Warehousing and Analyzing Massive RFID Data Sets*

Hector Gonzalez Jiawei Han Xiaolei Li Diego Klabjan
University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA
{hagonzal, hanj, xli10, klabjan}@uiuc.edu

Abstract

Radio Frequency Identification (RFID) applications are set to play an essential role in object tracking and supply chain management systems. In the near future, it is expected that every major retailer will use RFID systems to track the movement of products from suppliers to warehouses, store backrooms and eventually to points of sale. The volume of information generated by such systems can be enormous as each individual item (a pallet, a case, or an SKU) will leave a trail of data as it moves through different locations. As a departure from the traditional data cube, we propose a new warehousing model that preserves object transitions while providing significant compression and path-dependent aggregates, based on the following observations: (1) items usually move together in large groups through early stages in the system (e.g., distribution centers) and only in later stages (e.g., stores) do they move in smaller groups, and (2) although RFID data is registered at the primitive level, data analysis usually takes place at a higher abstraction level. Techniques for summarizing and indexing data, and methods for processing a variety of queries based on this framework are developed in this study. Our experiments demonstrate the utility and feasibility of our design, data structure, and algorithms.

1 Introduction

Radio Frequency Identification (RFID) is a technology that allows a sensor (RFID reader) to read, from a distance and without line of sight, a unique identifier that is provided (via a radio signal) by an "inexpensive" tag attached to an item. RFID offers a possible alternative to bar code identification systems and it facilitates applications like item tracking and inventory management in the supply chain. The technology holds the promise to streamline supply chain management, facilitate routing and distribution of products, and reduce costs by improving efficiency.

Large retailers like Walmart, Target, and Albertsons have already begun implementing RFID systems in their warehouses and distribution centers, and are requiring their suppliers to tag products at the pallet and case levels. Individual tag prices are expected to fall from around 25 cents per unit to 5 cents per unit by 2007. At that price level, we can expect tags to be placed at the individual item level for many products. The main challenge then becomes how can companies handle and interpret the enormous volume of data that an RFID application will generate. Venture Development Corporation [13], a research firm, predicts that when tags are used at the item level, Walmart will generate around 7 terabytes of data every day. Database vendors like Oracle, IBM, Teradata, and some startups are starting to provide solutions to integrate RFID information into enterprise data warehouses.

Example Suppose a retailer with 3,000 stores sells 10,000 items a day per store. Assume that we record each item movement with a tuple of the form: $(EPC, location, time)$, where EPC is an Electronic Product Code which uniquely identifies each item[1]. If each item leaves only 10 traces before leaving the store by going through different locations, this application will generate at least 300 million tuples per day. A manager may ask queries on the duration of paths like (Q_1): "*List the average shelf life of dairy products in 2003 by manufacturer*", or on the structure of the paths like (Q_2): "*What is the average time that it took coffee-makers to move from the warehouse to the shelf and finally to the checkout counter in January of 2004?*" New data structures and algorithms need to be developed that may provide fast responses to such queries even in the presence of terabyte-sized data. ■

Such enormous amount of low-level data and flexible high-level queries pose great challenges to traditional relational and data warehouse technologies since the processing may involve retrieval and reasoning over a large number of

* The work was supported in part by the U.S. National Science Foundation NSF IIS-02-09199/IIS-03-08215. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

[1] We will use the terms EPC and RFID tag interchangeably throughout the paper

inter-related tuples through different stages of object movements. No matter how the objects are sorted and clustered, it is difficult to support various kinds of high-level queries in a uniform and efficient way. A nontrivial number of queries may even require a full scan of the entire RFID database.

Model Proposal and Justification

In this paper we propose a new RFID data warehouse model to compress and aggregate RFID data in an organized way such that a wide range of queries can be answered efficiently. Our design is based on the following key observations.

First, we need to eliminate the redundancy present in RFID data. Each reader provides tuples of the form $(EPC, location, time)$ at fixed time intervals. When an item stays at the same location, for a period of time, multiple tuples will be generated. We can group these tuples into a single one of the form $(EPC, location, time\_in, time\_out)$. For example, if a supermarket has readers on each shelf that scan the items every minute, and items stay on the shelf on average for 1 day, we get a 1,440 to 1 reduction in size without loss of information.

Second, items tend to move and stay together through different locations. For example, a pallet with 500 cases of CDs may arrive at the warehouse; from there cases of 50 CDs may move to the shelf; and from there packs of 5 CDs may move to the checkout counter. We can register a single *stay* tuple of the form $(EPC\ list, location, time\_in, time\_out)$ for the CDs that arrive in the same pallet and stay together in the warehouse, and thus generate a 80% space saving.

An alternative compression mechanism is to store a single *transition* record for the 50 CDs that move together from the warehouse to the shelf, that is to group transitions and not stays. The problem with transition compression is that it makes it difficult to answer queries about items at a given location or going through a series of locations. For example, if we get the query "What is the average time that CDs stay at the shelf?" we can directly get the information from the *stay* records with $location = shelf$, but if we have *transition* records, we need to find all the transition records with $origin = shelf$ and the ones with $destination = shelf$, join them on EPC, and compute $departure\ time - arrival\ time$. Another method of compression would be to look at the sequence of locations that an item goes through as a string, and use a Trie data structure [6] to compress common path prefixes into a single node. The problem with this approach is that we lose compression power. In the CDs example, if the 50 items all stay at the warehouse together but they come from different locations, a Trie would have to create distinct nodes for each, thus gaining no compression.

Third, we can gain further compression by reducing the size of the EPC lists in the *stay* records by grouping items that move to the same locations. For example, let us say we have a *stay* record for the 50 CDs that stayed together at the warehouse, and that the CDs moved in two groups to shelf and truck locations. We can replace the list of 50 EPCs in the stay record for just two *generalized identifiers* (*gids*) which in turn point to the concrete EPCs. In this example we will store a total of 50 EPCs, plus two gids, instead of 100 EPCs (50 in the warehouse, 25 in the shelf, 25 in the truck). In addition to the compression benefits, we can gain query processing speedup by assigning path-dependent names to the gids. In the CDs example we could name the gid for the warehouse 1, and the gid for the shelf 1.1 and truck 1.2. If we get the query "What is the average time to go from the warehouse to the shelf for CDs?" instead of intersecting the EPC lists for the *stay* records at each location we can directly look at the gid name and determine if the EPCs are linked. The path-dependent naming scheme gives us the benefits of a tree structure representation of a Trie without taking a significant compression penalty.

