Modeling Massive RFID Datasets: A Gateway-Based Movement-Graph Approach

Hector Gonzalez, Jiawei Han, Hong Cheng, Xiaolei Li, Diego Klabjan
Department of Computer Science
University of Illinois at Urbana-Champaign

***Abstract*— Massive Radio Frequency Identification (RFID) datasets are expected to become commonplace in supply-chain management systems. Warehousing and mining this data is an essential problem with great potential benefits for inventory management, object tracking, and product procurement processes. Since RFID tags can be used to identify each individual item, enormous amounts of location-tracking data are generated. With such data, object movements can be modeled by *movement graphs*, where nodes correspond to locations, and edges record the history of item transitions between locations.**

In this study, we develop a *movement graph* model as a compact representation of RFID datasets. Since spatiotemporal as well as item information can be associated with the objects in such a model, the *movement graph* can be huge, complex, and multi-dimensional in nature. We show that such a graph can be better organized around *gateway* nodes, which serve as bridges connecting different regions of the *movement graph*. A graph-based object movement cube can be constructed by merging and collapsing nodes and edges according to an application-oriented topological structure. Moreover, we propose an efficient cubing algorithm that performs simultaneous aggregation of both spatiotemporal and item dimensions on a partitioned *movement graph*, guided by such a topological structure.

***Index Terms*— RFID, Data Warehousing, Data Models.**

I. Introduction

THE increasingly wide adoption of RFID technology by retailers to track containers, pallets, and even individual items as they move through the global supply chain, from factories in exporting countries, through transportation ports, and finally to stores in importing countries, creates enormous datasets containing rich multi-dimensional information on the movement patterns associated with objects and their characteristics. However, this information is usually hidden in terabytes of low-level RFID readings, making it difficult for data analysts to gain insight into the set of interesting patterns influencing the operation and efficiency of the procurement process. In order to realize the full benefits of detailed object tracking information, we need to develop a compact and efficient RFID cube model that provides OLAP-style operators useful to navigate through the movement data at different levels of abstraction of both spatiotemporal and item information dimensions. This is a challenging problem that cannot be efficiently solved by traditional data cube operators, as RFID datasets require the aggregation of high-dimensional graphs representing object movements, not just that of entries in a flat fact table.

We propose to model the RFID data warehouse using a *movement graph*-centric view, which makes the warehouse conceptually clear, better organized, and obtaining significantly deeper compression and performance gain over competing models in the processing of path queries. The importance of the *movement graph* approach to RFID data warehousing can be illustrated with an example.

Example. Consider a large retailer with a global supplier and distribution network that spans several countries, and that tracks objects with RFID tags placed at the item level. Such a retailer sells millions of items per day through thousands of stores around the world, and for each such item it records the complete set of movements between locations, starting at factories in producing countries, going through the transportation network, and finally arriving at a particular store where the item is purchased by a customer. The complete path traversed by each item can be quite long as readers are placed at very specific locations within factories, ships, and stores (*e.g.*, a production lane, a particular truck, or an individual shelf inside a store). Further, for each object movement, properties such as shipping cost, temperature, and humidity can be recorded.

The questions become "how can we present a clean and well organized picture about RFID objects and their movements?" and "whether such a picture may facilitate data compression, data cleaning, query processing, multi-level, multi-dimensional OLAPing, and data mining?"

Our movement-graph approach provides a nice and clean picture for modeling RFID objects at multiple levels of abstraction. And it facilitates data compression, data cleaning, and answering rather sophisticated queries, such as

- High-level aggregate/OLAP query: What is the average shipping cost of transporting electronic goods from factories in Shanghai, to stores in San Francisco in 2007? And then click to drill-down to month and see the trend.
- Path query: Print the transportation paths for the meat products from Argentina sold at L.A. on April 5, that were exposed to over 40°C heat for over 5 hours on the route.

We propose a *movement graph*-based model, which leads to concise and clean modeling of massive RFID datasets and facilitates RFID data compression, query answering, cubing, and data mining. The *movement graph* is a graph that contains a node for every distinct (or more exactly, interesting) location in the system, and edges between locations record the history of shipments (groups of items that travel together) between locations. For each shipment we record a set of interesting measures such as, travel time, transportation cost, or sensor readings like temperature or humidity. We show that this graph can be partitioned and materialized according to its topology to speedup a large number of queries, and that it can be aggregated into cuboids at different abstraction levels according to location, time, and

item dimensions, to provide multi-dimensional and multi-level summaries of item movements.

Figure 1 summarizes our proposed data warehousing architecture. We receive as input a sequence of RFID readings, and store them in database recording the path traversed by each item. Based on the structure of paths we construct a movement graph, which is partitioned along gateway nodes. Each partition is then cubed independently. And a query processing module answer OLAP queries over the aggregated movement graph.

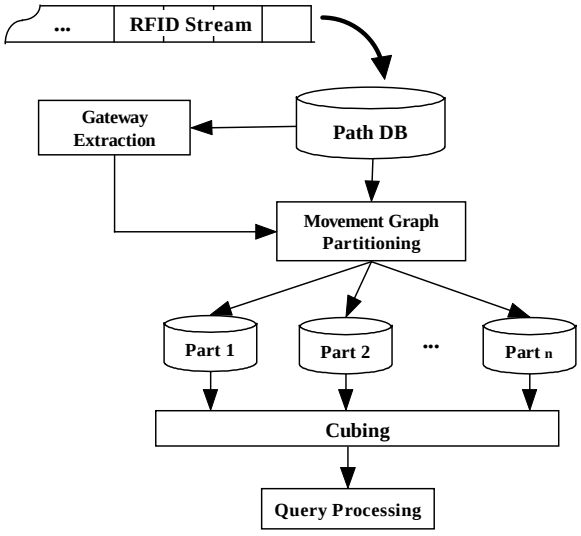

Fig. 1. RFID Warehouse Architecture

The technical contributions can be summarized as follows:

1) **Gateway-based partitioning of the movement graph**. We make the key observation that the *movement graph* can be divided into disjoint partitions, that are connected through special gateway nodes. Most paths with locations in more than one partition include the gateway nodes. For example, most items travel from China to the United States by going through major shipping ports in both countries. Gateways can be given by a user or be discovered by analyzing traffic patterns. Further, materialization can be performed for indirect edges between individual locations and their corresponding gateways and between gateways. Such materialization facilitates computing measures on the paths connecting locations in different partitions. An efficient graph partitioning algorithm, is developed, that uses the gateways to split the graph into clusters in a single scan of the path database.
2) **Redundancy elimination compression.** RFID readers provide 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 $(EPC, location, time\_in, time\_out)$. This form of compression is lossless, and it can significantly reduce the size of the raw RFID readings.
3) **Partition-based bulky movement compression**. 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 or transition record for thousands of items that stay and move together. Such record would point to a gid, which is a generalized identifier pointing to the subgroups that it contains. In global supply-chain applications, one may observe a "*merge-split*" process, *e.g.*, shipments grow in size as they approach the major ports, and then after a long distance bulky shipping, they gradually split (or even recombine) when approaching stores. We propose a partitioned map table, that creates a separate mapping for each partition, rooted at major shipping ports.
4) **Movement graph aggregation**. The movement graph can be aggregated to different levels of abstraction according to the location concept hierarchy that determines which subset of locations are interesting for analysis, and at which level. For example, the *movement graph* may only have locations inside Massachusetts, and it may aggregate every individual location inside factories, warehouses, and stores to a single node. This aggregation mechanism is very different from the one present in traditional data cubes as nodes and edges in the graph can be merged or collapsed, and the shipments along edges and their measures need to be recomputed using different semantics for the cases of node merging and node collapsing.

II. Related Work

RFID technology has been researched from several perspectives: (1) the *physics of building tags and readers* [11], [29], (2) the techniques required to guarantee *privacy and safety* [6], [14], [30], (3) the *software architecture required to collect, filter, organize, and answer online queries on tags* known and as the "EPC Global Network" which is defined by several standards including [2], [10], [26], [28], [32], (4) cleaning methods to filter noisy RFID observations [18], [22], [23], [27], (5) event processing systems on RFID data streams [3], [24], [25], [34], and (6) storage, warehousing, and mining of massive RFID datasets [8], [15]–[17].

EPC Global Inc. is a standards body that has defined specifications for many of the components of and RFID system. It starts by describing data contained in the tag [32], the communication protocols between reader and tags [21], [33], transmission of raw RFID readings by readers [28]. At a higher level, it specifies how raw readings are converted into events [2], and how such events are stored [10]. Finally, lookup of information about a particular EPC is defined by the object name service [26].

Cleaning of RFID data has been addressed both from the design of robust communication protocols, that operate at the hardware level [21], [33], and from post-processing software designed to detect incorrect readings. At the software level, the use of smoothing windows has been proposed as a method to reduce false negatives. [12] proposes the fixed-window smoothing. [22] proposes a variable-window smoothing that adapts window size dynamically according to tag detection rates, by using a binomial model of tag readings. [23] presents a framework to clean RFID data streams by applying a series of filters. [18] presents a cost-conscious cleaning framework that learns how to apply multiple techniques to increase accuracy and reduce costs.

