Efficient Storage Scheme and Query Processing for Supply Chain Management using RFID

Chun-Hee Lee
Div. of Computer Science
Korea Advanced Institute of Science and Technology (KAIST), Daejeon, Korea
leechun@islab.kaist.ac.kr

Chin-Wan Chung
Div. of Computer Science
Korea Advanced Institute of Science and Technology (KAIST), Daejeon, Korea
chungcw@islab.kaist.ac.kr

ABSTRACT

As the size of an RFID tag becomes smaller and the price of the tag gets lower, RFID technology has been applied to a wide range of areas. Recently, RFID has been adopted in the business area such as supply chain management. Since companies can get movement information for products easily using the RFID technology, it is expected to revolutionize supply chain management. However, the amount of RFID data in supply chain management is huge. Therefore, it requires much time to extract valuable information from RFID data for supply chain management.

In this paper, we define query templates for tracking queries and path oriented queries to analyze the supply chain. We then propose an effective path encoding scheme to encode the flow information for products. To retrieve the time information for products efficiently, we utilize a numbering scheme used in the XML area. Based on the path encoding scheme and the numbering scheme, we devise a storage scheme to process tracking queries and path oriented queries efficiently. Finally, we propose a method which translates the queries to SQL queries. Experimental results show that our approach can process the queries efficiently. On the average, our approach is about 680 times better than a recent technique in terms of query performance.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Query processing*

General Terms

Design, Performance

Keywords

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

1. INTRODUCTION

Different from existing technologies such as barcode systems and magnetic card systems that require contact between a detector and an object, it is possible for RFID (Radio Frequency IDentification) readers to detect RFID tags without contact in RFID systems. As the size of an RFID tag becomes smaller and the price of the RFID tag gets lower, RFID technology has been applied to many areas. A typical example is supply chain management. In supply chain management, an RFID tag is attached to a product and RFID readers in the detection region generate RFID data that is in the form of (*tag identifier* [1], *location, time*) when the product with the RFID tag moves or stays near the detection region. As the flow of the product is detected easily using the RFID technology, it is observed to revolutionize supply chain management.

The RFID data generated in various regions is sent to the central server. It can be grouped by the tag identifier and be transformed into stay records in the form of (*tag identifier, location, start time, end time*). While raw RFID data has many duplicates, transformed data (i.e., stay records) does not have duplicates. We can represent how long a tag stays at a location by *start time* and *end time* of stay records. Stay records for each tag compose a trace record that gives us the movement history with the time information for the tag. In this paper, we will use trace records as the basic building block to store RFID data in the central server.

As a straightforward method to store RFID data, we can store it in the relational table BASIC_TABLE(TAG_ID, LOC, START_TIME, END_TIME), 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. Queries to analyze the supply chain are related to the product transition. For example, a manager may ask the query "Find the number of notebooks which go through locations $Factory1$, $Distribution\ Center1$, and $Store1$." To evaluate the query with BASIC_TABLE, we must perform the self-join of BASIC_TABLE many times. Also, since the size of BASIC_TABLE is big, it requires much 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 [10]. In the model, STAY_TABLE(GID, LOC, START_TIME,

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

END_TIME, COUNT) is used to store RFID data efficiently. In many RFID applications, products usually move together in large groups at the early stages, and move in small groups at 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 by the prefix encoding scheme corresponds to GID in STAY_TABLE. To know whether two locations A and B are linked, they check whether the GID corresponding to A is the prefix of the GID corresponding to B.

Therefore, they reduce the size of the table significantly and improve the join cost. However, if products do not move together in large groups, the size of the table will not be reduced largely. In this case, the performance of the approach in [10] does not have a great benefit compared to that of the approach with BASIC_TABLE. Also, since the prefix encoding scheme uses the string comparison in joining tables, it needs much more time than the number comparison. Thus, we propose a new approach to store RFID data and process queries for supply chain management. Since queries related to the object transition in supply chain management was not defined formally in [10], we first define query templates for tracking and path oriented queries to analyze the supply chain. Then, to solve the above drawbacks, we propose a new approach.

While [10] focuses 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 for processing tracking queries and path oriented queries efficiently. The proposed path encoding scheme can encode a path with only two numbers (i.e., Element List Encoding Number and Order Encoding Number) which are motivated by [22]. In [22], to determine whether a relationship exists between two elements in an XML tree, the unique property of prime numbers is used. And, to preserve the global order for elements, simultaneous congruence values are used. In this paper, we use the unique property of prime numbers to encode nodes in the path, and simultaneous congruence values to encode ordering between 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 which 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 [23] 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.