Fourth, most queries are likely to be at a high level of abstraction, and will only be interested in the low-level individual items if they are associated with some interesting patterns discovered at a high level. For example, query (Q_1) asks about dairy products by manufacturer. It is possible after seeing the results, that the user may ask subsequent queries and drill down to individual items. We can gain significant compression by creating the stay records not at the raw level but at a minimal level of abstraction shared by most applications, while keeping pointers to the RFID tags. This allows us to operate on a much smaller data set, fetching the original data only when absolutely necessary.

The rest of the paper is organized as follows. Section 2 presents the structure of the input RFID data. Section 3 presents data compression and generalization methods important for the design of the RFID warehouse. Section 4 introduces algorithms for constructing the RFID warhouse. Section 5 develops methods for efficient processing of a variety of RFID queries. Section 6 reports the experimental and performance results. We discuss the related issues in Section 7 and conclude our study in Section 8.

2 RFID Data

Data generated from an RFID application can be seen as a stream of RFID tuples of the form $(EPC, location, time)$, where *EPC* is the unique identifier read by an RFID reader, *location* is the place where the RFID reader scanned the item, and *time* is the time when the reading took place. Tuples are usually stored according to a time sequence. A single EPC may have multiple readings at the same location, each reading is generated by the RFID reader scanning for tags at fixed time intervals

or on a continuous basis. Table 1 is an example of a raw RFID database where a symbol starting with r represents an RFID tag, l a location, and t a time. The total number of records in this example is 188.

Raw Stay Records
$(r1, l1, t1)$ $(r2, l1, t1)$ $(r3, l1, t1)$ $(r4, l1, t1)$ $(r5, l1, t1)$
$(r6, l1, t1)$ $(r7, l1, t1)$ ... $(r1, l1, t9)$ $(r2, l1, t9)$ $(r3, l1, t9)$
$(r4, l1, t9)$ ... $(r1, l1, t10)$ $(r2, l1, t10)$ $(r3, l1, t10)$ $(r4, l1, t10)$
$(r7, l4, t10)$ ... $(r7, l4, t19)$ ... $(r1, l3, t21)$ $(r2, l3, t21)$
$(r4, l3, t21)$ $(r5, l3, t21)$ ... $(r6, l6, t35)$ ... $(r2, l5, t40)$
$(r3, l5, t40)$ $(r6, l6, t40)$ ... $(r2, l5, t60)$ $(r3, l5, t60)$

Table 1. Raw RFID Records

In order to reduce the large amount of redundancy in the raw data, data cleaning should be performed. The output after data cleaning is a set of clean stay records of the form $(EPC, location, time\_in, time\_out)$ where *time_in* is the time when the object enters the location, and *time_out* is the time when the object leaves the location.

Data cleaning of stay records can be accomplished by sorting the raw data on EPC and time, and generating *time_in* and *time_out* for each location by merging consecutive records for the same object staying at the same location. Table 2 presents the RFID database of Table 1 after cleaning. It has been reduced from 188 records to just 17 records.

EPC	$Stay(EPC, location, time\_in, time\_out)$
r1	$(r1, l1, t1, t10)(r1, l3, t20, t30)$
r2	$(r2, l1, t1, t10)(r2, l3, t20, t30)(r2, l5, t40, t60)$
r3	$(r3, l1, t1, t10)(r3, l3, t20, t30)(r3, l5, t40, t60)$
r4	$(r4, 1, t1, t10)$
r5	$(r5, l2, t1, t8)(r5, l3, t20, t30)(r5, l5, t40, t60)$
r6	$(r6, l2, t1, t8)(r6, l3, t20, t30)(r6, l6, t35, t50)$
r7	$(r7, l2, t1, t8)(r7, l4, t10, t20)$

Table 2. A Cleansed RFID Database

3 Architecture of the RFID Warehouse

Before we describe our proposed architecture for warehousing RFID data, it is important to describe why a traditional data cube model would fail on such data. Suppose we view the cleansed RFID data as the fact table with dimensions $(EPC, location, time\_in, time\_out : measure)$. The data cube will compute all possible group-bys on this fact table by aggregating records that share the same values (or any *) at all possible combinations of dimension. If we use count as measure, we can get for example the number of items that stayed at a given location for a given month. The problem with this form of aggregation is that it does not consider links between the records. For example, if we want to get the number of items of type "dairy product" that traveled from the distribution center in Chicago to stores in Urbana, we cannot get this information. We have the count of "dairy products" for each location but we do not know how many of those items went from the first location to the second. We need a more powerful model capable of aggregating data while preserving its path-like structure.

We propose an RFID warehouse architecture that contains a fact table, *stay*, composed of cleansed RFID records; an information table, *info*, that stores path-independent information for each item, i.e., SKU information that is constant regardless of the location of the item such as manufacturer, lot number, color, etc.; and a *map* table that links together different records in the fact table that form a path. Figure 1 shows a logical view into the RFID warehouse schema. We call the *stay*, *info*, and *map* tables aggregated at a given abstraction level an *RFID-Cuboid*.

The main difference between the RFID warehouse and a traditional warehouse is the presence of the map table linking records from the fact table (*stay*) in order to preserve the original structure of the data.

The computation of *RFID-Cuboids* is more complex than that of regular cuboids as we will need to aggregate the data while preserving the structure of the paths at different abstraction levels.

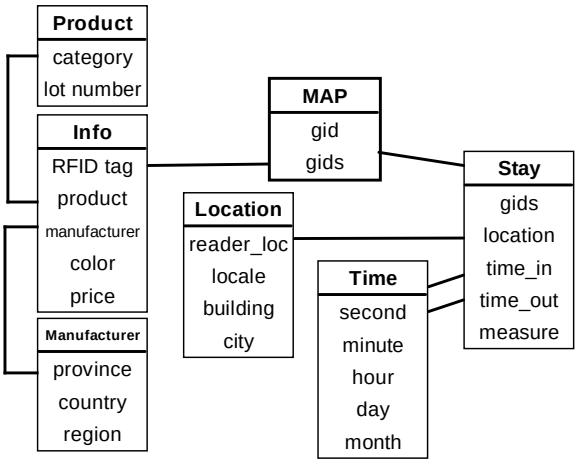

Figure 1. RFID Warehouse - Logical Schema

From the data storage and query processing point of view the RFID warehouse can be viewed as a multi-level database. The raw RFID repository resides at the lowest level, on its top are the cleansed RFID database, the minimum abstraction level *RFID-Cuboids* and a sparse subset of the full cuboid lattice composed of frequently queried (popular) *RFID-Cuboids*.

3.1 Key ideas of RFID data compression

Even with the removal of data redundancy from RFID raw data, the cleansed RFID database is usually still enormous. Here we explore several general ideas for constructing a highly compact RFID data warehouse.