Event processing systems is another area of research. RFID systems generate a stream of raw events, which themselves can form higher level events. Work on this area has focused on

efficient processing of very large data streams [24], [25], [34], definition of complex events, and extensions to standard query languages to account to unique temporal characteristics of RFID events [3], [34]. And [24] which copes with noise in the RFID stream by defining probabilistic events over low level events.

Closer to this work, there is significant research on storage, warehousing, and mining of massive RFID datasets. [17] introduced the problem of compressing and warehousing massive RFID datasets, it proposed the concept of the RFID-cuboid which compresses and summarizes an RFID dataset by recording information on items that stay together at a location with $stay$ records. This model takes advantage bulky shipments that are successively split into smaller shipments to compress data. This article is an extension of [17], to account for a more realistic, item movement model, where items not only split, but can merge and split multiple times as they move through a global supply chain. [25] proposes a very clever path encoding technique that improves on the encoding used in [17] to speed up query processing. At the mining end [15], [16] propose the discovery of workflows from RFID data, such workflows summarize major flow trends and significant flow exceptions at multiple abstraction levels.

Our work on RFID warehousing makes use of several traditional data warehousing techniques. An RFID data cube shares many common principles with the traditional data cube [1], [7], [19]. They both aggregate data at different levels of abstraction in multi-dimensional space, but the later is not able to handle aggregation of complex trajectory data. And the problem of RFID-cuboid materialization is analogous to the problem of partial data cube materialization studied in [20], [31]. Efficient computation of data cubes, has been a major concern for the research community and many of the techniques [4], [35], [36] can be applied to speedup construction of an RFID data cube.

III. RFID Data

In this section we give a brief introduction to data generation in RFID applications, and explain the difficulties of applying traditional data warehousing models to such data.

A. Data Generation

An RFID object tracking system is composed of a collection of RFID readers scanning for tags at periodic intervals. Each such reader is associated with a location, and generates a stream of time-ordered tuples of the form $(EPC,\ location,\ time)$ where, EPC[1] is a unique 96 bit electronic product code associated with a particular item, $location$ is the place where item was detected, and $time$ is the time when the detection took place. Significant data compression is possible by merging all the readings for an item that stays at a location for a period of time, into a tuple of the form $(EPC,\ location,\ time\_in,\ time\_out)$ where, $time\_in$ is the time when the item identified by EPC entered $location$, and $time\_out$ is the time when it left.

By sorting tag readings on EPC we can generate a *path database*, where we store the sequence of locations traversed by each item. Entries in the path database are of the form: $(EPC,\ (l_1, time\_in_1, time\_out_1)\ (l_2, time\_in_2, time\_out_2)\ ...\ (l_k, time\_in_k, time\_out_k)$. Table I presents an example path database for six items identified with tags $t1$ to $t6$, traveling through locations A, B, C, D, G_1, G_2, F, I, and J.

[1]We use EPC and tag interchangeably in the article

In addition to the location information collected by RFID readers, an RFID application has detailed information on the characteristics of each item in the system, such information can be represented with tuples of the form $(EPC,\ d_1,\ d_2,\ ...,\ d_m)$, where each d_i is a particular value for dimension D_i, typical dimensions could be $product\ type$, $manufacturer$, $weight$, or $price$. In many cases we can also have extra information describing measurements or properties collected during item shipments, this data has the form $(from,\ to,\ t_1,\ t_2,\ tag\_list : measure\_list)$ where, $from$ and to are the initial and final locations of the shipment, t_1 and t_2 are the starting and ending time of the shipment, tag_list is the set of items transported, and $measure\_list$ describes properties such as temperature, humidity, or shipping cost.

Tag	Path
t1	$(A,1,2)(D,4,5)(G_1,6,8)(G_2,9,10)(F,11,12)(J,14,17)$
t2	$(A,2,3)(D,4,5)(G_1,6,8)(G_2,9,10)(F,11,12)(K,16,18)$
t3	$(A,2,3)(D,4,6)$
t4	$(B,1,2)(D,3,5)(G_1,6,7)(G_2,9,10)(I,11,13)(K,14,15)$
t5	$(C,1,3)(E,4,5)(G_1,6,7)(G_2,9,10)(I,11,14)$
t6	$(A,3,3)(B,4,5)(I,10,11)$

TABLE I
AN EXAMPLE PATH DATABASE

B. Data Cubing Challenges

The path nature of RFID data makes it hard to incorporate into a traditional data cube, while preserving its structure. 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 dimensions. 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. The problem could be solved by increasing the number of dimensions, we could use a separate dimensions for distinct path segments; cells would then contain multi-location aggregates. The problem with this approach is that, as we increase the number of dimensions, the size of the data cube grows exponentially. We need a model capable of aggregating RFID data concisely while preserving its path-like structure for OLAP analysis.

IV. Gateway-Based Movement Graph

Among many possible models for RFID data warehouses, we believe the gateway movement graph model not only provides a concise and clear view over the movement data, but also facilitates data compression, querying, and analysis of massive RFID datasets (which will be clear later).

Definition 4.1: A movement graph $G(V, E)$ is a directed graph representing object movements; V is the set of locations, E is the set of transitions between locations. An edge $e(i, j)$ indicates

that objects moved from location v_i to location v_j. Each edge is annotated with the history of object movements along the edge, each entry in the history is a tuple of the form (t_{start}, t_{end}, tag_list : $measure\_list$), where all the objects in tag_list took the transition together, starting at time t_{start} and ending at time t_{end}, and $measure\_list$ records properties of the shipment.

Figure 2 presents the *movement graph* for the path database in Table I. We have a single node for each location, and there is an edge between locations if there is a direct transition between them in the path database, e.g., in paths 1, 2, and 3 items moves directly between A and D and thus the edge $e(A, D)$. We will explain later in the section the meaning of the dotted circles dividing the nodes in the movement graph.

In a global supply chain it is possible to identify important gateway locations, that serve to connect remote sets of locations in the transportation network. Gateways generally aggregate relatively small shipments from many distinct, regional locations into large shipments destined for a few well known remote locations; or they distribute large shipments from remote locations into smaller shipments destined to local regional locations. These special nodes are usually associated with shipping ports, *e.g.*, the port in Shanghai aggregates traffic from multiple factories, and makes large shipments to ports in the United States, which in turn split the shipments into smaller units destined for individual stores. The concept of gateways is important because it allows us to naturally partition the *movement graph* to improve query processing efficiency and reduce the cost of cube computation.

Gateways can be categorized into three classes: *Out-Gateways, In-Gateways*, and In-Out-Gateways, as described below.

Out-Gateways: In the supply chain it is common to observe locations, such as ports, that receive relatively low volume shipments from a multitude of locations and send large volume shipments to a few remote locations. For example, a port in Shanghai may receive products from a multitude of factories and logistics centers throughout China to later send the products through ship to a port in San Francisco. We call this type of node an *Out-Gateway*, and it is characterized by: (i) low ratio of average incoming shipment size to average outgoing shipment size, (ii) high ratio of the number of incoming to outgoing edges, and (iii) high centrality, in the sense that most shipments can only reach remote locations in the movement graph by going through an *Out-gateway*. Figure 3-a presents an *Out-gateway*. For our running example, Figure 2, location G_1 is an *Out-gateway*.

In-Gateways: *In-gateways* are the symmetric complement of *Out-Gateways*, they are characterized by: (i) high ratio of average incoming shipment size to average outgoing shipment size, (ii) low ratio of the number of incoming to outgoing edges, and (iii) high centrality, in the sense that most shipments can only enter remote locations in the movement graph by going through an *in-gateway*. An example of an *In-Gateway* may be sea port in New York where a large volume of imported goods arrive at the United States and are redirected to a multitude of distribution centers throughout the country before reaching individual stores. Figure 3-b presents an example *Out-gateway*. For our running example, Figure 2, location G_2 is an *In-gateway*.

In-Out-Gateways: *In-Out-gateways* are the locations that serve as both *In-gateways* and *Out-gateways*. This is the case of many ports that may for example serve as an *In-gateway* for raw materials being imported, and an *Out-gateway* for manufactured goods being exported. Figure 3-c presents such an example. It is possible to split an *In-Out-gateway* into separate In- and Out-gateways by matching incoming and outgoing edges carrying the same subset of items into the corresponding single direction traffic gateways.

Notice that gateways may naturally form hierarchies. For example, one may see a hierarchy of gateways, *e.g.*, country level sea ports →region level distribution centers →state level hubs.

V. Data Compression

A. Redundancy Elimination Compression

RFID data contains large amounts of redundancy. Each reader scans for items at periodic intervals, and thus generates hundreds or even thousands of duplicate readings for items in its range, that are not moving. For example, if a pallet stays at a warehouse for 7 days, and the reader scans for items every 30 seconds, there will be 20,160 readings of the form (EPC, $warehouse$, $time$). We could compress all these readings, without loss of information, to a single tuple of the form (EPC, $warehouse$, $time\_in$, $time\_out$), where $time\_in$ is the first time that the EPC was detected in the warehouse, and $time\_out$ the last one.

Redundancy elimination 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.

B. Bulky Movement Compression

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.

The process of selecting the most efficient grouping for items, both in terms of compression and query processing, depends on the movement graph topology.

