Cuboid on an RDBMS to support tracking queries and path oriented queries.

1) Data Set: Since we use stay records instead of raw RFID data, we generate stay records. To reflect real environments, we generate data sets based on a real-life example in Figure 21. Figure 21 shows the food distribution [9] in the United States. In the food distribution, the minimum length of paths is 3 and the maximum length is 9. The products are distributed from import markets and agricultural input suppliers. Since it is difficult to find detailed distribution information of each product from a real-life example, we assume that the products are distributed equally when they move to the next location if there are many next locations.

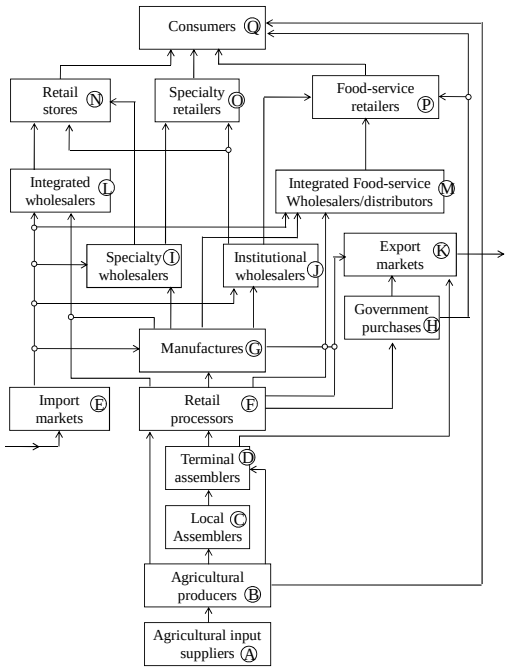


Fig. 21. Food Distribution in the United States [9]

Initially, products are generated at starting nodes such as import markets and agricultural input

suppliers. Then, they move to the next location in groups according to the links of Figure 21. As products move together in groups in many RFID applications, we generate stay records with the grouping factor, which is the number of data generated at a starting node simultaneously. We consider two kinds of data in data generation: GData (grouping factor: 10000) and IData (grouping factor: 1000). Although we set the grouping factor of GData to 10000 and that of IData to 1000, the values (i.e., 10000, 1000) are not important. The key point in generating GData and IData is that products in GData move in larger groups compared to those in IData. By generating GData and IData, we analyze how the query performance is affected by the grouping factor. We set the number of instances for each location to 2 since we consider all possible paths and the number of paths increases exponentially.

6×10^6, 12×10^6, 18×10^6, 24×10^6, and 30×10^6 stay records are generated for both GData and IData. The DECIMAL type in an RDBMS is used to store Element List Encoding Number and Order Encoding Number.

2) Query Set: 15 queries are formulated to test various features. Query 1 is a tracking query, Query 2-5 are path oriented retrieval queries, and Query 6-15 are path oriented aggregate queries. The queries are shown in Figure 22. Note that we represent the location name in queries as the character in the circle in Figure 21 and the instance identifier since the real names are long. Query 1 tests the performance of the tracking query. Query 2, 6, 12, and 15 evaluate the performance of the path condition with a single location while other queries with multiple locations. In particular, Query 5 and 9 have the time condition.