Taking advantage of bulky object movements

Since a large number of items travel and stay together through several stages, it is important to represent such a collective movement by a single record no matter how many items were originally collected. As an example, if 1,000 boxes of milk stayed in location loc_A between time t_1 (time_in) and t_2 (time_out), it would be advantageous if only one record is registered in the database rather than 1,000 individual RFID records. The record would have the form: $(gid, prod, loc_A, t_1, t_2, 1000)$, where 1,000 is the count, *prod* is the product id, and gid is a generalized id which will not point to the 1,000 original EPCs but instead point to the set of new gids which the current set of objects move to. For example, if this current set of objects were split into 10 partitions, each moving to one distinct location, gid will point to 10 distinct new gids, each representing a record. The process iterates until the end of the object movement where the concrete EPCs will be registered. By doing so, no information is lost but the number of records to store such information is substantially reduced.

Taking advantage of data generalization

Since many users are only interested in data at a relatively high abstraction level, data compression can be explored to group, merge, and compress data records. For example, if the minimal granularity of time is hour, then objects moving within the same hour can be seen as moving together and be merged into one movement. Similarly, if the granularity of the location is shelf, objects moving to the different layers of a shelf can be seen as moving to the same *shelf* and be merged into one. Similar generalization can be performed for products (*e.g.*, merging different sized milk packages) and other data as well.

Taking advantage of the merge and/or collapse of path segments

In many analysis tasks, certain path segments can be ignored or merged for simplicity of analysis. For example, some non-essential object movements (*e.g.*, from one shelf to another in a store) can be completely ignored in certain data analysis. Some path segments can be merged without affecting the analysis results. For store managers, merging all the movements before the object reaches the store could be desirable. Such merging and collapsing of path segments may substantially reduce the total size of the data and speed-up the analysis process.

3.2 RFID-CUBOID

With the data compression principles in mind, we propose, the *RFID-Cuboid* a data structure, for storing aggregated data in the RFID warehouse. Our design ensures that the data are disk-resident, summarizing the contents of a cleansed RFID database in a compact yet complete manner while allowing efficient execution of both OLAP and tag-specific queries.

The *RFID-Cuboid* consists of three tables: (1) Info, which stores product information for each RFID tag, (2) Stay, which stores information on items that stay together at a location, and (3) Map, which stores path information necessary to link multiple stay records.

Information Table

The information table stores path-independent dimensions such as product name, manufacturer, product price, product category, etc. Each dimension can have an associated concept hierarchy. All traditional OLAP operations can be performed on these dimensions in conjunction with various RFID-specific analysis. For example, one could drill-down on the product category dimension from "clothing" to "shirts" and retrieve shipment information only on shirts.

The information table (*info*) contains a set of attributes that provide a path-independent extended description of RFID tags. Each entry in Info is a record of the form: $\langle (EPC\_list), (d_1, \ldots, d_m) : (m_1, \ldots, m_i) \rangle$, where the code list contains a set of items (*i.e.*, rfids) that share the same values for dimensions $d_1, \ldots, d_m$, and $m_1, \ldots, m_i$ are measures of the given items, e.g., price. Table 3 presents and example *info* table.

EPC_List	Product	Manufacturer	Price	Weight
(r2,r3,r7)	TV	Sony	$300	50 lb
(r1,r4)	Computer	Sony	$900	6 lb

Table 3. Info Table

Stay Table

As mentioned in the introduction, items tend to move and stay together through different locations. Compressing multiple items that stay together at a single location is vital in order to reduce the enormous size of the cleansed RFID database. In real applications items tend to move in large groups. At a distribution center there may be tens of pallets staying together, and then they are broken into individual pallets at the warehouse level. Even if products finally move at the individual item level from a shelf to the checkout counter, our stay compression will save space for all previous steps taken by the item.

The Stay Table (*stay*) contains an entry for each group of items that stay together at a certain location. Each entry in *stay* is a record of the form: $\langle (gids, location, time\_in, time\_out) : (m_1, \ldots, m_k) \rangle$, where $gids$ is a set of generalized record ids each pointing to a list of RFID tags or lower level $gids$, $location$ is the location where the items stayed together, $time\_in$ is the time when the items entered the location, and $time\_out$ the time when they left. If the items did not leave the location, $time\_out$ is $NULL$. $m_1, \ldots, m_n$ are the measures recorded for the stay, e.g., count, average time at $location$, and the maximal time at $location$.

Table 4 presents the stay table for the cleansed data from Table 2. We have now gone from 188 records in the raw data, to 17 in the cleansed data, and then to 7 in the compressed data.

gid	loc	t1	t2	count	measure
0.0	l1	t1	t10	4	9
0.1	l2	t1	t8	3	7
0.0.0	l3	t20	t30	3	9
0.1.0	l3	t20	t30	2	19
0.1.1	l4	t10	t20	1	19
0.0.0.0,0.1.0.0	l5	t40	t60	3	19
0.1.0.1	l6	t35	t50	1	14

Table 4. Stay Table

Map Table

The *map* table is an efficient structure that allows query processing to link together stages that belong to the same path in order to perform structure-aware analysis, which could not be answered by a traditional data warehouse. There are two main reasons for using a *map* table instead of recording the complete EPC lists at each stage: (1) data compression, and (2) query processing efficiency.

First, we do not want to record each RFID tag on the EPC list for every *stay* record it participated in. For example, if we assume that 10000 items move in the system in groups of 10000, 1000, 100, and 10 through 4 stages, instead of using 40,000 units of storage for the EPCs in the *stay* records, we use only 1,111 units [2] (1000 for the last stage, 100, 10, and 1 for the ones before).

The second and the more important reason for having such a map table is the efficiency in query processing. Suppose each map entry were given a path-dependent label. To compute, for example, the average duration for milk to move from the distribution center (D), to the store backroom (B), and finally to the shelf (S), we need to locate the stay records for milk at each stage. To get three sets of records D, B, and S, one has to intersect the EPC lists of the records in D with those in B and S to get the paths. By using the map, the EPC lists can be orders of magnitude shorter and thus reduce IO costs. Additionally, if we use path-dependent naming of the map entries, we can compute list intersection much faster.

The map table contains mappings from higher level $gids$ to lower level ones or EPCs. Each entry in *map* is a record of the form: $\langle gid, (gid_1, \ldots, gid_n) \rangle$, meaning that, gid is composed of all the EPCs pointed to by $gid_1, \ldots, gid_n$. The lowest level $gids$ will point directly to individual items.