1) Split-Only Model: In some applications, the movement graph presents a tree-like structure, with a few factories near the root, warehouses and distribution centers in the middle, and a large number of individual stores at the leaves. In such topology, it is common to observe items moving in large groups near the factories, and splitting into smaller groups as they approach individual stores. We say that movement graphs with this topology present an *Split-Only* model of object movements.

In the *Split-Only* model, we can gain significant compression by creating a hierarchy of $gids$, rooted at factories where items move in the largest possible groups, and pointing to successively smaller

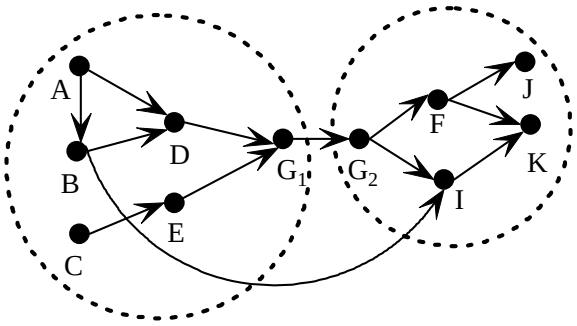

Fig. 2. An example movement graph

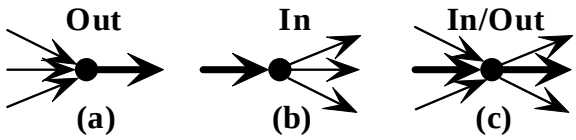

Fig. 3. Three types of gateways

groups as items move down the supply chain. In this model a single grouping schema provides good compression, because the basic groups, in which objects move, are preserved throughout the different locations, *i.e.*, the smallest groups that reach the stores are never shuffled, but are preserved all the way from the Factory. In the next section we will present a more general model that can accommodate both split and merging of groups.

2) Merge-Split Model: A more complex model of object movements is observed in a global supply chain operation, where items may merge, split, and groups of items can be shuffled several times. One such case is when items move between exporting and importing countries. At the exporting country, items merge into successively large groups in their way from factories to logistic centers, and finally to large shipping ports. In the importing country, the process is usually reversed, items split into successively smaller groups as they move from the incoming port, to distribution centers, and all the way to individual stores. We say that movement graphs with this topology present an *Merge-Split* model of object movements.

A single object grouping model, such as the one used in a *Split-Only* model would not be optimal when groups of items can both split and merge. A better option is to partition the movement graph around gateways, and define an item grouping model at the partition level. For example, the exporting country would get a hierarchy of groups rooted at the port and ending at the factories, while the importing country will have a separate hierarchy rooted at the port and ending at the individual stores. Using a single grouping for both partitions has the problem that each group would have to point to many small subgroups, or even just individual items, that are preserved throughout the entire supply chain, after multiple operations of merge, split, and shuffle. Separate groupings prevents this problem by requiring bulky movement only at the partition level, and allowing for merge, split, and even shuffling of items without loss of compression.

C. 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. This type of compression as opposed to the previous two compression methods is lossy, because once we aggregate the data at a high level of abstraction, *e.g.*, time aggregated from second to hour, we can not ask queries for any level below the aggregated one.

There are two types of data generalization. Item based, which is the same encountered in traditional data cubes, and does not involve spatio-temporal dimensions. And, path based, which is unique to RFID datasets.

Path Level Generalization

A new type of data generalization, not present in traditional data cubes, is that of merging and collapsing path stages according to time and location concept hierarchies. 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.

Another type of path generalization, is that of expanding different types of locations to different levels of abstraction depending on the analysis task. For example, a transportation manager my want to collapse all movements inside stores, and warehouses, while expanding movements within trucks and transportation centers to a very detailed level. On the other hand, store managers may want to collapse all object movements outside their particular stores.

An important difference between path level generalization and the more conventional data cube generalization along concept hierarchies, is that in path level aggregation we need to preserve the path structure of the data, *i.e.*, we need to make sure that the new times, locations, and transitions are consistent with the original data.

VI. Movement Graph Partitioning

In this section we discuss the methods for identifying gateways, partitioning based on the *movement graph*, and associating partitions to gateways.

A. Gateway Identification

In many applications it is possible for data analysts to provide the system with the complete list of gateways, this is realistic in a typical supply-chain application where the set of transportation ports is well known in advance, *e.g.*, Walmart knows all the major ports connecting its suppliers in Asia to the entry ports in the United States. In some other cases, we need to discover gateways automatically. We can use existing graph partitioning techniques such as *balanced minimum cut* or *average minimum cut* [9], to find a small set of edges that can be removed from the graph so that the graph is split into two disconnected components; such edges will typically be associated with the strong traffic edges of in- or out- gateways. Gateways could also be identified by using the concept of betweenness and centrality in social network

analysis as they will correspond to nodes with high betweenness as defined in [13] and we can use an efficient algorithm such as [5] to find them.

Here we propose a simple but effective approach to discover gateways, that works well for typical supply-chain operations where gateways have strong characteristics that are easy to identify. We can take a *movement graph*, and rank nodes as potential gateways based on the following observations: (i) a large volume of traffic goes through gateway nodes, (ii) gateways carry *unbalanced traffic*, *i.e.*, incoming and outgoing edges carrying the same tags but having very different average shipment sizes, and (iii) gateways split paths into largely disjoint sets of nodes that only communicate through the gateway. The algorithm can find gateways by eliminating first low traffic nodes, and then the nodes with balanced traffic, *i.e.*, checking the number of incoming and outgoing edges, and the ratio of the average incoming (outgoing) shipment sizes vs. the average of the outgoing (incoming) shipment sizes. Finally, for the edges that pass the above two filters, check which locations split the paths going through the location into two largely disjoint sets. That is, the locations in paths involving the gateway can be split into two subsets, locations occurring in the path before the gateway and those occurring in the path after the gateway.

B. Partitioning Algorithm

The movement graph partitioning problem can be framed as a traditional graph clustering problem and we could use techniques such as spectral clustering [9], [20]. But for the specific problem of partitioning supply-chain *movement graphs* we can design a less costly algorithm that takes advantage of the topology of the graph to associate locations to those gateways to which they are more strongly connected.

The key idea behind the partitioning algorithm is that in the movement graph for a typical supply chain application locations only connect directly (without going through another gateway) to a few gateway nodes. That is, very few items in Europe reach the major ports in the United States without first having gone through Europe's main shipping ports. Using this idea, we can associate each location to the set of gateways that it directly reaches (we use a frequency threshold to filter out gateways that are reached only rarely), when two locations l_i and l_j have a gateway in common we merge their groups into a single partition containing the two locations and all their associated gateways. We repeat this process until no additional merge is possible. At the end we do a postprocessing step where we associate very small partitions to the larger partition to which it most frequently directly connects to.

Analysis. Algorithm 1 presents the details of *movement graph* graph partitioning given a set of gateways. In a single scan of the path database, we compute statistics on the traffic from each node to the different gateways. We then go through the list of locations merging sets of locations that share common gateways. Finally, we merge small clusters into larger ones. This algorithm scales linearly with the size of the path database, linearly with the number of nodes in the *movement graph*, and quadratically with the number of gateways in the *movement graph*. We can further speed up the algorithm by running it on a random sample of the original database instead of running it on the full data. This is possible because the structure of the supply chain is usually fairly stable over time, and a representative random sample is enough to capture the topology of the graph.

Algorithm 1 Movement graph partitioning

Input: $G(V, E)$: a movement graph, $W \subset V$: the set of gateways, D: a path database, min_nodes: min. # of vertices per partition, $min\_connectivity$: min. # of paths to gateway.
Output: A partition of V into $V_1, \ldots, V_k$ s.t. $V_i \bigcap V_j = \phi, \forall i \neq j$.
Method:

1: Let $\mathcal{C}$ be a connection matrix, with entries $\mathcal{C}[l_i, g_j]$ that indicate the number of times that location l_i connects to gateway g_j. Initialize every entry in $\mathcal{C}$ to 0.
2: **for** each path p in D' **do**
3: **for** each location l in p **do**
4: $\mathcal{C}[l, g] += 1$ where g is the next gateway after l in p, or g is the previous gateway before l in p.
5: **end for**
6: **end for**
7: **for** each location $l \in V$ **do**
8: $L_g =$ all gateways s.t. $\mathcal{C}[l, g] > min\_connectivity$
9: add l to the partition containing all gateways in L_g, if the gateways in L_g reside in different partitions merge the partitions and add l and the the merged partition, if no partition exists that contains any element in L_g create a new partition containing $L_g \bigcup \{l\}$.
10: **end for**
11: **for** all partitions V_i s.t. $|V_i| < min\_nodes$ **do**
12: merge V_i with the partition V_j s.t. the traffic from/to gateways in V_i to/from gateways in V_j is maximum, this can be determined using the connection matrix $\mathcal{C}$.
13: **end for**
14: return partitions $V_1, ..., V_k$

C. Handling Sporadic Movements: Virtual Gateways

An important property of gateways is that all the traffic leaving or entering a partition goes through them. However, in reality it is still possible to have small sets of sporadic item movements between partitions that bypass gateways. Such movements reduce the effectiveness of gateway-based materialization because path queries involving multiple partitions will need to examine some path segments of the graph unrelated to gateways. This problem can be easily solved by adding a special *virtual gateway* to each partition for all outgoing and incoming traffic from and to other partitions that does not go through a gateway. Virtual gateways guarantee that inter-gateway path queries can be resolved by looking at gateway-related traffic only.