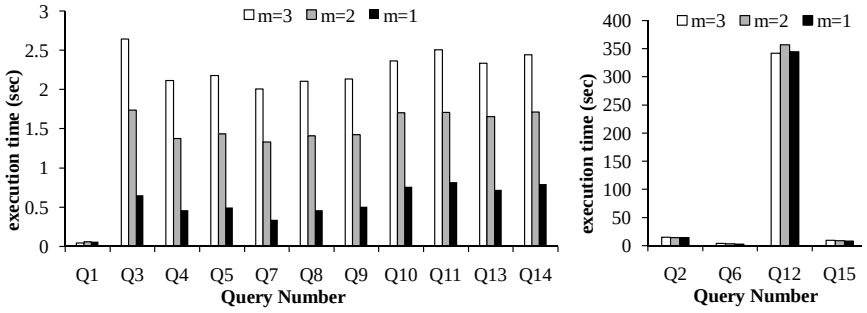
Query Number	Query	Query Number	Query
Query 1	TagID = 1	Query 9	<COUNT(), //D0//F0[StartTime<2000]//I0//O0>
Query 2	<//D0>	Query 10	<AVG(D0.EndTime-D0.StartTime), //D0/F0//I0/O0>
Query 3	</A0/B0/C0/D0/F0>	Query 11	<AVG(D0.EndTime-D0.StartTime), //D0//F0//I0//O0>
Query 4	<//D0//F0//I0//O0>	Query 12	<AVG(D0.StartTime), //D0>
Query 5	<//D0//F0[StartTime<2000]//I0//O0>	Query 13	<MIN(D0.EndTime-D0.StartTime), //D0/F0//I0/O0>
Query 6	<COUNT(), //D0>	Query 14	<MIN(D0.EndTime-D0.StartTime), //D0//F0//I0//O0>
Query 7	<COUNT(), //D0/F0//I0/O0>	Query 15	<MIN(D0.StartTime), //D0>
Query 8	<COUNT(), //D0//F0//I0//O0>		

Fig. 22. Query Set

B. Experimental Results

In experimental results, if the query is not finished within 2 hours, we denote it by ”X” in the graph.

Figure 23 shows the query performance in IData with 3×10^7 as m, the maximum number of path segments in a path, changes. Figure 23-(a) shows the query performance when the execution time is less than 3 seconds and Figure 23-(b) shows the performance when the time is more than 3 seconds. As m increases, the computation overhead becomes generally large since we need the additional computation for segmented paths. However, as can be seen in Figure 23-(b), the performance sometimes is not affected much by m. This is partly because the RDBMS does not properly optimize the translated SQL query as the query is quite complex. While a lower value of m is favorable for query performance, a higher value must be used to avoid the overflow of the DECIMAL type. This is especially necessary for RDBMSs that do not provide a large DECIMAL type. Therefore, we set m to 2 for the following experiments.



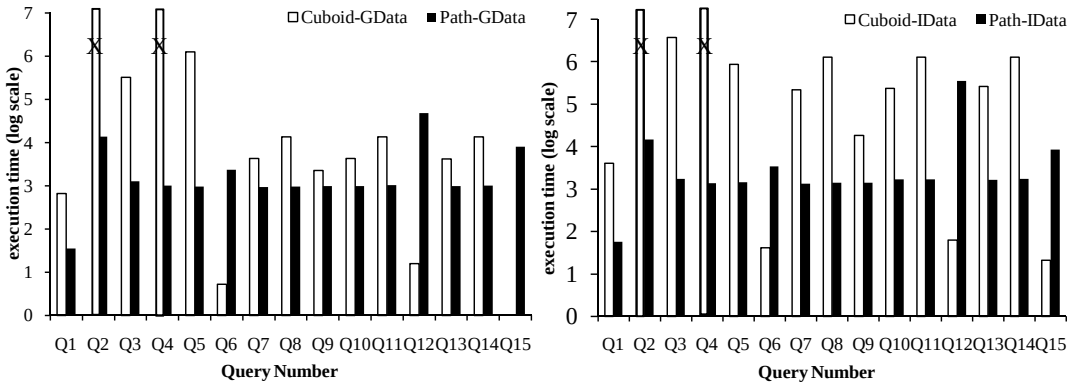
(a) When the execution time is less than 3 seconds (b) When the execution time is more than 3 seconds

Fig. 23. Experiment with respect to m

Figure 24 shows the query performance for 15 queries. Figure 24-(a) is the performance in GData with 3×10^7 tuples and Figure 24-(b) is the performance in IData with 3×10^7 tuples. Path denotes our approach and Cuboid RFID-Cuboid in the figures of this section. Since the performance gap between our approach and RFID-Cuboid is wide, we use the logarithmic scale (base 10) for the execution time in Figure 24.