In order to facilitate query processing we will assign path-dependent labels to high level $gids$. The label will contain one identifier per location traveled by the items in the gid. Figure 2 presents the gid map for the cleansed data in Table 2. We see that the group of items with gid 0.0 arrive at l1, and then subdivide into 0.0.0, and s0.0, the items in s0.0 (we prefix the label with *s* to denote that the items stay in the location) stay in location l1 while the items in 0.0.0 travel to l3, and subdivide further into 0.0.0.0 and s0.0.0, the items in s0.0.0 stay at location l3, while the ones in 0.0.0.0 move to location l5.

Map - Tabular View

0	0.0, 0.1
0.0	0.0.0,s0.0
0.0.0	0.0.0.0,s0.0.0
0.0.0.0	r2,r3
s0.0	r4
s0.0.0	r1
0.1	0.1.0,0.1.1
0.1.0	0.1.0.0,0.1.0.1
0.1.0.0	r5
0.1.0.1	r6
0.1.1	r7

Map - Graphical View

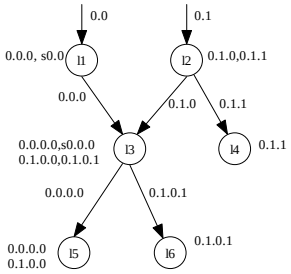

Figure 2. GID Map - Tabular and Graphical Views

3.3 Hierarchy of RFID-Cuboids

Each dimension in the *stay* and *info* tables has an associated concept hierarchy. A concept hierarchy is a partial order of mappings from lower levels of abstraction to higher ones. The lowest corresponds to the values in the raw RFID data stream itself, and the associated per item information, which could be at the Stock Keeping Unit (SKU) level. The highest is $*$ which represents any value of the dimension.

In order to provide fast response to queries specified at various levels of abstraction, it is important to pre-compute some *RFID-Cuboids* at different levels of the concept hierarchies for the dimensions of the *info* and *stay* tables. It is obviously too expensive to compute all the possible generalizations, and partial materialization is a preferred

[2] This figure does not include the size of the map itself which should use 12,221 units of storage, still much smaller than the full EPC lists

choice. This problem is analogous to determining which set of cuboids in a data cube to materialize in order to answer OLAP queries efficiently given the limitations on storage space and precomputation time. This issue has been studied extensively in the data cube research [2, 8], and the principles are generally applicable to the selective materialization of *RFID-Cuboids*.

In our design, we suggest to compute a set of *RFID-Cuboids* at the minimal interesting level at which users will be interested in inquiring the database, and a small set of higher level structures that are frequently requested and that can be used to quickly compute non-materialized *RFID-Cuboids*.

An *RFID-Cuboid* residing at the minimal interesting level will be computed directly from the cleansed RFID database and will be the lowest cuboid that can be queried unless one has to dig directly into the cleansed data in some very special cases.

4 Construction of an RFID Warehouse

In this section, we study the construction of the RFID warehouse from the cleansed RFID database.

In order to construct the *RFID-Cuboid*, as described in the paper, we need a compact data structure that allows us to do the following efficiently: (1) assign path-dependent labels to the gids, (2) minimize the number of output *stay* records while computing aggregates that preserve the path-like nature of the data, and (3) identify path segments that can be collapsed. We argue that using a tree-like structure to represent the different paths in the database is an ideal solution, where each node in the tree will represent a path stage; all common path prefixes in the database will share the same branch in the tree. Using the tree we can traverse the nodes in the breath-first fashion while assigning path-dependent labels. We can quickly determine the minimum number of output *stay* records by aggregating the measures of all the items that share the same branch in the tree. And we can collapse path segments by simply merging parent/child nodes that correspond to the same location. Additionally, the tree can be constructed by doing a single scan of the cleansed RFID database, and it can be discarded after we have materialized the output *info*, *stay*, and *map* tables.

The most common implementation of the group-by operator, which would sort the input records on location, time_in, and time_out, and generate a list of RFID tags that share the same values on those dimensions, would fail in generating the *RFID-Cuboid* in an efficient manner. The problem with this approach is that we lose the ability to determine which of the items that share the same location, time_in, and time_out values can actually have their measures aggregated into a single measure. The reason is that we cannot quickly determine which subset of all the items followed the same path throughout the system without doing an unmanageable number of item list intersections. For this same reason we cannot easily construct the map table, assign path-dependent labels to the gids, or determine if two different records can be collapsed into one.

Algorithm 1 summarizes the method for constructing an *RFID-Cuboid* from the cleansed RFID database. It takes as input the clean *stay* records S, the *info* records I describing each item, and the level of abstraction for each dimension L. The output of the algorithm is an *RFID-Cuboid* represented by the *stay*, *info*, and *map* tables aggregated at the desired abstraction level.

First, we aggregate the information table to the level of abstraction specified by L. This can be done by using a regular cubing algorithm.

Second, we call the BuildPathTree procedure (Algorithm 2) which constructs a tree for the paths traveled by items in the cleansed database. The paths that have the same branch in the tree share the same path prefix. Each node in the tree is of the form $(location, time\_in, time\_out, measure\_list, children\_list)$, where $time\_in$ is the time at which the items entered $location$, $time\_out$ the time at which they left, $measure\_list$ contains the stay measures (from the parent's location to this node's location) for every item that stayed at the node, and $children\_list$ is a list of nodes where the items in the node moved next. All processing from this point on in the algorithm will be done using the tree which we will call *path_tree*.

Third, we merge consecutive nodes in the tree that correspond to the same location. This happens because it is possible for two distinct locations to be aggregated to the same higher level location. This step achieves compression by collapsing paths at the given abstraction level.

Fourth, we generate gids for the nodes in the tree. This is done in the breath-first order, where each node receives a unique id that is appended to the gid of its parent node. This naming scheme is used to speed up the computation of linked stay records for query processing.

Fifth, we take the $measure\_list$ at each node and compact it to one aggregate measure for each group of RFID tags that share the same leaf (descendant from the node) and *info* record. Further compression is done when several leaves share the same measure.

Finally, we traverse the tree, generating the new *stay* table: each node generates as many records as the number of distinct measures it contains. It is possible for multiple nodes to share the same *stay* record if they share all the attributes and measures. For example, if there are two nodes in the path_tree that have the same location, $time\_in$, $time\_out$, and measures, we generate a single *stay* record that has two gids, one for each node.

4.1 Construction of Higher Level RFID-Cuboids from Lower Level Ones

Once we have constructed the minimum abstraction level *RFID-Cuboid*, it is possible to gain efficiency by constructing higher level *RFID-Cuboids* starting from the existing *RFID-Cuboid* instead of directly from the cleansed RFID data. This can be accomplished by running Algorithm 1 with the *stay* and *info* tables of the lower level *RFID-Cuboid* as input, and using the *map* table of the input *RFID-Cuboid* to expand each gid to the EPCs it points to. The benefit of such approach is that the input *stay* and *info* tables can be significantly smaller than the cleansed tables, thus greatly gaining in space and time efficiency.