For our running example, Figure 2, we can partition the *movement graph* along the dotted circles, and associate gateway G_1 with the first partition, and gateway G_2 with the second one. In this case we need to create a virtual gateway G_x to send outgoing traffic from the first partition (*i.e.* traffic from B) that skips G_1, and another virtual gateway G_y to receive incoming traffic into the second partition (*i.e.* traffic to I), that skips G_2.

VII. Storage Model

With the tremendous amounts of RFID data, it is crucial to study the storage model. We propose to use three data structures to store both compressed and generalized data: (1) an edge

table, storing the list of edges, or alternatively an stay table, storing the list of nodes (2) a map table, linking groups of items moving together, and (3) an information table, registering path-independent information related to the items in the graph. Figure 4 presents a summarized view of the proposed schema.

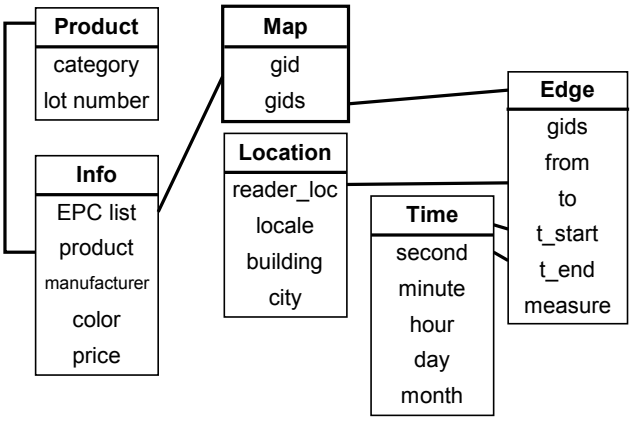

Fig. 4. RFID warehouse - logical schema

A. Edge Table

This table registers information on the edges of the *movement graph*, the format is $\langle from, to, t\_start, t\_end, direct, gid\_list, : measure\_list\rangle$, where $from$ is the originating node, to is the destination node, t_start is the time when the items departed the location $from$, t_end is the time when the items arrived at the location to, $direct$ is a boolean value that is true if the items moved directly between $from$ and to and false if intermediate nodes were visited, gid_list is the list of items that traveled together, and $measure\_list$ is a set of aggregate functions computed on the items in gid_list while they took the transition, *e.g.*, it can be count, average temperature, average movement cost, *etc.*. We will elaborate more on concept of $gids$ in the section describing the map table.

An alternative to recording edges in the graph is to record nodes, and the history of items staying at each node. The particular representation should match the application needs, if most queries ask about properties of items at a given location, materializing nodes may be better, and if most queries ask about properties during transition, materializing edges is better. And if appropriate we can materialize both.

B. Map Table

Bulky movement means a large number of items move together. A generalized identifier gid can be assigned to every group of items that moves together, which will substantially reduce the size of the tag_lists at each edge. When groups of items split into smaller groups, gid (original group) can be split into a set of children gids, representing these smaller groups. The map table contains entries of the form $\langle partition, gid, contained\_list, contains\_list\rangle$, where $partition$ is the subgraph of the *movement graph* where this map is applicable, $contained\_list$ is the list of all $gids$ with a list of items that is a superset of the items gid, $contains\_list$ is the list $gids$ with item lists that are a subset of gid, or a list of individual tags if gid did not split into smaller groups.

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.

Compression: 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).

Since real RFID datasets involve both merge and split movement models, the *Split-Only* model cannot have much sharing and compression. Here we adopt a *Merge-Split* model, where objects can be merged, shuffled, and split in many different combinations during transportation. Our mapping table takes a *gateway-centered model*, where mapping is centered around gateways, *i.e.*, the largest merged and collective moving sets at the gateways become the root gids, and their children gids can be spread in both directions along the gateways. This will lead to the maximal gid sharing and $gid_$list compression.

Query Processing: The second and the more important reason for having such a map table is the efficiency in query processing. By creating gid lists that are much shorter than EPC lists, we can compute path related queries very quickly. 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 edge records for milk between the stages and intersect the EPC lists of each. By using the map, the EPC lists can be orders of magnitude shorter and thus reduce IO costs.

C. Information Table

The information table records other attributes that describe properties of the items traveling through the edges of the *movement graph*. The format of the tuples in the information table is $\langle gid\_list, D_1, \ldots, D_n\rangle$, where gid_list is the list of items that share the same values on the dimensions D_1 to D_n, and each dimension D_i describes a property of the items in gid_list. An example of attributes that may appear in the information table could be $product$, $manufacturer$, or $weight$. Each dimension of the information table can have an associated concept hierarchy, *e.g.*, the $product$ dimension may have a hierarchy such as $EPC \rightarrow SKU \rightarrow product \rightarrow category$.

VIII. Materialization Strategy

Materialization of path segments in the *movement graph* may speedup a large number of path-related queries. Since there is an exponential number of possible path segments that can be pre-computed in a large *movement graph*, it is only realistic to partially materialize only those path segments that provide the highest expected benefit at a reasonable cost. We will develop such a strategy here.

[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

A. Path Queries

A path query requires the computation of a measure over all the items with a path that matches a given *path pattern*. It is of the form: $q \leftarrow \langle \sigma_c info,\ path\_expression,\ measure \rangle$, where $\sigma_c info$ is a selection on the information table that retrieves the relevant items for analysis; $path\_expression$ is a sequence of stage conditions on location and time that should appear in every path, in the given order but possibly with gaps; and $measure$ is a function to be computed on the matching paths. An example path query may be $\sigma_c info$ = {product = meat, sale_date = 2006}, $path\_expression$ = {Argentina farm A, San Mateo store S}, and $measure = average\ temperature$, which asks for the average temperature recorded for each meat package, traveling from a certain farm in Argentina to a particular store in San Mateo.

There may be many strategies to answer a path query, but in general we will need to retrieve the appropriate tag lists and measures for the edges along the paths involving the locations in the *path expression*; retrieve the tag list for the items matching the info selection; intersect the lists to get the set of relevant tags; and finally, if needed, retrieve the relevant paths to compute the measure.

B. Path-segment Materialization

We can model path segments as indirect edges in the *movement graph*. For example, if we want to pre-compute the list of items moving from location l_i to location l_j through any possible path, we can materialize an edge from l_i to l_j that records a history of all tag movements between the nodes, including movements that involve an arbitrary number of intermediate locations. Indirect edges are stored in the same format as direct ones, but with the flag $direct$ set to $false$.

The benefit of materializing a given indirect edge in the *movement graph* is proportional to the number of queries for which this edge reduces the total processing cost. Indirect edges, involved in a path query, reduce the number of edges that need to be analyzed, and provide shorter tag lists that are faster to retrieve and intersect. In order for an indirect edge to help a large number of queries, it should have three properties: (1) carry a large volume of traffic, (2) be part of a large portion of all the paths going from nodes in one partition of the graph to nodes in any other partition, and (3) be involved directly or indirectly in a large number of path queries. The set of edges that best match these characteristics are the following.

1) Node-to-gateway: . In supply-chain implementations it is common to find a few well defined *Out-gateways* that carry most of the traffic leaving a partition of the graph where items are produced, before reaching a partition of the graph where items are consumed. For example, products manufactured in China destined for exports to the United States leave the country through a set of ports. We propose to *materialize the (virtual) edges from every node to the Out-gateways that it first reaches*. Such materialization, for example, would allow us to quickly determine the properties of shipments originating at any location inside China and leaving the country.

2) Gateway-to-node: . Another set of important nodes for indirect edge materialization are *In-gateways*, as most of the item traffic entering a region of the graph where items are consumed has to go through an *In-gateway*. For example, imported products coming into the United States all arrive through a set of major ports. When we need to determine which items sold in the U.S. have paths that involve locations in foreign countries, we can easily get this information by pre-computing the list of items that arrived at the location from each of the *In-gateways*. We propose to *materialize all the (virtual) edges from an* In-gateway *to the nodes that it reaches without passing through any other gateway*.

3) Gateway-to-gateway: . Another interesting set of indirect edges to materialize are *the ones carrying inter-gateway traffic*. For example, we want to pre-compute which items leaving the Shanghai port finally arrive at the New York port. The benefit of such indirect edge is twofold: First, it aggregates a large number possible paths between two gateways and pre-computes important measures on the shipments; and second, it allows us to quickly determine which items travel between partitions.