1.1 Contributions

The contributions of the paper are as follows:

- **Encoding Scheme to Encode the Flows for 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 corresponding to the nodes in the path. Based on the Chinese Remainder Theorem, Order Encoding Number is computed. Using the two numbers, we can easily find the paths which 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 to translate tracking queries and path oriented queries into SQL queries.

- **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 [2].

- **Experimentation to validate our proposed approach** Through experiments, we show that our storage scheme and query processing are efficient. Experimental results show that the query performance of our approach is on the average 680 times and maximum 14278 times better than a recent approach.

1.2 Organizations

The rest of the paper is organized as follows. In Section 2, we discuss the related work on managing RFID data. In Section 3, we deal with data formats and define query templates for tracking queries and path oriented queries in supply chain management. We show the architecture to store RFID data and process queries in Section 4. We describe our path encoding scheme in Section 5 and devise a new relational schema in Section 6. We propose a method to translate tracking queries and path oriented queries into SQL queries in Section 7, and experimentally show the superiority of our approach in Section 8. We make a conclusion in Section 9.

2. RELATED WORK

In contrast to the initial study for RFID which focuses on the device, much work has been done recently to manage RFID data as the amount of RFID data becomes large.

The system architecture for managing RFID data is discussed in [7, 8, 11]. Bornhövd et al. [7] describe the Auto-ID infrastructure (Device/Deivce Operation/Business Processing Bridging/Enerpise application Layer) which integrates data from smart items (e.g., RFID, sensor) with existing business processes and Chawathe et al. [8] suggest a layered architecture for managing RFID data (Tag/Reader/Savant/ EPC-IS/ONS server). In [11], 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 and 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 exists two types of approaches for managing RFID data. One approach is on-line processing for RFID data and is related to data stream processing [20, 5, 4, 13]. The other approach is off-line processing and is related to stored data processing [6, 3, 12, 16, 19, 10, 9].

In the aspect of viewing RFID data as data streams, event processing and data cleaning have been studied. RFID data has a temporal property and it is important in analyzing data. Therefore, temporal RFID event can be defined. However, it cannot be well supported by traditional ECA (Event-Condition-Action) rule systems. Therefore, Wang et al. [20] 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. [5] explore the limitation of SQL in supporting the temporal event detection and discuss an SQL-based stream query language to provide a 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) [13] is proposed. SMURF controls the window size of the smoothing filter adaptively using statistical sampling. Also, Bai et al. [4] propose several methods to filter RFID data.

In the aspect of viewing RFID data as stored data, there exist various approaches to manage RFID data. Since RFID readers can detect RFID tags easily and quickly, object tracking using RFID is widely used. To trace tag locations, a new index is proposed in [6]. Ban et al. [6] point out the problem of representing the 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. To represent a collection of tag identifiers generated by item tracking applications compactly, the bitmap data type is proposed and a set of bitmap access and manipulation routines is provided in [12]. Agrawal et al. [3] 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 using the traditional ER-model. Therefore, Wang et al. [19] propose a new model called Dynamic Relationship ER Model (DRER) which simply adds a new relationship (dynamic relationship). They also propose methods to express temporal queries based on DRER using SQL queries. Although we can use the above techniques such as index and bitmap data type in order 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. [10] propose a new warehousing model for the object transition and a method to process a path selection query. To get aggregate measures on the 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.

3. 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 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 stay records of a tag, we can construct the trace record of the tag in the form of $TagID : L_1[S_1, E_1]-> \cdots -> L_k[S_k, E_k]$, where $L_1, \cdots, L_k$ 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 deal with 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) a set of trace records corresponding 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,* $tag1 : A[2,3]-> B[5,7]-> C[8,9]$.

(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

Figure 1: Raw Data and Trace Records

To analyze the supply chain, we use queries about the object transition. Although [10] mentions the path selection query, it is insufficient to express the relationship between locations. 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 the path conditions for products like XPath [2].