Algorithm 1 BuildCuboid

Input: Stay records S, Information records I, aggregation level L
Output: *RFID-Cuboid*
Method:

1: I' = aggregate I to L;
2: path_tree = BuildPathTree (S, L);
3: Merge consecutive nodes with the same location in the path_tree;
4: Traverse path_tree in the breath-first order and assign gid to each node (gid = parent.gid + '.' + unique id), where the parent/children gid relation in the path_tree defines the output MAP;
5: Compute aggregate measures for each node, one per group of items with the same information record and in the same leaf;
6: Create S' by traversing path_tree in the breath-first order, and generating stay records for each node (Multiple nodes can contribute to the same stay record);
7: Output MAP, S', and I'

Observation. Given a cleansed stay and info input tables, the RFID cuboid structure has the following properties:

1. **(Construction cost)** The *RFID-Cuboid* can be constructed by doing a single sequential scan on the cleansed stay table.

2. **(Completeness)** The RFID cuboid contains sufficient information to reconstruct the original RFID database aggregated at abstraction level L.

3. **(Compactness)** The number of records in the output *stay* and *info* tables is no larger than the number of records in the input *stay* and *info* tables respectively.

   Rationale. The first property has been shown in the *RFID-Cuboid* construction algorithm. The completeness property can be proved by using the *map* table to expand the gids for each *stay* record to get the original data. The compactness property is proved noticing that Algorithm 1 emits at most one output record per distinct measure per node, and the number of distinct measures in a node is limited by the number of input *stay* records. The size of the output *info* table, by definition of the group-by operation is bound by the size of the input *info* table. ■

Algorithm 2 BuildPathTree

Input: Stay S, aggregation level L
Output: path_tree
Method:

1: root = new node;
2: **for** each record s in S **do**
3: s' = aggregate s to level L;
4: parent = lookup node for s'.rfid;
5: **if** parent == NULL **then**
6: parent = root;
7: **end if**
8: node = lookup s'.rfid in parent's children;
9: **if** node == NULL **then**
10: node = new node;
11: node.loc = s.loc;
12: node.t1 = s.t1;
13: node.t2 = s.t2;
14: add node to parent's children;
15: **end if**
16: node.measure_list += $\langle s'.gid, s'.measure \rangle$;
17: **end for**
18: return root

5 Query Processing

In this section we discuss the implementation of the basic OLAP operations, i.e., drill-down, roll-up, slice, and dice, applied to the RFID data warehouse, and introduce a new operation, Path Selection, relevant to the paths traveled by items.

Given the very large size and high dimensionality of the RFID warehouse we can only materialize a small fraction of the total number of *RFID-Cuboids*. We will compute the *RFID-Cuboid* that resides at the minimum abstraction layer that is interesting to users, and those *RFID-Cuboids* that are frequently requested. Initially, the warehouse designer may decide to materialize a subset of *RFID-Cuboids* that are interesting to the users, and as the query log is built, we can perform frequency counting to determine frequently requested *RFID-Cuboids* that should be pre-computed. When a roll-up or drill-down operation requires an *RFID-Cuboid* that has not yet been materialized, it would have to be computed on the fly from an existing *RFID-Cuboid* that is close

to the required one but at a lower abstraction level.

The slice and dice operations can be implemented quite efficiently by using relational query execution and optimization techniques. An example of the dice operation could be: "Give me the average time that milk stays at the shelf in store S_1 in Illinois". This query can be answered by the relational expression:

$\sigma_{stay.location='shelf',info.product='milk'}(stay \bowtie_{gid} info)$.

Path Selection

Path queries, which ask about information related to the structure of object traversal paths, are unique to the RFID warehouse since the concept of object movements is not modeled in traditional data warehouses. It is essential to allow users to inquire about an aggregate measure computed based on a predefined sequence of locations (path). One such example could be: "What is the average time for milk to go from farms to stores in Illinois?".

Queries on the paths traveled by items are fundamental to many RFID applications and will be the building block on top of which more complex data mining operators can be implemented. We will illustrate this point with two real examples. First, the United States government is currently in the process of requiring the containers arriving into the country by ship to carry an RFID tag. The information can be used to determine if the path traveled by a given container has deviated from its historic path. This application may need to first execute a path-selection query across different time periods, and then use outlier detection and clustering to analyze the relevant paths. Second, plane manufacturers are planning to tag airplane parts to better record maintenance operations on each part. Again using path selection queries you could easily associate a commonly defective part with a path that includes a certain set of suppliers that provide raw materials for its construction and are likely the sources of the defect. This task may need frequent itemset counting and association mining on the relevant paths. These examples show that path selection can be crucial to successful RFID data analysis and mining, and it is important to design good data structures and algorithms for efficient implementation.

More formally, a path selection query is of the form:

$$q \leftarrow \langle \sigma_c \textit{info}, (\sigma_{c_1} stage_1, \ldots, \sigma_{c_k} stage_k) \rangle,$$

where σ_c*info* means the selection on the *info* table based on condition c, and $\sigma_{c_i} stage_i$ means the selection based on condition c_i on the *stay* table $stage_i$. The result of a path selection query is a set of paths whose stages match the stage conditions in the correct order (possibly with gaps), and whose items match the condition c. The query expression for the example path query presented above is $c \leftarrow \langle product = \text{“}milk\text{”} \rangle$, $c_1 \leftarrow \langle location = \text{“}farm\text{”} \rangle$, and $c_2 \leftarrow \langle location = \text{“}store\text{”} \rangle$. We can compute aggregate measures on the results of a path selection query, e.g., for the example query the aggregate measure would be the average time.

Algorithm 3 illustrates the process of selecting the gids matching a given query. We first select the $gids$ for the stay records that match the conditions for the initial and final stages of the query expression. For example, g_{start} may look like $\langle 1.2, 8.3.1, 3.4 \rangle$ and g_{end} may look like $\langle 1.2.4.3, 4.3, 3.4.3 \rangle$. We then compute the pairs of gids from g_{start} that are a prefix of a gid in g_{end}. Continuing with the example we get the pairs $\langle (1.2, 1.2.4.3), (3.4, 3.4.3) \rangle$. For each pair we then retrieve all the stay records. The pair $(1.2, 1.2.4.3)$ would require us to retrieve stay records that include gids 1.2, 1.2.4, and 1.2.4.3. Finally, we verify that each of these records matches the selection conditions for each $stage_i$ and for $info$, and add those paths to the answer set.