Lemma 8.1: A *movement graph* with k partitions, $p_1, ..., p_k$, each partition i with p_i^n nodes and p_i^g gateways, will require the materialization of a number of indirect edges that is bounded by $\sum_{i=1}^{k} (p_i^n \times p_i^g) + \sum_{i \neq j} p_i^g \times p_j^g$

Proof. For each node in each partition we will materialize at most p_i^g indirect edges, this is when the node has traffic to or from every gateway, the maximum number of node to gateway edges is then $\sum_{i=1}^{k} p_i^n \times p_i^g$, the maximum number of gateway to gateway edges occurs when every gateway has traffic to every other gateway for a maximum number of gateway to gateway edges that is $\sum_{i \neq j} p_i^g \times p_j^g$ ∎

The implication of lemma 8.1 is that the size of our materialization scheme is small in size, especially when compared to a full materialization model in which we compute direct edges between every pair of nodes in which case we would require $(\sum_{i=1}^{k} p_i^n)^2$ additional edges. In practice the overhead of gateway materialization is usually smaller than the bound found in lemma 8.1, the reason is that nodes in a partition will tend to associate with a single gateway, and partitions will connect to a small subset of other partitions.

Lemma 8.2: Given a *movement graph* with k partitions, $p_1, \ldots, p_k$, each partition i with p_i^n nodes and p_i^g gateways, any pairwise path query q involving a path expression with two nodes (n_i, n_j), where $n_i \in p_i$, $n_j \in p_j$, can be answered by analyzing at most $p_i^g + p_j^g + p_i^g \times p_j^g$ edges.

Proof. In the worst case traffic can travel from node n_i to node n_j through all the gateways in partition p_i and all the ones in partition p_j so we need to analyze at most $p_i^g + p_j^g$ node to/from gateway indirect edges, in the worst case traffic can travel between any pair of gateways in p_i and p_j for a maximum number of inter-gateway indirect edges of $p_i^g \times p_j^g$. ∎

Lemma 8.2 provides a bound on the cost of query processing when gateway materialization has been implemented.

In general, when we need to answer a path query involving all paths between two nodes, we need to retrieve all edges between the nodes and aggregate their measures. This can be very expensive if the locations are connected by a large number of distinct paths, which is usually the case when nodes are in different partitions of the graph. By using gateway materialization we reduce this cost significantly as remote nodes can always be connected through a few edges to, from, and between gateways.

IX. RFID Cube

So far we have examined the *movement graph* at a single level of abstraction. Since items, locations (as nodes in the graph), and the history of shipments along each edge all have associated

concept hierarchies, aggregations can be performed at different levels of abstraction. We propose a data cube model for such *movement graphs*. The main difference between the traditional cube model and the *movement graph* cube is that the former aggregates on simple dimensions and levels but the latter needs to aggregate on path-dimensions as well, which may involve path collapsing as a new form of generalization. In this section we develop a model for *movement graph* cubes and introduce an efficient algorithm to compute them.

A. Movement Graph Aggregation

With a concept hierarchy associated with locations, a path can be aggregated to an abstract location by aggregating each location to a generalized location, collapsing the corresponding movements, and rebuilding the *movement graph* according to the new path database.

Location aggregation. We can use the location concept hierarchy to aggregate particular locations inside a store to a single store location, or particular stores in a region to a single region location. We can also completely disregard certain locations not interesting for analysis, *e.g.*, a store manager may want to eliminate all factory-related locations from the *movement graph* in order to see a more concise representation of the data. Figure 5 presents a hierarchy of locations, where grayed nodes represent interesting location levels. In this example, we are interested in transportation locations at the lowest level, but all store locations are collapsed into a single node.

Figure 6 presents a *movement graph* and some aggregations. All the locations are initially at the lowest abstraction level. By generalization, the transportation-related locations are collapsed into a single node $Transportation$, and the store-related locations into a single node $Store$ (shown as dotted circles). Then the original single path: $Factory \rightarrow Dock \rightarrow Hub \rightarrow Backroom \rightarrow shelf$ is collapsed to the path $Factory \rightarrow Transportation \rightarrow Store$ in the aggregated graph. If we completely remove transportation locations, we will get the path $Factory \rightarrow Store$.

Edge aggregation semantics. From the point of view of the edge table, graph aggregation corresponds to merging of edge entries, but it is different from regular grouping of fact table entries in a data cube because collapsing paths will create edges that did not exist before, and some edges can be completely removed if they are not important for analysis. In a traditional data cube, fact table entries are never created or removed, they are just aggregated into larger or smaller groups. For example, in Figure 6, if we remove all transportation-related locations, a new edge ($Factory$, $Store$) will be created, with all the edges to and from transportation locations removed.

Graph aggregation involves different operations over the gid_lists at each edge, when we remove nodes we need to intersect gid_lists to determine the items traveling through the new edge, but when we simply aggregate locations to higher levels of abstraction (without removing them) we need instead to compute the union of the gid_lists of several edges. For example, looking at Figure 6 in order to determine the gid_list for the edge ($Factory$, $Store$) we need to intersect the gid_lists of all outgoing edges from the node $Factory$ with the incoming edges to the node $Store$; on the other hand if we aggregate transportation locations to a single node, in order to determine the gid_list for the edge ($Transportation$, $Store$), we need to union the gid_lists of the edges ($Hub, Store$) and ($Weighting, Store$).

B. Cube Structure

Fact table. The fact table contains information on the *movement graph*, and the items aggregated to the minimum level of abstraction that is interesting for analysis. Each entry in the fact table is a tuple of the form: $\langle from,\ to,\ t\_start,\ t\_end,\ d_1,\ d_2,\ \ldots,\ d_k : gid\_list : measure\_list\rangle$, where gid_list is list of $gids$ that contains all the items that took the transition between $from$ and to locations, starting at time t_in and ending at time t_out, and all share the dimension values $d_1, \ldots, d_k$ for dimensions $D_1, \ldots, D_k$ in the info table, $measure\_list$ contains a set of measures computed on the $gids$ in gid_list.

Measure. For each entry in the fact table we register the gid_list corresponding to the tags that match the dimension values in the entry. We can also record for each gid in the list a set of measures recorded during shipping, such as average temperature, total weight, or count. We can use the gid_list to quickly retrieve those paths that match a given *slice*, *dice*, or *path selection* query at any level of abstraction. When a query is issued for aggregate measure that are already pre-computed in the cube, we do not need to access the path database, and all query processing can be done directly on the aggregated *movement graph*. For example, if we record $count$ as a measure, any query asking for counts of items moving between locations can be answered directly by retrieving the appropriate cells in the cube. When a query asks for a measure that has not been pre-computed in the cube, we can still use the aggregated cells to quickly determine the list of relevant $gids$ and retrieve the corresponding paths to compute the measure on the fly.

RFID-cuboids. A cuboid in the RFID cube resides at a level of abstraction of the location concept hierarchy, a level of abstraction of the time dimension, and a level of abstraction of the *info dimensions*. *Path aggregation* is used to collapse uninteresting locations, and *item aggregation* is used to group-related items. *Cells* in a *movement graph* RFID-cuboids group both items, and edges that share the same values at the RFID-cuboid abstraction level.

It is possible for two separate RFID-cuboids to share a large number of common cells, namely, all those corresponding to portions of the *movement graph* that are common to both RFID-cuboids, and that share the same *item aggregation* level. A natural optimization is to compute cells only once. The size of the full *movement graph* cube is thus the total number of distinct cells in all the RFID-cuboids. When materializing the cube or a subset of RFID-cuboids we compute all relevant cells to those RFID-cuboids without duplicating shared cells between RFID-cuboids.

C. Cube Computation

In this section we introduce an efficient algorithm, that in a single scan of the fact table, simultaneously computes the set of interesting RFID-cuboids, as defined by the user or determined through selective cube materialization techniques such as those proposed in [20]. The computed RFID-cuboids can then be used to answer the most common OLAP and path query operations efficiently, and can also be used to quickly compute non-materialized RFID-cuboids on the fly.

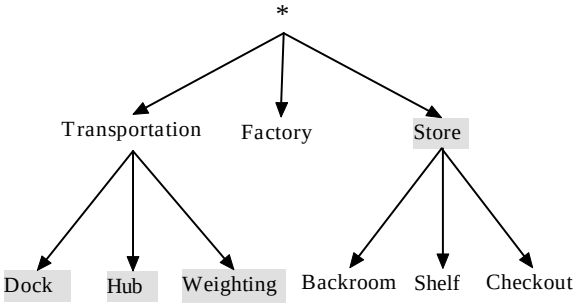

Fig. 5. Location concept hierarchy

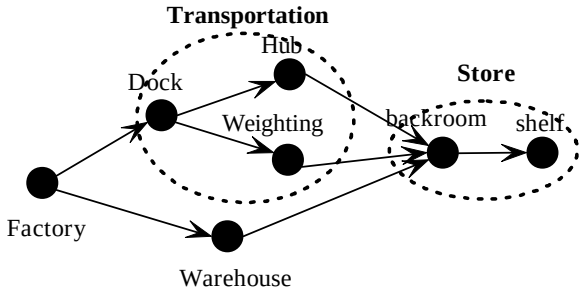

Fig. 6. Graph aggregation

Partition-based aggregation. We will aggregate each partition of the graph independently, *i.e.*, paths will be divided into disjoint segments according to the partitions defined in the *movement graph*, and each segment in the path will be aggregated independently, without merging locations from separate segments. This technique guarantees that for any RFID-cuboid we can still use the gateway-based materialization to improve query performance. If we need to compute inter-partition aggregation, it can be done at runtime by using the best available RFID-cuboids from each partition, and computing required aggregation on top of those at runtime.