In GData and IData, our approach is better than RFID-Cuboid for most queries in terms of the execution time (except Query 6, 12 and 15). We can also observe that the performance gap between our approach and RFID-Cuboid in IData is larger than that in GData. Since the shapes

of the graphs for the execution time for different sizes are similar, the execution time for only GData with 3×10^7 and IData with 3×10^7 is shown in Figure 24.



* In Cuboid-GData, the execution time for Query 15 is 0 in terms of microseconds. Therefore, we cannot represent it with the logarithmic scale.

(a) GData (b) IData

Fig. 24. Execution Time for 15 Queries

Figure 25, 26, and 27 show the query performance according to the number of stay records. Figure 25-(a) shows the performance for a tracking query. For the tracking query, our approach is faster than RFID-Cuboid in both GData and IData. We can also observe that our approach is much faster than RFID-Cuboid in IData. While our approach finds only Element List Encoding Number and Order Encoding Number for the given tag identifier, RFID-Cuboid scans STAY_TABLE (the table for stay records). Therefore, in IData, RFID-Cuboid has much worse performance than our approach compared to GData since the number of tuples of STAY_TABLE in IData is more than that in GData.

The performance of Query 2, 3, and 4 is shown in Figure 25-(b), (c), and (d). However, most cases of RFID-Cuboid do not finish within 2 hours. They are denoted by X in a rectangle. For example, ■ : X in a rectangle on the horizontal axis 24 means that the ■ approach does not finish within 2 hours when the number of tuples is 24×10^6. These queries are path oriented retrieval queries. To process path oriented retrieval queries, RFID-Cuboid joins STAY_TABLE and MAP_TABLE. MAP_TABLE contains the mapping from GID to TAG_ID in RFID-Cuboid. Since RFID-Cuboid uses the prefix encoding scheme, it needs string comparisons to process the join between STAY_TABLE and MAP_TABLE. Therefore, for path oriented retrieval queries, our approach has a much better performance than RFID-Cuboid.

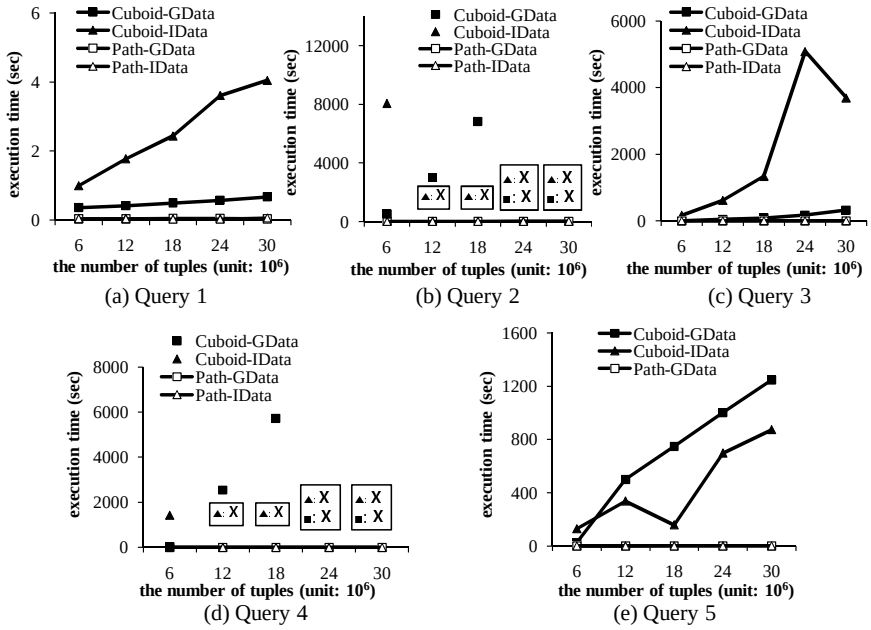


Fig. 25. Execution Time for Query 1, 2, 3, 4, and 5

The performance of Query 5 is shown in Figure 25-(e). Although RFID-Cuboid in GData shows a much better performance than that in IData for the cases of Query 2, 3, and 4, the performance gap between GData and IData in RFID-Cuboid does not have a big difference in Figure 25-(e). On the contrary, the performance in IData is better than that in GData. Query 5 has the time condition (StartTime$<$2000). Therefore, in IData, many tuples of STAY_TABLE are removed by the condition and tuples to join are reduced significantly. Also, the number of results in IData is less than that in GData. Therefore, the performance gap between IData and GData for RFID-Cuboid is small for Query 5.