If we have statistics on query selectivity, it may be possible to find a better optimization query execution plan than that presented in Algorithm 3. If we have a sequence of stages $(stage_1, \ldots, stage_k)$, we could retrieve the records for the most selective stages, in addition to retrieving the *stay* records for $stage_1$ and $stage_k$, in order to further prune the search space.

Algorithm 3 PathSelection

Input: $q \leftarrow \langle \sigma_c \textit{info}, (\sigma_{c_1} stage_1, \ldots, \sigma_{c_k} stage_k) \rangle$,, RFID warehouse.
Output: the paths that match query conditions, q.
Method:

1: g_{start} = select gids of stay records matching the condition at $stage_1$;
2: g_{end} = select gids of stay records matching the condition at $stage_k$ and that for $info$;
3: **for** every pair of gids (s, e) in g_{start}, g_{end} such that s is a prefix of e **do**
4: $path$ = retrieve stay records for all gids from s to e;
5: **if** the stay records in $path$ match conditions for *info* and for the remaining stages **then**
6: $answer = answer + path$;
7: **end if**
8: **end for**
9: return $answer$

Analysis. Algorithm 3 can be executed efficiently if we have a one-dimensional index for each dimension of the stay table; a one-dimensional index on the gid dimension of the map table; and a one-dimensional index on the EPC dimension of the info table. The computation of g_{start} and g_{end} can be done by retrieving the records that match each condition at $stage_1$ and $stage_k$, and intersecting the results. Verification of the condition σ_c*info* is done by using the map

table to retrieve the base gids for each gid in $stage_k$ that has as prefix a gid in $stage_1$. The info record for each base_gid can be retrieve efficiently by using the EPC index on the info table.

6 Performance Study

In this section, we perform a thorough analysis of our model and algorithms. All experiments were implemented using C++ and were conducted on an Intel Xeon 2.5GHz (512KB L2 cache) system with 3GB of RAM. The system ran Red Hat Linux with the 2.4.21 kernel and gcc 3.2.3.

6.1 Data Synthesis

The RFID databases in our experiments were generated using a tree model for object movements. Each node in the tree represents a set of items in a location, and an edge represents a movement of objects between locations. We assumed that items at locations near the root of the tree move in larger groups, while tags near the leaves move in smaller groups. The size of the groups at each level of the tree define the bulkiness, $\mathcal{B} = (s_1, s_2, \ldots, s_k)$, where s_i is the number of objects that stay and move together at level i of the tree. By making $s_i \geq s_j$ for $i > j$ we create the effect of items moving in larger groups near the factory and distribution centers, and smaller groups at the store level. We generated the databases for the experiments by randomly constructing a set of trees with a given level of Bulkiness, and generating the cleansed RFID records corresponding to the item movements indicated by the edges in the tree.

As a notational convenience, we use the following symbols to denote certain dataset parameters.

$\mathcal{B} = (s_1, \ldots, s_k)$	Path Bulkiness
k	Average path length
$\mathcal{P}$	Number of products
$\mathcal{N}$	Number of cleansed RFID records

6.2 RFID-Cuboid compression

The *RFID-Cuboids* form the basis for future query processing and analysis. As mentioned previously, the advantage of these data structures is that they aggregate and collapse many records in the cleansed RFID database. Here, we examine the effects of this compression on different data sets. We will compare two different compression strategies, both use the *stay* and *info* tables, but one uses the map table as described in the paper (map), whereas the other uses a tag list to record the tags at each stay record (nomap).

Figure 3 shows the size of the cleansed RFID database (raw) compared with the map and nomap *RFID-Cuboids*. The datasets contains 1,000 distinct products, traveling in groups of 500, 150, 40, 8, and 1 through 5 path stages, and 500 thousand to 10 million cleansed RFID records. The *RFID-Cuboid* is computed at the same level of abstraction of the cleansed RFID data, and thus the compression is lossless. As it can be seen from Figure 3 the *RFID-Cuboid* that uses the map has a compression power of around 80% while the one that uses tag lists has a compression power of around 65%. The benefit of the map comes from the fact that it avoids registering each tag at each location. In both cases the compression provided by collapsing stay records is significant.

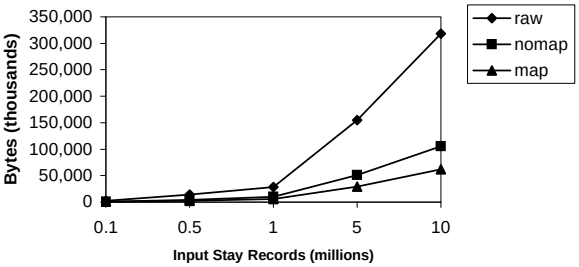

Figure 3. Compression vs. Cleansed Data Size. $\mathcal{P} = 1000$**,** $\mathcal{B} = (500, 150, 40, 8, 1)$**,** $k = 5$**.**

Figure 4 also shows the size of the cleansed RFID database (raw) compared with the map and nomap *RFID-Cuboids*. In this case we vary the degree of bulkiness of the paths, e.g., the number of tags that stay and move together through the system. We define 5 levels of bulkiness $a = (500, 230, 125, 63, 31)$, $b = (500, 250, 83, 27, 9)$, $c = (500, 150, 40, 8, 1)$, $d = (200, 40, 8, 1, 1)$, and $e = (100, 10, 1, 1, 1)$. The bulkiness decreases from dataset a to e. As it can be seen in the figure, for more bulky data the *RFID-Cuboid* that uses the map clearly outperforms the nomap cuboid; as we move towards less bulky data the benefits of the map decrease as we get many entries in the map that point to just one gid. For paths where a significant portion of the stages are traveled by a single item the benefit of the map disappears and we are better off using tag lists. A possible solution to this problem is to compress all map entries that have a single child into one.

Figure 5 shows the compression obtained by climbing along the concept hierarchies of the dimensions in the stay and info tables. Level-0 cuboids have the same level in the hierarchy as the cleansed RFID data. The three higher level cuboids offer one, two, and three levels of aggregation respectively at all dimensions (location, time, product, manufacturer, color). As expected the size of the cuboids at higher levels decreases. In general the cuboid using the map is smaller, but for the top most level of abstraction the size is the same as for the nomap cuboid. At level 3 the size of the stay table is just 96 records, and most of the space is actually used by recording the RFID tags themselves and thus

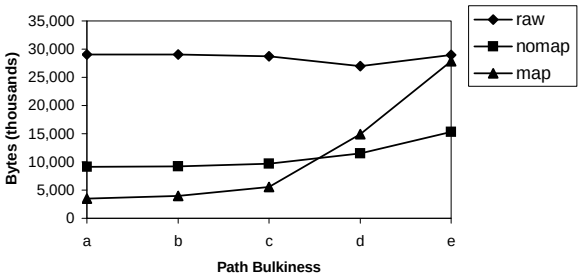

Figure 4. Compression vs. Data Bulkiness. $\mathcal{P} = 1000$**,** $\mathcal{N} = 1,000,000$**,** $k = 5$**.**

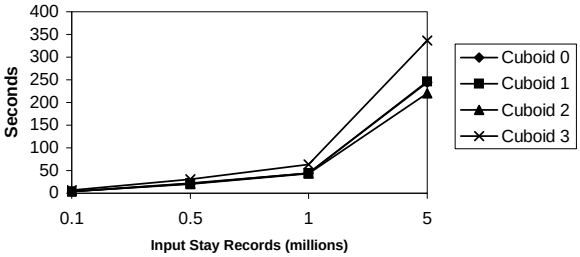

Figure 6. Construction Time. $\mathcal{P} = 1000$**,** $\mathcal{B} = (500, 150, 40, 8, 1)$**,** $k = 5$**.**

it makes little difference if we use a map or not.