Algorithm 2 presents an efficient cubing algorithm that does simultaneous aggregation of every interesting cell in parallel, with a single scan of the path database.

Path prefix tree. The algorithm first constructs a prefix tree for each partition of the *movement graph*. For this purpose paths are divided into disjoint fragments separated by gateways, in the case when locations belonging to separate partitions appear in the same path fragment we separate them with a virtual gateway. Each path is converted into a sequence of edges of the form ($from$, to, t_in, t_out), the first edge of every path has a $from$ location equal to the special symbol $\vdash$, and the last edge in the path has a to location that is the special symbol $\dashv$. After the prefix trees have been constructed, we assign a unique gid to the items aggregated in each node of the tree, that share the same values on all *item dimensions*. Figure 7 presents the path prefix tree computed on the path database of Table I, and the first partition in Figure 2. Notice that we have created the virtual gateway G_x and added it to the prefix path.

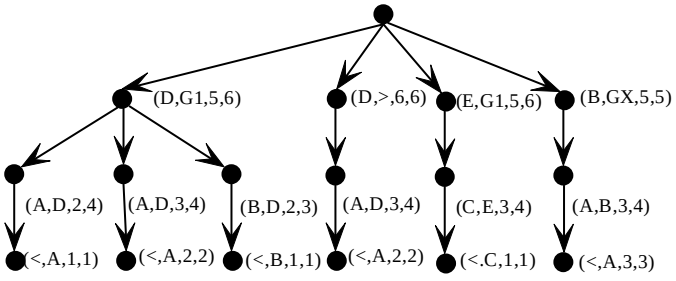

Fig. 7. Prefix tree partition 1

Cuboid materialization. After building the path prefix trees, we are ready to start building the relevant set of RFID-cuboids, this is done by traversing each prefix tree, one branch at a time, and generating all possible edges from the branch, including: (1) direct edges that correspond to a single node in the tree, (2) edges generated by performing *path collapsing* to all interesting location levels of abstraction, and (3) indirect edges from each location in the path to a gateway. All the cells involving these edges are then updated to include the $gids$ associated with the edge and their measures adjusted. We also do aggregation of RFID-cuboids that include only *item dimensions* (*i.e.*, location and time dimensions are aggregated to *all*) by aggregating the *info* entry for the relevant $gids$ to the levels indicated by the set of RFID-cuboids to compute. The simultaneous multi-level, multi-dimensional aggregation is similar to the aggregation done in multi-way array aggregation [36], and we can make use of very similar techniques to minimize the amount of memory required to materialize the cube; the idea is to sort the dimensions of the info table in decreasing cardinality and to build the prefix path tree in lexicographic order, so that we minimize the number of cells that need to be kept in main memory.

Analysis. Algorithm 2 can be implemented efficiently, as it requires a single scan of the path database, which is compressed into a compact prefix tree representation, which in turn is traversed only once to generate all relevant aggregated cells in parallel. It is more efficient than the cubing algorithm proposed in [17] which computes RFID-cuboids one at a time and thus misses the efficiency gains provided by shared computation of multiple path segments in parallel. [17] also incurs in the cost of maintaining a separate info and map table for every RFID-cuboid, we instead keep a single map table for all RFID-cuboids, and incorporate the item dimensions in the graph itself, so no extra overhead of accessing the info table is required when slicing on item dimensions.

X. Experimental Evaluation

In this section we report our comprehensive evaluation of the proposed model and algorithms. All the experiments were conducted on a Pentium 4 3.0 GHz, with 1.5Gb RAM, running Win XP; the code was written in C++ and compiled with Microsoft Visual Studio 2003.

A. Data Synthesis

The path databases used for performance evaluation were generated using a synthetic path generator. We first generate a random *movement graph* with 5 partitions and 100 locations, each partition has a random number of gateways. Locations inside a partition are arranged according to a *producer configuration*, where we simulate factories connecting to intermediate locations that aggregate traffic, which in turn connect to *Out-gateways*; or

Algorithm 2 Graph cube construction

Input: D: path database, P: location partitions, W: set of gateways for each partition, and $C = \{c_1, \ldots, c_m\}$: set of interesting RFID-cuboids to materialize
Output: The cells for the RFID-cuboids in C, and a map table
Method:

1: Scan D once and build a prefix tree of edges for each partition. Each path in D is broken into partitions according to P and W, if needed virtual gateways are inserted to separate partitions. Paths are aggregated to the minimum interesting abstraction level. Nodes in the prefix trees have the form ($from$, to, t_in, t_out). Prefix trees should always be rooted at the gateways.
2: **for** each prefix tree T_i **do**
3: Assign $gids$ to each node in the tree, by linking each gid to all of its direct children in the tree, and the complete list of ancestors
4: **for** each branch p in T_i **do**
5: Let p_d be the set of direct edges
6: compute p_i the set of indirect edges created by path collapsing according to the location abstraction level of RFID-cuboids in C
7: compute p_g the set of node to/from gateway edges
8: aggregate each relevant cell using all elements in p_d, p_i, and p_g.
9: aggregate cells involving only *item dimensions*
10: **end for**
11: **end for**

a *consumer configuration*, where we simulate products moving from *In-gateways*, to intermediate locations such as distribution centers, and finally to stores. We generate paths by simulating groups of items moving inside a partition, or between partitions, and going usually through gateways, but sometimes, also "jumping" directly between non-gateway nodes; we increase shipment sizes close to gateways. We control the number of items moving together by a *shipment size* parameter, which indicates the smallest number of items moving together in a partition. Each item in the system has an entry in the path database, and an associated set of *item dimensions*. We characterize a dataset by $\mathcal{N}$ the number of paths, and $\mathcal{S}$ the minimum shipment size.

In most of the experiments in this section we compare three competing models. The first model is *part gw mat*, it represents a partitioned *movement graph* where we have performed materialization of path segments to, from, and between gateways. The second model is *part gw no mat*, it represents a model where we partition the *movement graph* but we do not perform gateway materialization. The third model is *no part*, which represents a *movement graph* that has not been partitioned and corresponds to the model introduced in [17].

B. Model Size

In these experiments we compare the sizes of three models of *movement graph* materialization, and the size of the original path database *path db*. For all the experiments, we materialize the graph at the same level of abstraction as the one in the path database and thus is a lossless representation of the data.

Figure 8 presents the size of the four models on path databases with a varying number of paths. For this experiment we can clearly see that the partitioned graph without gateway materialization *part no gw mat* is always significantly smaller than the non partition model *non part*, this comparison is fair as both models materialize only direct edges. When we perform gateway materialization the size of the model increases, but the increase is still linear on the size of the *movement graph* (much better than full materialization of edges between every pair of nodes which is quadratic in the size of the *movement graph*), and close to the size of the model in [17].

Figure 9 presents the size of the map table for the partitioned *part gw mat* and non-partitioned *no part* models. The difference in size is almost a full order of magnitude. The reason is that our partition level maps capture the semantics of collective object movements much better than [17]. This has very important implications in compression power, and more importantly, in query processing efficiency.

Figure 10 presents the size of the fact table, as we vary the shipment size, under four different models *part no gw mat*, *part gw mat*, *no part*, and *path db*. We see that compression improves as we increase shipment sizes. Gateway materialization increases the size of the fact table, it is still much smaller than the original path database, except for very small shipment sizes, and it is also smaller than a non-partitioned fact table.

Figure 11 present the size of the map table, as we vary shipment size, for the *part gw mat* and *no part* models. This experiment isolates the effect on compression of the map table, due to shipment size. We can observe a very significant reduction in map size, even for small shipment sizes. This is a clear indication that partition level map tables provide a clear advantage over a global map table. This is important not only in terms of space, but also in terms of query processing, as we will see in the next section.

C. Query Processing

An important contribution of our model is efficiency in query processing. In these experiments we generate 100 random path queries that, ask for a measure on the path segments, for items matching an *item dimension* condition, that go from a single initial location to a single ending location, and that occurred within a certain time interval. We compare the partition *movement graph* with gateway materialization *part gw mat*, against the partitioned graph *part no gw mat* without gateway materialization, and the non-partitioned graph *no part*. All the queries were answering a *movement graph* at the same abstraction level as the original path database. For lack of space we restrict the analysis queries with starting and ending locations in different partitions as those are in general more challenging to answer. Based on our experiments on single partition queries, our method has a big advantage given by their compact map tables.