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 parent axis (/) or 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 using the query templates in Figure 2. We show some examples for tracking queries and path oriented queries in Figure 3.

```
[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 = 'Computer', 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.

Figure 2: Query Templates for Tracking Queries and Path Oriented Queries

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.

4. 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. The raw RFID data is transformed into trace records after sorting RFID data by the tag identifier and the time. We store trace records using an RDBMS since the RDBMS is a well-matured DBMS and our encoding scheme can be well implemented on it. Since we use prime numbers instead of location names, (Location, Prime Number) list is kept in memory as a hash structure. If a user requests a tracking or path oriented query, Query Translator translates it into an SQL query. Then, the SQL query is processed on an RDBMS and the result is sent to the user.

Semantics	Query
Find the movement history for the tag whose identifier is XXX. (Tracking Query)	TagID = XXX
Find the tags which go through locations L_1, ..., L_n . (Path Oriented Retrieval Query)	<//L_1//...//L_n>
Find the tags which 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 which 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 notebooks which go from L_1 to L_2. (Path Oriented Aggregation Query)	<min(L_2.StartTime), //L_1//L_2, ProductName='notebook'>

Figure 3: Examples for Tracking Queries and Path Oriented Queries

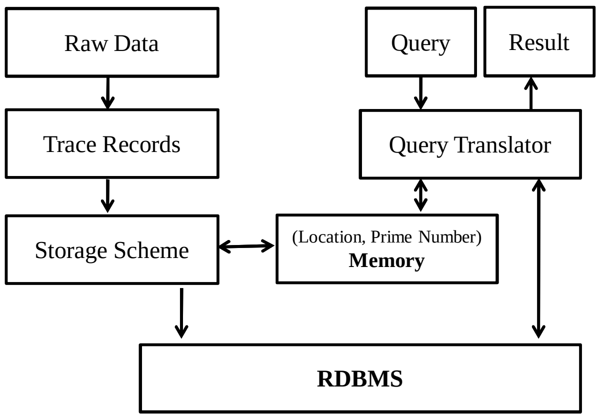


Figure 4: Architecture

5. 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 $TagID : L_1[S_1, E_1] - > \cdots - > L_k[S_k, E_k]$. 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 $L_1 - > \cdots - > L_k$. We then propose a path encoding scheme for encoding the path $L_1 - > \cdots - >$

L_k in order to analyze the object transition efficiently.

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, it is allowed that different paths include the same location (See the node C in Figure 6). The path of each tag becomes a path from the root in the tree by the elimination of duplicate nodes. The numbers beside nodes are prime numbers which will be explained subsequently. In Figure 6, the dark node means that there are tags whose final location is the node. We store all paths ending 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.

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]

tag5: A[2,3]->B[7,8]->D[14,18]

tag6: A[2,3]->E[4,6]->C[7,8]

tag7: A[2,3]->E[4,6]->C[7,8]

tag8: A[2,3]->E[4,6]

tag9: A[2,3]->D[4,5]

tag10: A[2,3]->D[5,6]

Figure 5: Example of Trace Records

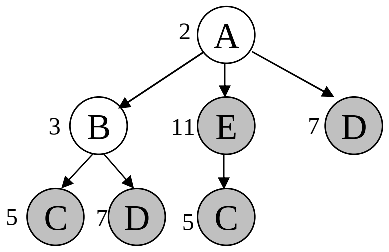


Figure 6: Tree Structure for the Trace Records

To encode a path, we can consider various techniques in the XML area [23, 14, 18, 21, 17, 15, 22]. However, in those techniques, it is inefficient to process queries which have many ancestor-descendant relationships such as "Find the tags which 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.

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 there does not exist the product of any other prime number combination for 231.

THEOREM 2. *The Chinese Remainder Theorem: Suppose that $n_1, n_2, \cdots, n_k$ are pairwise relatively prime numbers. Then, there exists X between 0 and $N(= n_1 n_2 \cdots n_k)$ solving the system of simultaneous congruences.*

$$\begin{aligned} X \ mod \ n_1 &= a_1 \\ X \ mod \ n_1 &= a_1 \\ &\cdots \\ X \ mod \ n_k &= a_k \end{aligned}$$

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 between 0 and $3 \times 7 \times 11$. In this example, X is 101. We can compute X using the extended Euclidean algorithm.

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

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