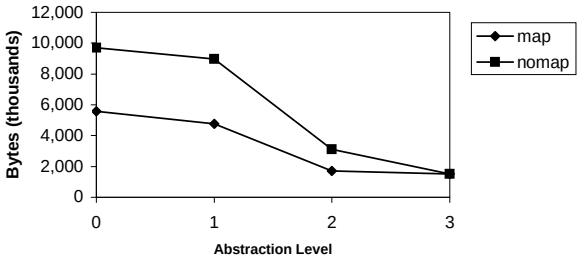

Figure 5. Compression vs. Abstraction Level. $\mathcal{P} = 1000$**,** $\mathcal{B} = (500, 150, 40, 8, 1)$**,** $k = 5$**,** $\mathcal{N} = 1,000,000$**.**

Figure 6 shows the time to build the *RFID-Cuboids* at the same four levels of abstraction used in Figure 5. In all cases the cuboid was constructed starting from the cleansed database. We can see that cuboid construction time does not significantly increase with the level of abstraction. This is expected as the only portion of the algorithm that incurs extra cost for higher levels of abstraction is the aggregation of the info table, and in this case it contains only 1,000 entries. This is common as we expect the cleansed RFID stay table to be orders of magnitude larger than the info table. The computation of *RFID-Cuboids* can also be done from lower level cuboids instead of doing it from the cleansed database. For the cuboids 1 to 3 of Figure 6 we can obtain savings of 50% to 80% in computation time if we build cuboid i from cuboid $i - 1$.

6.3 Query Processing

A major contribution of the RFID data warehouse model is the ability to efficiently answer many types of queries at various levels of aggregation. In this section, we show this efficiency in several settings. We compare query executing under three scenarios: the first is a system that directly uses the cleansed RFID database (raw), the second one that uses the *stay* table but no *map*, it instead uses tag lists at each stay record (nomap), and the third is the *RFID-Cuboid* described in the paper using *stay* and *map* tables (map). We assume that for each of the scenarios we have a B+Tree on each of the dimensions. In the case of the map cuboid the index points to a list of gids matching the index entry. In the case of the nomap cuboid and the cleansed database the index points to the tuple $(RFID\ tag, record\ id)$. This is necessary as each RFID tag can be present in multiple records. The query answering strategy used for the map cuboid is the one presented in Algorithm 3. The strategy for the other two cases is to retrieve the $(RFID\ tag, record\ id)$ pairs matching each component of the query, intersecting them, and finally retrieving the relevant records.

For the experiments we assumed that we have a page size of 4096 bytes, and that RFID tags, record ids, and gids use 4 bytes each. We also assume that all the indices fit in memory except for the last level. For each of the experiments we generated 100 random path queries. The query specifies a product, a varying number of locations (3 on average), and a time range to enter the last stage (time_out). Semantically this is equivalent to asking "What is the average time for product X to go through locations $L_1, \ldots, L_k$ entering location L_k between times $t_1 - t_2$? ".

Figure 7 shows the effect of different cleansed database sizes on query processing. The map cuboid outperforms the cleansed database by several orders of magnitude, and most importantly the query answer time is independent of database size. The nomap cuboid is significantly faster than the cleansed data but it suffers from having to retrieve very long RFID lists for each stage. The map cuboid benefits for using very short gid lists, and using the path-dependent gid naming scheme that facilitates determining if two stay records form a path without retrieving all intermediate stages.

Figure 8 shows the effects of path bulkiness on query processing. For this experiment we set the number of stay records constant at 1 million. The bulkiness levels are the same as those used for the experiment in Figure 4. As with the compression experiment since we have more bulky paths, the map cuboid is an order of magnitude faster

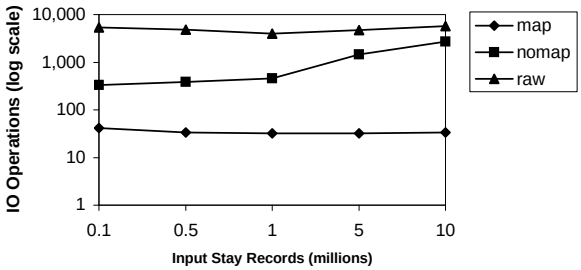

Figure 7. I/O Cost vs. Cleansed Data Size. $\mathcal{P} = 1000$**,** $\mathcal{B} = (500, 150, 40, 8, 1)$**,** $k = 5$**.**

than the cleansed RFID database. As we get less bulky paths, the benefits of compressing multiple stay records decreases until the point at which it is no better than using the cleansed database. The difference between the map and nomap cuboids is almost an order of magnitude for bulky paths, but as in the previous case, for less bulky paths the advantage of using the map decreases.

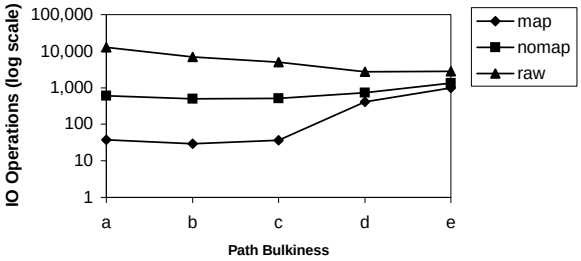

Figure 8. I/O Cost vs. Path Bulkiness. $\mathcal{P} = 1000$**,** $k = 5$**.**

7 Discussion

In this section, we discuss the related work and explore the possible extensions of the methods.

7.1 Related Work

RFID technology has been researched from several perspectives: (1) the *physics of building tags and readers* [4, 9], (2) the techniques required to guarantee *privacy and safety* [11], and (3) the *software architecture required to collect, filter, organize, and answer online queries on tags* [3, 5, 10].