For the case of non-partitioned graph, we use the same query processing algorithm presented in [17]. For the case of partitioned graph without gateway materialization, we retrieve all the relevant edges from the initial node to the gateways in its partition, edges between gateways, and edges from the gateways in the ending location's partition and the location. In this case we do not perform inter-gateway join of the gid lists, but the overhead of such join can be small if we keep an inter-gateway join table, or if our measure does not require matching of the relevant edges in both partitions. For the gateway materialization case, we retrieve only relevant gateway-related edges. For this method we compute

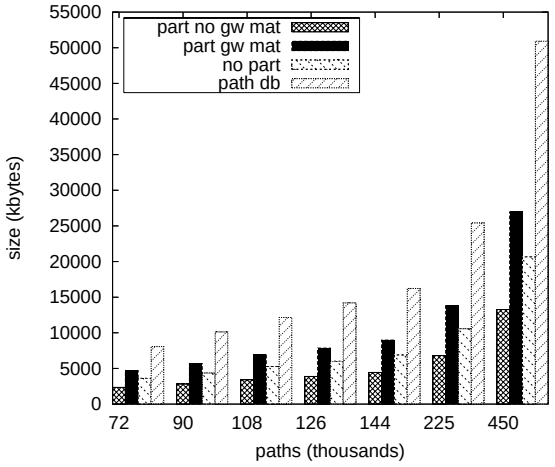

Fig. 8. Fact table size vs. Path db size ($\mathcal{S} = 300$)

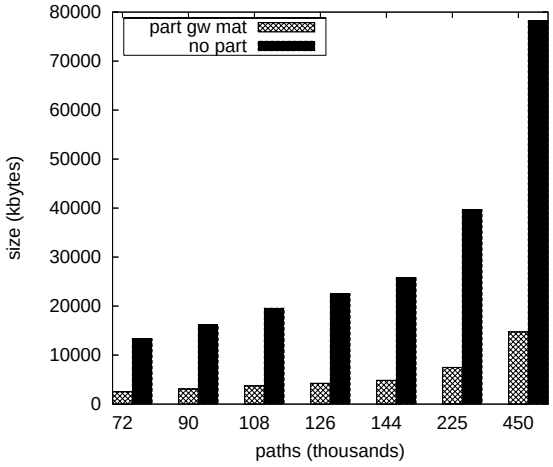

Fig. 9. Map table size vs. Path db size ($\mathcal{S} = 300$)

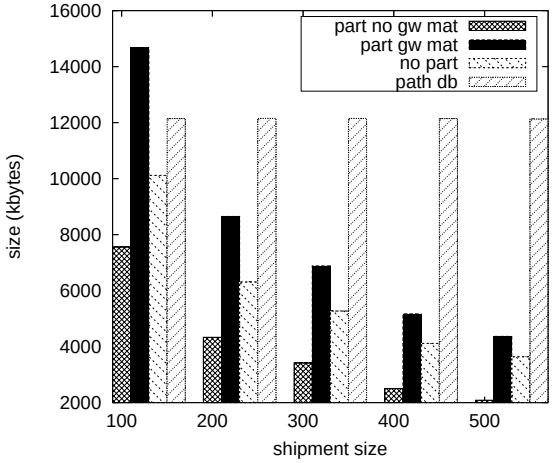

Fig. 10. Fact table size vs. Shipment size ($\mathcal{N} = 108,000$)

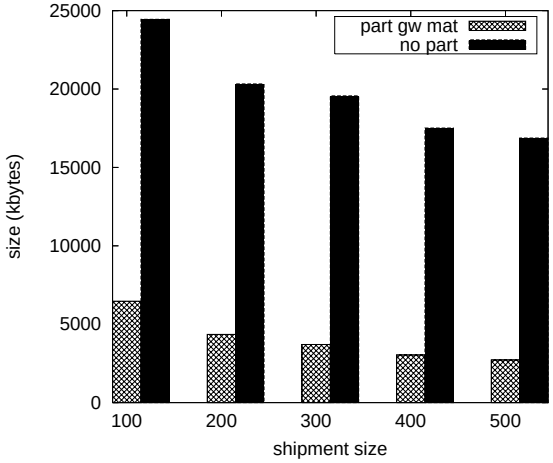

Fig. 11. Map table size vs. Shipment size ($\mathcal{N} = 108,000$)

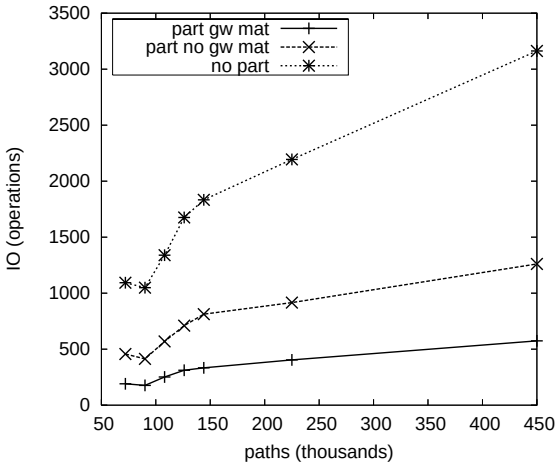

Fig. 12. Query IO vs. Path db size ($\mathcal{S} = 300$)

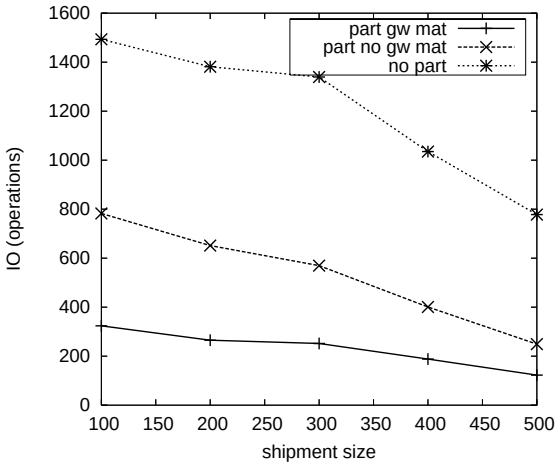

Fig. 13. Query IO vs. Shipment size ($\mathcal{N} = 108,000$)

the cost using tag lists instead of gid lists on materialized edges to and from gateways.

In Figure 12 we analyze query performance with respect to path database size. We see that the gateway-based materialization method is the clear winner, its cost is almost an order of magnitude smaller than the method proposed in [17]. We also see that our method has the lowest growth in cost with respect to database size. Figure 13 presents the same analysis but for a path database with different minimum shipment sizes. Our model is the clear winner in all cases, and as expected performance improves

with larger shipment sizes.

D. Cubing

For the cubing experiments we compute a set of 5 random cuboids, with significant shared dimensions among them, *i.e.*, the cuboids share a large number of interesting locations, and *item dimensions*. We are interested in the study of such cuboids because it captures the gains in efficiency that we would obtain if we used our algorithm to compute a full *movement graph* cube, as ancestor/descendant cuboids in the cube lattice benefit most from shared computation. Figure 14 presents the total runtime to compute 5 cuboids, we can see that *shared* significantly outperforms the level by level cubing algorithm presented in [17]. For the case when cuboids are very far apart in the lattice the shared computation has a smaller effect and our algorithm performs similarly to [17].

Figure 15 presents the total size of the cells in the 5 cuboids for the case of a partitioned graph without gateway materialization, and a non-partitioned graph. The compression advantage of our method increases for larger database sizes. This advantage becomes even more important as more cuboids are materialized. We can thus use our model to create compact *movement graphs* at different levels of abstraction, and furthermore, use them to answer queries significantly more efficiently than competing models. If we want even better query processing speed, we can sacrifice some compression and perform gateway-based materialization.

E. Partitioning

The final experiment evaluates the scalability of our *movement graph* partitioning algorithm. For this experiment we assume that the set of gateway nodes is given, which is realistic in many applications as this set is small and well-known (*e.g.*, major shipping ports). Figure 16 shows that the algorithm scales linearly with the size of the path database. And for our test cases, which are generated following the operation of a typical global supply chain operation, the algorithm always finds the correct partitions. This algorithm can partition very large datasets quickly, and at a fraction of the cost of standard graph clustering algorithms. It is important to note that for the case when we are dealing with a more general *movement graph*, that does not represent a supply chain, more expensive algorithms are likely required to find good partitions.

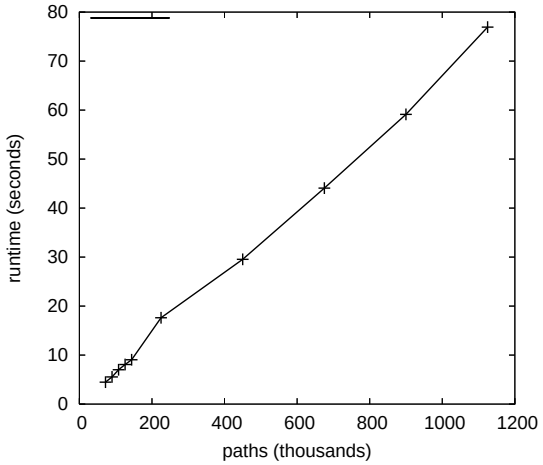

Fig. 16. Partitioning time vs. Path db size

XI. Conclusions

In this article we have introduced a new, *gateway-based movement graph model* for warehousing massive, transportation-based RFID datasets. This model captures the essential semantics of supply-chain application as well as many other RFID applications that explore object movements of similar nature. It provides a clean and concise representation of large RFID datasets. Moreover, it sets up a solid foundation for modeling RFID data and facilitates efficient and effective RFID data compression, data cleaning, multi-dimensional data aggregation, query processing, and data mining.

A set of efficient methods have been developed in this study for movement graph construction, gateway identification, gateway-based graph partitioning, efficient storage structuring, multi-dimensional aggregation, graph cube computation, and cube-based query processing. This weaves an organized picture for systematic modeling and implementation of such an RFID data warehouse. Our implementation and performance study shows that the methods proposed here are much more efficient in both storage cost, cube computation, and query processing comparing with a previous study [17] that uses a global map table without gateway-based movement graph modeling and partitioning.