In Query 6, 12 and 15 (Figure 26-(a), Figure 27-(b) and Figure 27-(e)), RFID-Cuboid is better than our approach. The path condition in Query 6, 12, and 15 has only one location. Since RFID-Cuboid focuses on groups in which products move together, RFID-Cuboid is efficient in the case of getting information at one location. However, in supply chain management, it is important to analyze the object transition. For queries related to the object transition (Query 3, 4, 5, 7, 8, 9, 10, 11, 13, and 14), our approach is superior to RFID-Cuboid. However, for Query 2, our approach has a better performance than RFID-Cuboid, although Query 2 is used to get information at one location. This is because RFID-Cuboid uses the string comparison to get tags.

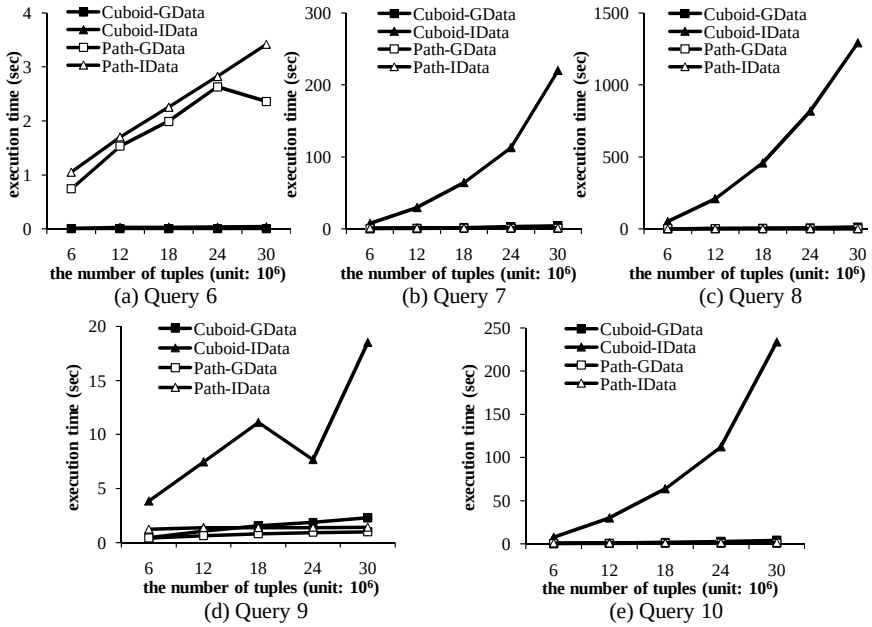


Fig. 26. Execution Time for Query 6, 7, 8, 9, and 10

Consider the query performance of Query 6, 12, and 15 (Figure 26-(a), Figure 27-(b) and Figure 27-(e)) versus that of Query 7, 8, 10, 11, 13, and 14 (Figure 26-(b), Figure 26-(c), Figure 26-(e), Figure 27-(a), Figure 27-(c) and Figure 27-(d)). The path condition in Query 6, 12 and 15 has only one location while the path condition in Query 7, 8, 10, 11, 13, and 14 has multiple locations. Since our approach uses Element List Encoding Number and Order Encoding Number, our approach can easily find paths, although the path condition has many ancestor-descendant relationships. Therefore, in Query 7, 8, 10, 11, 13, and 14, our approach is better than RFID-Cuboid.