The software architecture line of research is the closest to our work but differs in that it is mainly concerned with online transaction processing (OLTP) but not OLAP-based data warehousing. [10] presents a comprehensive framework for online management of RFID data, called the EPC Global Network, which is composed of servers that collect data from readers and contact computers responsible for keeping detailed information on each tag through a directory service. The system answers queries on tags through an XML language called PML [5]. [3] presents an overview of RFID data management from a high-level perspective and it introduces the idea of an online warehouse but without going into the detail at the level of data structure or algorithm.

An RFID data warehouse shares many common principles with the traditional data cube [1, 2, 7]. They both aggregate data at different levels of abstraction in multi-dimensional space. Since each dimension has an associated concept hierarchy, both can be (at least partially) modelled by a *Star* schema. The problem of deciding which *RFID-Cuboids* to construct in order to provide efficient answers to a variety of queries specified at different abstraction levels is analogous to the problem of partial data cube materialization studied in [8, 12]. However, *RFID-Cuboid* differs from a traditional data cube in that it also models *object transitions* in multi-dimensional space, which is crucial to answer queries related to object movements/transitions as demonstrated in our analysis.

7.2 Possible Extensions

Here we discuss methods for incremental update in the RFID warehouse, construction of fading RFID model, and its linkage with mining RFID data.

Incremental Update

Incremental update is crucial to RFID applications. A warehouse will receive constant updates as new tags enter the system and objects move to new locations. We can apply Algorithm 1 only to the new data to generate new *stay* and *map* tables (we need to initialize the gid naming as to not generate duplicates with the existing map). The updated cuboid will be the union of the old *stay* and *map* tables with the new ones. The main reason that we can just run the algorithm in the new data without worrying about the old data is that the updates are only for item movements with higher timestamps than we already have so they will necessarily form new nodes in the path tree.

Construction of a Fading RFID Warehouse Model

In most situations, the more remote in time or distance the data is, the less interest a user would have to study it in detail. For example, one may not be interested in the detailed tag movements among shelves if the data is years old, or thousands of miles away, or not being in the same sector (*e.g.*, manufacturer vs. store manager). This will relieve our burden to store the enormous size of historical or remote or unrelated RFID databases.

The RFID warehouse can be easily adapted to a *fading model* where remote historical or distantly located data can

be stored with low resolution (*i.e.,* not retaining such data at a low abstraction level). This can be done incrementally by further summarizing data being faded at an abstraction level higher than the recent and close-by data or by simply tossing some low-level cuboid and raising the level of the corresponding minimum abstraction level *RFID-Cuboid* for such data.

Data Mining in the RFID Warehouse

The RFID warehouse model facilitates efficient data mining since the data in the model is well structured, aggregated and organized. Take the frequent itemset mining as an example. One can easily mine frequent itemsets at high abstraction level since such data are already aggregated with count and other measures computed. The Apriori pruning can be used to prune the search along the infrequent high-level itemsets. Efficient methods, such as progressive deepening, can be further explored to reduce the search space. For example, if dairy products and meat are not sold together frequently, there is no need to drill-down to see whether milk and beef will be sold together frequently. The RFID warehouse model naturally facilitates level-wise pruning based its multi-level structure and level-wise precomputation.

8 Conclusions

We have proposed a novel model for warehousing RFID data that allows high-level analysis to be performed efficiently and flexibly in multidimensional space. The model is composed of a hierarchy of highly compact summaries (*RFID-Cuboids*) of the RFID data aggregated at different abstraction levels where data analysis can take place. Each cuboid records tag movements in the *stay*, *info*, and *map* tables that take advantage of the fact that individual tags tend to move and stay together (especially at higher abstraction levels) to collapse multiple movements into a single record without loss of information. Our performance study shows that the size of *RFID-Cuboids* at interesting abstraction levels can be orders of magnitude smaller than the original RFID database and can be constructed efficiently. Moreover, we show the power of our data structures and algorithms in efficient answering of a wide range of RFID queries, especially those related to object transitions.

Our study has been focused on efficient data warehousing and OLAP-styled analysis of RFID data. Efficient methods for a multitude of other data mining problems for the RFID data (*e.g.,* trend analysis, outlier detection, path clustering) remain open and should be a promising line of future research.

Notice that our proposal of the RFID model and its subsequent methods for warehouse construction and query analysis is based on the assumption that RFID data tend to move together in bulky mode, especially at the early stage. This fits a good number of RFID applications, such as supply chain management. However, there are also other applications where RFID data may not have such characteristics. We believe that further research is needed to construct efficient models for such applications.

References

[1] S. Agarwal, R. Agrawal, P. M. Deshpande, A. Gupta, J. F. Naughton, R. Ramakrishnan, and S. Sarawagi. On the computation of multidimensional aggregates. In *Proc. 1996 Int. Conf. Very Large Data Bases (VLDB'96)*.

[2] S. Chaudhuri and U. Dayal. An overview of data warehousing and OLAP technology. *SIGMOD Record*, 26:65–74, 1997.

[3] S. Chawathe, V. Krishnamurthy, S. Ramachandran, and S. Sarma. Managing RFID data. In *Proc. Intl. Conf. on Very Large Databases (VLDB'04)*.

[4] K. Finkenzeller. *RFID Handbook: Fundamentals and Applications in Contactless Smart Cards and Identification*. John Wiley and Sons, 2003.

[5] C. Floerkemeier, D. Anarkat, T. Osinski, and M. Harrison. PML core specification 1.0. White paper, MIT Auto-ID Center, `http://www.epcglobalinc.org/standards_technology/Secure/v1.0/PML_Core_Specification_v1.0.pdf`, 2003.

[6] E. Fredkin. Trie memory. In *Communications of the ACM*, pages 490–499, September 1960.

[7] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab and sub-totals. *Data Mining and Knowledge Discovery*, 1:29–54, 1997.

[8] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. In *Proc. 1996 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'96)*.

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

[10] S. Sarma, D. L. Brock, and K. Ashton. The networked physical world. White paper, MIT Auto-ID Center, `http://archive.epcglobalinc.org/publishedresearch/MIT-AUTOID-WH-001.pdf`, 2000.

[11] S. E. Sarma, S. A. Weis, and D. W. Engels. RFID systems, security & privacy implications. White paper, MIT Auto-ID Center, `http://archive.epcglobalinc.org/publishedresearch/MIT-AUTOID-WH-014.pdf`, 2002.

[12] A. Shukla, P. Deshpande, and J. F. Naughton. Materialized view selection for multidimensional datasets. In *Proc. 1998 Int. Conf. Very Large Data Bases (VLDB'98)*.

[13] Venture Development Corporation (VDC). `http://www.vdc-corp.com/`.