$1, 2, \cdots, \ and \ k$ are the levels (i.e., orders) of the nodes corresponding to $L_1, L_2, \cdots, \ and \ L_k$, respectively. Since $Prime(L_1), Prime(L_2), \cdots, \ and \ Prime(L_k)$ are prime numbers, they are pairwise relatively prime numbers. Thus, by Theorem 2, there exists X between 0 and $Prime(L_1) \times Prime(L_2) \times \cdots \times Prime(L_k)$ solving 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 $X \ mod \ Prime(L_i)$. Our use of prime numbers is similar to that in [22]. However, [22] assigns different prime numbers to each element in an XML tree which can have millions of elements while our approach assigns different prime numbers to different locations in a tree for trace records which has at most a few hundred locations. Therefore, in a typical application, [22] generates extremely large prime numbers. Furthermore, [22] orders entire 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 using Element List Encoding Number and Order Encoding Number. Although a path condition has multiple ancestor-descendant relationships, we can find 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 using the Element List Encoding Number and Order Encoding Number in Section 7.

Example 2. *Assume that in Figure 6, the prime numbers corresponding 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 between 0 and 30. In this case, $X = 23$. Similarly, we encode other four paths, and get Element List Encoding Numbers and Order Encoding Numbers for the paths.*

Even though we can store the flow information for products effectively 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 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 two nodes have the same location, they are classified as different nodes since they have the different time information.

To retrieve the time information efficiently, we store numbers with nodes in the time tree using the region numbering scheme [23] 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 nodes associated with a tag, we attach the tag to the node corresponding to the final location of the trace record of the tag. Thus, the region number corresponding 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 corresponding 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 which satisfy Start≤13 and End≥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.

6. SCHEMA FOR TRACKING QUERIES AND PATH ORIENTED QUERIES

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

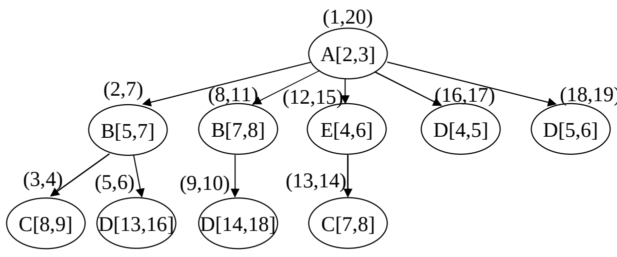


Figure 7: Time Tree Structure for the Trace Records

PATH_TABLE

PATH_ID	ELEMENT_ENC	ORDER_ENC

TAG_TABLE

TAG_ID	PATH_ID	START	END	TYPE

TIME_TABLE

START	END	LOC	START_TIME	END_TIME

INFO_TABLE

TYPE	PRODUCT_NAME	MANUFACTURER	PRICE

Figure 8: Relational Schema for Supply Chain Management

PATH_TABLE stores the path information using the encoding scheme in Section 5. In PATH_TABLE, the attribute ELEMENT_ENC corresponds to Element List Encoding Number and ORDER_ENC Order Encoding Number. TIME_TABLE stores the time information for trace records 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. And, TYPE is for the product information. INFO_TABLE stores the information of products. In this paper, we do not focus on INFO_TABLE.

Figure 9 shows the algorithm to store trace records using the relational schema. As the input for the algorithm, trace records are given. To fill PATH_TABLE, we construct the tree from the paths of the trace records *tr* using *constructTree(Tree tree, TraceRecord tr)* (Line 2). In *constructTree(Tree tree, TraceRecord tr)*, if there is not the path for *tr* in *tree*, it inserts the 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_TABLE (Line 4). In Line 5, we insert *tag identifier* and *path_id* into TEMP_PATH_TABLE. TEMP_PATH_TABLE is used to fill TAG_TABLE later.

To fill TIME_TABLE, we construct the time tree from the trace records *tr* using *constructTimeTree(TimeTree time_tree, TraceRecord tr)* (Line 6-7). In *constructTimeTree(TimeTree time_tree, TraceRecord tr)*, if there is not *tr* in *time_tree*, it inserts the new trace record into *time_tree*. After the con-