Since Query 9 has the time information, our approach joins TAG_TABLE and TIME_TABLE. Therefore, our approach has a little better performance than RFID-Cuboid for GData, as shown in Figure 26-(d).

Consequently, our approach is superior to RFID-Cuboid in most cases. Also, our approach is less sensitive than RFID-Cuboid for the grouping factor. Therefore, our approach can be used in a wide range of applications. For the path oriented aggregate query whose path condition has only one location, our approach may be worse than RFID-Cuboid. However, since it is important to analyze the object transition in supply chain management, our approach will be more useful.

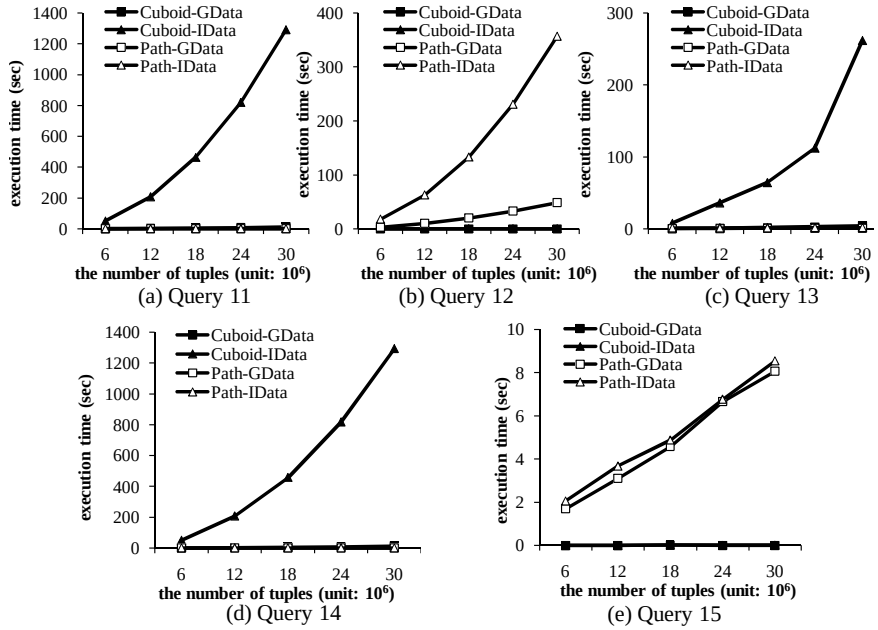


Fig. 27. Execution Time for Query 11, 12, 13, 14, and 15

IX. Conclusion

We expect that RFID technology will revolutionize supply chain management. In supply chain management, a large amount of RFID data is generated. However, since RFID data has flow information that is different from the traditional data, it is difficult to store data and process queries. Therefore, we propose an efficient storage scheme and efficient query processing for supply chain management.

The proposed approach in this paper is very efficient in tracking and analyzing the flow of products. We believe that it is useful for real-life logistic applications after conducting experiments with data sets based on a real-life logistic example. In addition, we devise a method that divides a path. Therefore, our approach can be implemented on an RDBMS easily, even if long paths exist in some logistic applications and we cannot store a large number in one attribute. Therefore, our approach is efficient and practical. Its application is not limited to supply chain management nor logistic applications. It will be useful for many applications that need flow-based processing. However, since we did not consider a distributed environment, all RFID data should be collected in the central server. Data collection in the central server may be difficult in some environments. Therefore, in the future, we plan to adapt and develop our approach in

a distributed environment.

ACKNOWLEDGMENTS

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (grant number 2009-0081365).

REFERENCES

[1] Chinese remainder theorem. http://en.wikipedia.org/wiki/Chinese_remainder_theorem.

[2] Epc global. http://www.epcglobalinc.org/home.

[3] Xpath. http://www.w3.org/TR/xpath.

[4] R. Agrawal, A. Cheung, K. Kailing, and S. Schönauer. Towards traceability across sovereign, distributed rfid databases. In *IDEAS*, 2006.