The gateway-based movement graph model proposed here captures the semantics of bulky, sophisticated, but collective object movements, including merging, shuffling, and splitting processes. Its applications are not confined to RFID data sets but also extensible to other bulky object movement data. However, further study is needed to model and warehouse objects with scattered movements, such as traffic on highways where each vehicle moves differently from others.

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)*, pages 506–521, Bombay, India, Sept. 1996.

[2] The application level events (ALE) specification. Specification, EPCglobal Inc, `http://www.epcglobalinc.org/standards_technology/EPCglobal_ApplicationALE_Specification_v112-2005.pdf`, 2003.

[3] Y. Bai, F. Wang, P. Liu, C. Zaniolo, and S. Liu. RFID data processing with a data stream query language. In *Proc. 2007 Int. Conf. Data Engineering (ICDE'07)*, pages 1184–1193, Istanbul, Turkey, April 2007.

[4] K. Beyer and R. Ramakrishnan. Bottom-up computation of sparse and iceberg cubes. In *Proc. 1999 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'99)*, pages 359–370, Philadelphia, PA, June 1999.

[5] U. Brandes. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology*, 25:163–177, 2001.

[6] S. Chari, C. Jutla, J.R. Rao, and P. Rohatgi. A cautionary note regarding evaluation of aes candidates on smart-cards. In *Second Advance Encryption Standard (AES) Candidate Conference, Rome, Italy.*, 1999.

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

[8] Q. Chen, Z. Li, and H. Liu. Optimizing complex event processing over RFID data streams. In *Proc. 2008 Int. Conf. Data Engineering (ICDE'08)*, pages 1442–1444, Cancun, Mexico, April 2008.

[9] R.K. Chung. *Spectral Graph Theory*, volume 92. CBMS Regional Conference Series in Methematics, 1997.

[10] Epcis standard v. 1.0.1. Standard, EPCglobal, `http://www.epcglobalinc.org/standards/epcis/epcis_1_0_1-standard-20070921.pdf`, 2008.

[11] Klaus Finkenzeller. *RFID-Handbook, 2nd edition*. Wiley and Sons, 2003.

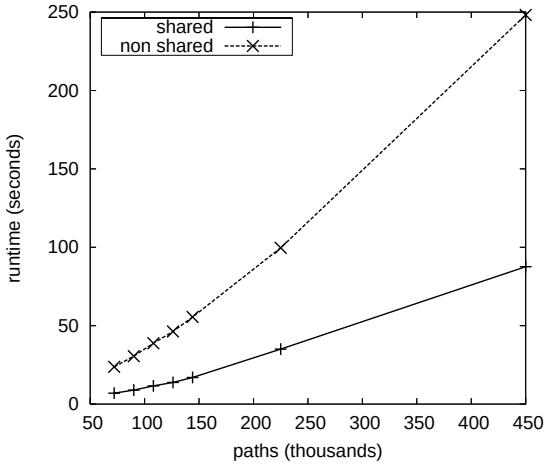

Fig. 14. Cubing time vs. Path db size ($\mathcal{S} = 300$ $\mathcal{N} = 108,000$)

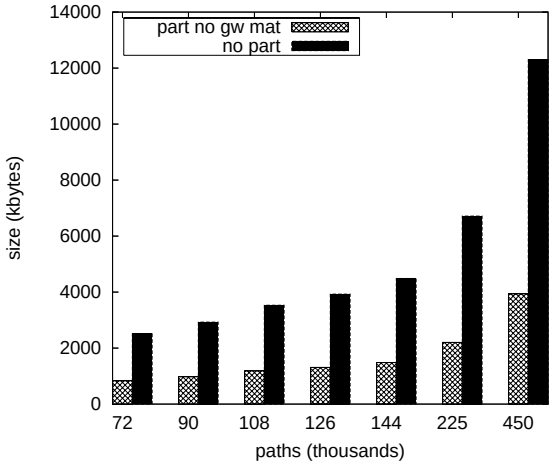

Fig. 15. Cube size vs. Path db size ($\mathcal{S} = 300$ $\mathcal{N} = 108,000$)

[12] C. Floerkemeier and M. Lampe. Issues with rfid usage in ubiquitous computing applications. In *Pervasive Computing (PERVASIVE) Lecture notes in Compute Science*.

[13] L.C. Freeman. A set of measures of centrality based on betweenness. 40:35–41, 1977.

[14] H. Gobioff, S. Smith, J.D. Tygar, and B. Yee. Smart cards in hostile environments. In *2nd USENIX Workshop on Elec. Commerce*, 1996.

[15] H. Gonzalez, J. Han, and X. Li. Flowcube: Constructuing RFID flowcubes for multi-dimensional analysis of commodity flows. In *Proc. 2006 Int. Conf. Very Large Data Bases (VLDB'06)*, Seoul, Korea, Sept. 2006.

[16] H. Gonzalez, J. Han, and X. Li. Mining compressed commodity workflows from massive rfid data sets. In *Proc. 2006 Conf. On Information and Knowledge Management (CIKM'06)*, Virginia, November 2006.

[17] H. Gonzalez, J. Han, X. Li, and D. Klabjan. Warehousing and analysis of massive RFID data sets. In *Proc. 2006 Int. Conf. Data Engineering (ICDE'06)*, Atlanta, Georgia, April 2006.

[18] H. Gonzalez, J. Han, and X. Shen. Cost-conscious cleaning of massive RFID data sets. In *Proc. 2007 Int. Conf. Data Engineering (ICDE'07)*, Istanbul, Turkey, April 2007.

[19] 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.

[20] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. In *Proc. 1996 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'96)*, pages 205–216, Montreal, Canada, June 1996.

[21] 13.56 mhz ism band class 1 radio frequency identification tag interface specification. Technical report, MIT Auto-ID Center, 2003.

[22] Shawn R. Jeffery, Minos Garofalakis, and Michael J. Franklin. Adaptive cleaning for RFID data streams. In *Proc. 2006 Int. Conf. on Very Large Data Bases (VLDB'06)*, Seoul, Korea, Sept. 2006.

[23] S. R. Jeffrey, G. Alonso, M.J. Franklin, W. Hong, and J. Widom. A pipelined framework for online cleaning of sensor data streams. In *Proc. 2006 Int. Conf. on Data Engineering (ICDE'06)*, Atlanta, Georgia, April 2006.

[24] N. Khoussainova, M. Balazinska, and D. Suciu. Peex: Extracting probabilistic events from rfid data. In *Proc. 2008 Int. Conf. Data Engineering (ICDE'08)*, pages 1480–1482, Cancun, Mexico, April 2008.

[25] C. Lee and C. Chung. Efficient storage scheme and query processing for supply chain management using RFID. In *Proc. 2008 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'08)*, pages 291–302, Vancouver, Canada, June 2008.

[26] Epcglobal object name service (ons) 1.0.1. Standard, EPCglobal, `http://www.epcglobalinc.org/standards/ons/ons_1_0_1-standard-20080529.pdf`, 2008.

[27] J. Rao, S. Doraiswamy, H. Thakar, and L.S. Colby. A deferred cleansing method for RFID data analytics. In *Proc. 2006 Int. Conf. on Very Large Data Bases (VLDB'06)*, Seoul, Korea, Sept. 2006.

[28] Reader protocol (rp) standard. Standard, EPCglobal, `http://www.epcglobalinc.org/standards/rp/rp_1_1-standard-20060621.pdf`, 2006.

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

[30] Sanjay E. Sarma, Stephen A. Weis, and Daniel W. Engels. RFID systems, security & privacy implications. White paper, Auto-ID Labs, `http://www.autoidlabs.org/uploads/media/MIT-AUTOID-WH-014.pdf`, 2002.

[31] A. Shukla, P. M. Deshpande, and J. F. Naughton. Materialized view selection for multidimensional datasets. In *Proc. 1998 Int. Conf. Very Large Data Bases (VLDB'98)*, pages 488–499, New York, NY, Aug. 1998.

[32] Tag data standard v. 1.4. Standard, EPCglobal, `http://www.epcglobalinc.org/standards/tds/tds_1_4-standard-20080611.pdf`, 2008.

[33] Class 1 generation 2 uhf air interface protocol standard "gen 2". Standard, EPCglobal, `http://www.epcglobalinc.org/standards/uhfc1g2/uhfc1g2_1_1_0-standard-20071017.pdf`, 2007.

[34] E. Wu, Y. Diao, and S. Rzvi. High-performance complex event processing over streams. In *Proc. 2006 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'06)*, pages 407 – 418, Chicago, Illinois, June 2006.

[35] D. Xin, J. Han, X. Li, and B. W. Wah. Star-cubing: Computing iceberg cubes by top-down and bottom-up integration. In *Proc. 2003 Int. Conf. Very Large Data Bases (VLDB'03)*, pages 476–487, Berlin, Germany, Sept. 2003.

[36] Y. Zhao, P. M. Deshpande, and J. F. Naughton. An array-based algorithm for simultaneous multidimensional aggregates. In *Proc. 1997 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'97)*, pages 159–170, Tucson, AZ, May 1997.