```
Input: trace records tr
begin
 1: for i=0; i<the number of trace records; i++
 2:     <path_id, store_flag>:=constructTree(tree, tr[i])
 3:     if store_flag == FALSE
 4:         store the path of trace record tr[i] in PATH_TABLE
                using the encoding scheme
 5:     store (tag identifier from tr[i], path_id) in
                TEMP_PATH_TABLE

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

 9: while traversing nodes in time_tree by the breath-first search
10:     store nodes of time_tree in TIME_TABLE
11:     store (tag identifier, region numbers for node) in
              TEMP_TIME_TABLE for all tags attached to node

12: after joining TEMP_PATH_TABLE and
          TEMP_TIME_TABLE on TAG_ID, fill TAG_TABLE
end
```

Figure 9: Algorithm to Store Trace Records using the Relational Schema

struction of *time_tree*, we assign region numbers to nodes in *time_tree* (Line 8). Then, we store (START, END, LOC, START_TIME, END_TIME) in TIME_TABLE (Line 10) by traversing nodes in *time_tree* by the breath-first search. If there are tags attached to a node, we store (tag identifier, region number for the node) in TEMP_TIME_TABLE (Line 11). Finally, to fill TAG_TABLE, we join TEMP_PATH_TABLE and TEMP_TIME_TABLE on TAG_ID (Line 12).

Figure 10 shows the status of tables after storing trace records in Figure 5 by the algorithm in Figure 9. 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. 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.

7. QUERY TRANSLATION

Based on the schema, we provide a method to process tracking queries and path oriented queries. Since we store RFID data using an RDBMS, we translate tracking queries and path oriented queries into SQL queries. We then can easily process the queries by executing the SQL queries.

7.1 Tracking Query

We can process a tracking query efficiently using the relational schema in Section 6. To trace a tag, we get Element List Encoding Number and Order Encoding Number corresponding to the tag. To get them, we join PATH_TABLE and TAG_TABLE. Figure 11 shows the SQL query to get Element List Encoding Number and Order Encoding Number corresponding to the tag with TagID = *my_tag_id*.

To know locations in the flow of the tag, we factorize Element List Encoding Number. We then order the prime

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_ID	PATH_ID	START	END	TYPE
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

Figure 10: Status of Tables after Storing Trace Records

```
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
```

Figure 11: SQL Query for the Tracking Query

number factors $P_1, \cdots, P_k$ by computing *Order Encoding Number mod* P_i. We finally transform the prime number into the location name corresponding to it.

Though the SQL query in Figure 11 has the join between PATH_TABLE and TAG_TABLE, it takes a little 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.

7.2 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 $ELEMENT\_ENC\ mod\ (L_1 \times L_2 \times \cdots \times L_k) = 0$. Therefore, if the path condition contains locations $L_1, L_2, \cdots,$ L_k, we insert $ELEMENT\_ENC\ mod\ (L_1 \times L_2 \times \cdots \times L_k) = 0$

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 12 shows the SQL query corresponding to the path oriented retrieval query <//A//B/C>. In Figures 12 through 17, 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
```

Figure 12: 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 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 13 shows the SQL query corresponding 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
```

Figure 13: SQL Query for <//A//B[(EndTime-StartTime)<10]/C>

Path oriented retrieval queries may also have the product information conditions such as PRODUCT_NAME = 'notebook'. 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 TYPE. Figure 14 shows the SQL query corresponding to the path oriented retrieval query <//A//B/C, PRODUCT_NAME = 'notebook'>

```
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 = 'notebook' AND I.TYPE = T.TYPE
```

Figure 14: SQL Query for <//A//B/C, PRODUCT_NAME = 'notebook'>

7.3 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 15 shows the SQL query corresponding 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
```

Figure 15: SQL Query for <COUNT(), //A//B/C>

In the case of aggregate functions that need the 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 16 shows the SQL query corresponding 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
```

Figure 16: 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 17 shows the SQL query corresponding 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
```

Figure 17: SQL Query for <AVG(C.EndTime - A.StartTime), //A//B/C>

8. EXPERIMENTS

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

8.1 Experimental Environment

We experiment on a Pentium 3GHz with 1GB main memory using Java. Since there is no a well known RFID data set, we generate synthetic data and formulate 12 queries (1 tracking query, 4 path oriented retrieval queries, 7 path oriented aggregate queries). The query performance is measured by processing queries 3 times and averaging the execution time. As the comparison system, we use RFID-Cuboid [10]. For fairness, we implement RFID-Cuboid on an RDBMS to support tracking queries and path oriented queries.

8.1.1 Data Set

Since we use stay records instead of raw RFID data, we generate stay records. As products move together in groups, in many RFID applications, we generate stay records with the grouping factor like [10]. We consider two kinds of data in data generation, GData and IData. In GData, many products move together in large groups and there are many stay records with the same location and time. In contrast to GData, in IData, products move together in small groups or individually. By generating GData and IData, we analyze how the query performance is affected by the grouping factor.

To generate GData and IData, we set parameters as shown in Figure 18. The grouping factor $(g_1, g_2, \cdots, g_k)$ means that, in the i-th stage, the number of products in a group is g_i. Since g_i in GData is much larger than g_i in IData, products in GData move together in much larger groups than those in IData. We set the minimum path length to 4 and the maximum path length to 8 in both GData and IData. To generate the paths from the minimum path length to the maximum path length, we use the parameter, the rate of

	GData	IData
Grouping factor	(1000, 500, 200, 100, 30, 10, 3, 1)	(50, 20, 10, 5, 3, 1, 1, 1)
Minimum path length	4	4
Maximum path length	8	8
The rate of moving to the location in the next stage	0.5	0.5

Figure 18: Parameters for GData and IData

moving to the location in the next stage. The generation for the next location is stopped or the next location is generated according to the rate. We set it to 0.5 in both GData and IData. We set the time information in stay records randomly with the maximum value. We generate 2×10^5, 4×10^5, 6×10^5, 8×10^5, and 10^6 stay records for both GData and IData with the parameters in Figure 18.