[5] Y. Bai, F. Wang, and P. Liu. Efficiently Filtering RFID Data Streams. In *VLDB Workshop on Clean Databases*, 2006.

[6] Y. Bai, F. Wang, P. Liu, C. Zaniolo, and S. Liu. Rfid data processing with a data stream query language. In *ICDE*, 2007.

[7] C. Ban, B. Hong, and D. Kim. Time Parameterized Interval R-Tree for Tracing Tags in RFID Systems. In *DEXA*, pages 503–513, 2005.

[8] C. Bornhövd, T. Lin, S. Haller, and J. Schaper. Integrating Automatic Data Acquisition with Business Processes - Experiences with SAP's Auto-ID Infrastructure. In *VLDB*, pages 1182–1188, 2004.

[9] D. J. Bowersox and D. J. Closs. *Logistical Management*. McGraw-Hill, 1996.

[10] P. Buneman, M. Grohe, and C. Koch. Path Queries on Compressed XML. In *VLDB*, pages 141–152, 2003.

[11] S. S. Chawathe, V. Krishnamurthy, S. Ramachandran, and S. Sarma. Managing RFID data. In *VLDB*, pages 1189–1195, 2004.

[12] L. Forlizzi, R. H. Güting, E. Nardelli, and M. Schneider. A Data Model and Data Structures for Moving Objects Databases. In *SIGMOD*, pages 319–330, 2000.

[13] H. Gonzalez, J. Han, and X. Li. FlowCube: Constructuing RFID FlowCubes for Multi-Dimensional Analysis of Commodity Flows. In *VLDB*, pages 834–845, 2006.

[14] H. Gonzalez, J. Han, X. Li, and D. Klabjan. Warehousing and Analyzing Massive RFID Data Sets. In *ICDE*, 2006.

[15] R. H. Güting, M. H. Böhlen, M. Erwig, C. S. Jensen, N. A. Lorentzos, M. Schneider, and M. Vazirgiannis. A Foundation for Representing and Querying Moving Objects. *ACM Transactions on Database Systems (TODS)*, 25(1):1–42, 2000.

[16] H. Haidarian-Shahri, G. Namata, S. Navlakha, A. Deshpande, and N. Roussopoulos. A Graph-based Approach to Vehicle Tracking in Traffic Camera Video Streams. In *Workshop on Data Management for Sensor Networks, in conjunction with VLDB (DMSN)*, pages 19–24, 2007.

[17] J. E. Hoag and C. W. Thompson. Architecting rfid middleware. *IEEE Internet Computing*, 10(5):88–92, 2006.

[18] Y. Hu, S. Sundara, T. Chorma, and J. Srinivasan. Supporting RFID-based Item Tracking Applications in Oracle DBMS Using a Bitmap Datatype. In *VLDB*, pages 1140–1151, 2005.

[19] S. R. Jeffery, M. N. Garofalakis, and M. J. Franklin. Adaptive Cleaning for RFID Data Streams. In *VLDB*, pages 163–174, 2006.

[20] N. K. Kanhere and S. T. Birchfield. Real-Time Incremental Segmentation and Tracking of Vehicles at Low Camera Angles Using Stable Features. *IEEE Transactions on Intelligent Transportation Systems (TITS)*, 9(1):148–160, 2008.

[21] Z. Kim and J. Malik. Fast Vehicle Detection with Probabilistic Feature Grouping and its Application to Vehicle Tracking. In *International Conference on Computer Vision (ICCV)*, pages 524–531, 2003.

[22] C.-H. Lee and C.-W. Chung. Efficient storage scheme and query processing for supply chain management using rfid. In *SIGMOD*, 2008.

[23] Q. Li and B. Moon. Indexing and querying xml data for regular path expressions. In *VLDB*, 2001.

[24] W. Lian, N. Mamoulis, D. W.-L. Cheung, and S.-M. Yiu. Indexing Useful Structural Patterns for XML Query Processing. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 17(7):997–1009, 2005.

[25] J.-K. Min, M.-J. Park, and C.-W. Chung. Xpress: A queriable compression for xml data. In *SIGMOD*, 2003.