8.1.2 Query Set

We formulate 12 queries to test various features. Query 1 is a tracking query, Query 2-5 are path oriented retrieval queries, and Query 6-12 are path oriented aggregate queries. The queries are is shown in Figure 19. Note that the location names in Figure 19 are not the real location names of the synthetic data since the real names are long.

Query Number	Query
Query 1	TagID = 1
Query 2	<//F>
Query 3	</A/B/C/D>
Query 4	<//B//D//E>
Query 5	<//B//D[StartTime<200]//E>
Query 6	<COUNT(), //F>
Query 7	<COUNT(), //B//D//E>
Query 8	<COUNT(), //B//D[StartTime<200]//E>
Query 9	<AVG(B.EndTime - B.StartTime), //B//D//E>
Query 10	<AVG(F.StartTime), //F>
Query 11	<MIN(B.EndTime - B.StartTime), //B//D//E>
Query 12	<MIN(F.StartTime), //F>

Figure 19: Query Set

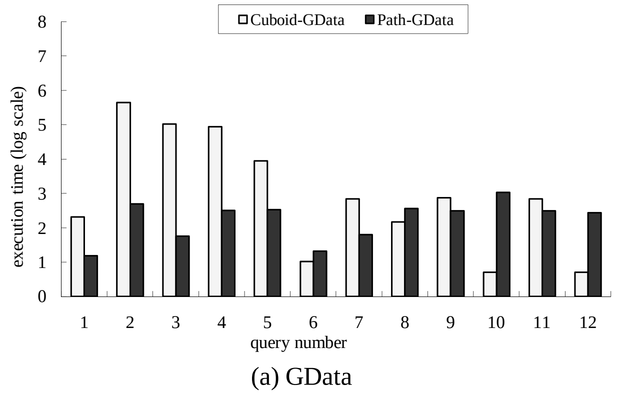
Query 1 tests the performance of the tracking query. Query 2, 6, 10, and 12 evaluate the performance for the simple path condition while other queries evaluate the performance for the complex path condition. Especially, Query 5 and 8 have the time condition.

8.2 Experimental Results

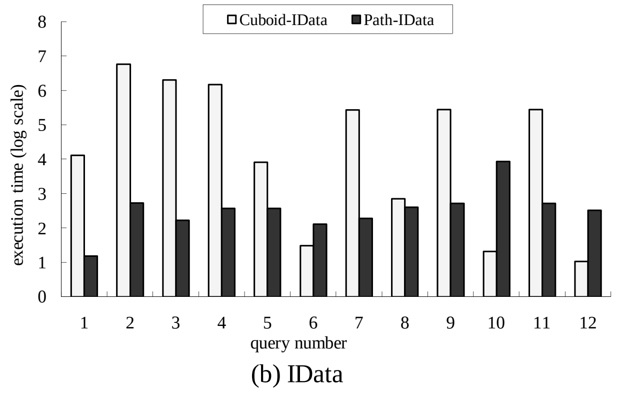
Figure 20 shows the query performance for 12 queries. Figure 20-(a) is the performance in GData with 10^6 tuples and Figure 20-(b) the performance in IData with 10^6 tuples.

We denote our approach Path and RFID-Cuboid Cuboid in figures of this section. Since the performance gap between our approach and RFID-Cuboid is wide, we use the logarithmic scale for the execution time in Figure 20. If the execution time of our approach is 2 times better than that of RFID-Cuboid in the graph with the logarithmic scale (base 10), the execution time of our approach is 100 times better than that of RFID-Cuboid.

In GData and IData, our approach is better than RFID-Cuboid for most queries in terms of the execution time (except Query 6, 10 and 12) Also, we can observe that the performance gap between our approach and RFID-Cuboid in IData is larger than that in GData. Since the shapes of graphs for the execution time for different sizes are similar, we show the execution time for only GData with 10^6 and IData with 10^6 in Figure 20.



(a) GData

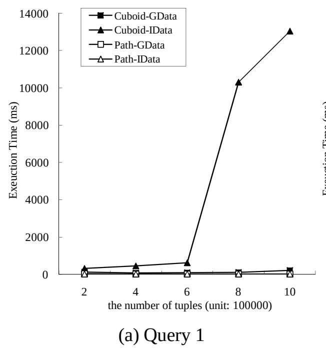


(b) IData

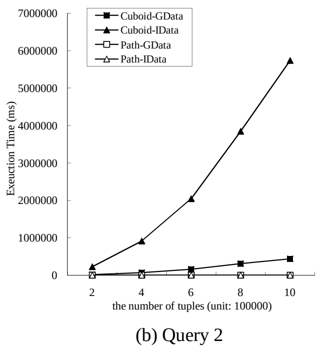
Figure 20: Execution Time for 12 Queries

Figure 21, 22, 23, 24, 25, and 26 show the query performance according to the number of stay records. Figure 21-(a) shows the performance for a tracking query. For the tracking query, our approach is faster than RFID-Cuboid in both GData and IData. Also, we can 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 query performance of Query 2, 3, and 4 is shown in

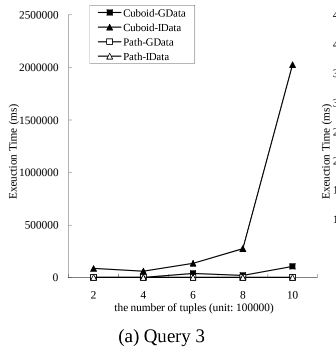


(a) Query 1

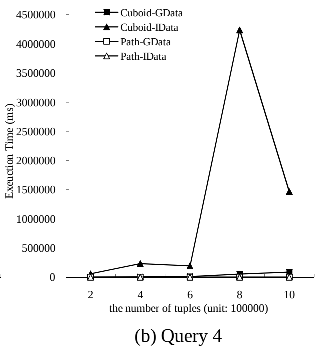


(b) Query 2

Figure 21: Execution Time for Query 1 and 2



(a) Query 3

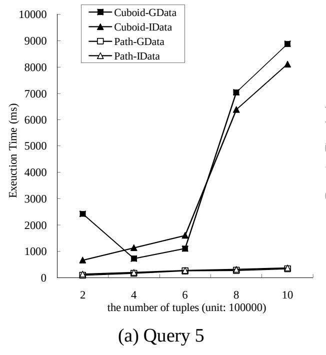


(b) Query 4

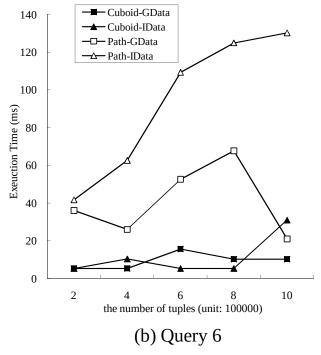
Figure 22: Execution Time for Query 3 and 4

Figure 21-(b), 22-(a), 22-(b). These queries are path oriented retrieval queries. To process path oriented retrieval queries, RFID-Cuboid joins STAY_TABLE and MAP_TABLE. MAP_TABLE is the table which contains mapping from GID to TAG_ID in RFID-Cuboid. Since RFID-Cuboid uses the prefix encoding scheme, it needs the string comparisons in processing the join between STAY_TABLE and MAP_TABLE. Therefore, for path oriented retrieval queries, our approach has much better performance than RFID-Cuboid.

The performance of Query 5 is shown in Figure 23-(a). Although RFID-Cuboid in GData shows 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 big difference in Figure 23-(a). Query 5 has the time condition (StartTime<200). There-