[26] M. F. Mokbel, X. Xiong, and W. G. Aref. SINA: Scalable Incremental Processing of Continuous Queries in Spatio-temporal Databases. In *SIGMOD*, pages 623–634, 2004.

[27] D. Pfoser, C. S. Jensen, and Y. Theodoridis. Novel Approaches to the Indexing of Moving Object Trajectories. In *VLDB*, pages 395–406, 2000.

[28] K. Porkaew, I. Lazaridis, and S. Mehrotra. Querying Mobile Objects in Spatio-Temporal Databases. In *SSTD*, pages 59–78, 2001.

[29] J. Rao, S. Doraiswamy, H. Thakkar, and L. S. Colby. A Deferred Cleansing Method for RFID Data Analytics. In *VLDB*, pages 175–186, 2006.

[30] P. Rao and B. Moon. Prix: Indexing and querying xml using prufer sequences. In *ICDE*, 2004.

[31] K. H. Rosen. *Discrete Mathematics and Its Applications*. McGraw-Hill, 2003.

[32] Y. Tao and D. Papadias. MV3R-Tree: A Spatio-Temporal Access Method for Timestamp and Interval Queries. In *VLDB*, pages 431–440, 2001.

[33] I. Tatarinov, S. Viglas, K. S. Beyer, J. Shanmugasundaram, E. J. Shekita, and C. Zhang. Storing and querying ordered xml using a relational database system. In *SIGMOD*, 2002.

[34] P. M. Tolani and J. R. Haritsa. XGRIND: A Query-Friendly XML Compressor. In *ICDE*, pages 225–234, 2002.

[35] F. Wang and P. Liu. Temporal Management of RFID Data. In *VLDB*, pages 1128–1139, 2005.

[36] F. Wang, S. Liu, P. Liu, and Y. Bai. Bridging physical and virtual worlds: Complex event processing for rfid data streams. In *EDBT*, 2006.

[37] H. Wang, S. Park, W. Fan, and P. S. Yu. Vist: A dynamic index method for querying xml data by tree structures. In *SIGMOD*, 2003.

[38] W. Wu, W. Guo, and K.-L. Tan. Distributed Processing of Moving K-Nearest-Neighbor Query on Moving Objects. In *ICDE*, pages 1116–1125, 2007.

[39] X. Wu, M.-L. Lee, and W. Hsu. A prime number labeling scheme for dynamic ordered xml trees. In *ICDE*, 2004.

[40] C. Zhang, J. F. Naughton, D. J. DeWitt, Q. Luo, and G. M. Lohman. On supporting containment queries in relational database management systems. In *SIGMOD*, 2001.

Chun-Hee Lee received the B.S. degree in computer science from the Korea Advanced Institute of Science and Technology (KAIST), Korea, in 2003. He is currently a Ph.D. Candidate at KAIST. His research interests include sensor network and stream data management.

Chin-Wan Chung received the B.S. degree in electrical engineering from Seoul National University, Korea, in 1973, and the Ph.D. degree in computer engineering from the University of Michigan, Ann Arbor, USA, in 1983. From 1983 to 1993, he was a Senior Research Scientist and a Staff Research Scientist in the Computer Science Department at the General Motors Research Laboratories (GMR). While at GMR, he developed Dataplex, a heterogeneous distributed database management system integrating different types of databases. Since 1993, he has been a professor in the Department of Computer Science at the Korea Advanced Institute of Science and Technology (KAIST), Korea. At KAIST, he developed a full-scale object-oriented spatial database management system called OMEGA, which supports ODMG standards. His current major project is about mobile social networks in Web 3.0. He was in the program committees of major database conferences including ACM SIGMOD, VLDB, and IEEE ICDE. He was an associate editor of ACM TOIT, and is an associate editor of WWW Journal. His current research interests include the semantic Web, mobile Web, sensor network and stream data management, and multimedia databases. More information is available at http://islab.kaist.ac.kr/chungcw.