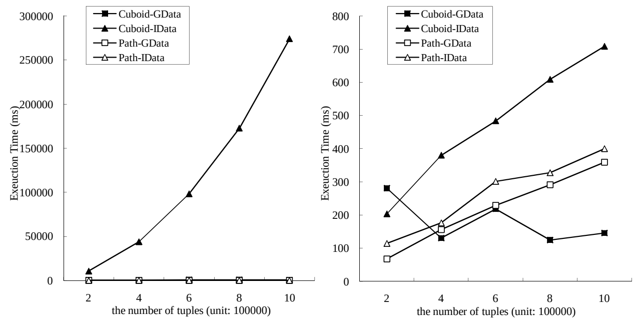


(a) Query 5



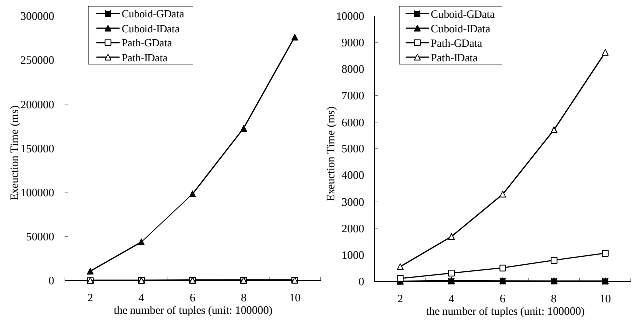
(b) Query 6

Figure 23: Execution Time for Query 5 and 6



(a) Query 7 (b) Query 8

Figure 24: Execution Time for Query 7 and 8

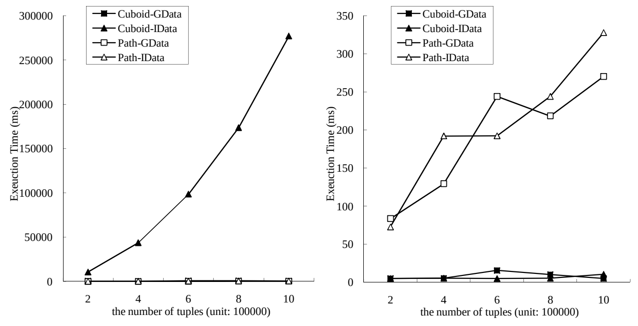


(a) Query 9 (b) Query 10

Figure 25: Execution Time for Query 9 and 10

fore, in IData, many tuples of STAY_TABLE are removed by the condition and tuples to join are reduced significantly. Therefore, the performance gap between IData and GData for RFID-Cuboid is small in Query 5.

In Query 6, 10 and 12 (Figure 23-(b), Figure 25-(b) and Figure 26-(b)), RFID-Cuboid is better than our approach. The path condition in Query 6, 10, and 12 has only one location. Since RFID-Cuboid focuses on groups in which products move together, it is efficient in the case of getting information on 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, and 11), our approach is superior to RFID-Cuboid. However, for Query 2, our approach has better performance than RFID-Cuboid although Query 2 is to get information on one location. This is because RFID-Cuboid uses the string comparison to get tags.



(a) Query 11 (b) Query 12

Figure 26: Execution Time for Query 11 and 12

Consider the query performance of Query 6, 10, and 12 (Figure 23-(b), Figure 25-(b) and Figure 26-(b)) versus that of Query 7, 9 and 11 (Figure 24-(a), Figure 25-(a) and Figure 26-(a)). The path condition in Query 6, 10 and 12 has only one location while the path condition in Query 7, 9 and 11 has multiple locations. Since our approach uses Element List Encoding Number and Order Encoding Number, it can easily find paths although the path condition has many ancestor-descendant relationships. Therefore, in Query 7, 9 and 11, our approach is better than RFID-Cuboid.

Since Query 8 has the time information, our approach joins TAG_TABLE and TIME_TABLE. Therefore, our approach has a little worse performance than RFID-Cuboid for GData as shown in Figure 24-(b).

Consequently, our approach is superior to RFID-Cuboid in most cases. The query performance of our approach is on the average 680 times and maximum 14278 times better than that of RFID-Cuboid. To compute the average ratio, we compute the ratios between the execution time of our approach (t_path) and that of RFID-Cuboid (t_cuboid) for all combinations of queries and data sets. We compute the ratio as t_cuboid/t_path if our approach is faster, while $-t\_path/t\_cuboid$ otherwise. Then we take the average of all the ratios. In addition, 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.

9. 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 the flow information different from the traditional data, it is difficult to store data and process queries. Therefore, we propose an efficient storage scheme and query processing for supply chain management. Also, we propose query templates to analyze the object transition in supply chain management. Since we can represent paths compactly and efficiently using our proposed path encoding scheme, queries for supply chain management are processed efficiently in our storage scheme. Finally, we show the superiority of our approach. In most queries, our approach is better than a recent approach. Our approach is not sensitive for the grouping factor.

10. ACKNOWLEDGMENTS

This work was partially supported by Defense Acquisition Program Administration and Agency for Defense Development under the contract (UD060048AD).

11. REPEATABILITY ASSESSMENT RESULT

All the results in this paper were verified by the SIGMOD repeatability committee.

12. REFERENCES

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

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

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

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

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

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

[7] 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.

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

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

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

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

[12] 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.

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

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

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

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

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

[18] 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.

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

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

[21] 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.

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

[23] 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.