Modeling Object Flows from Distributed and Federated RFID Data Streams for Efficient Tracking and Tracing (Supplemental)

Yanbo Wu, Quan Z. Sheng, *Member, IEEE,* Hong Shen, and Sherali Zeadally

1 Why Centralized Models Do Not Work

If data from different nodes are stored in a centralized database using the schema above, the definitions themselves can answer the queries. Proper indices can improve the performance. However, there are still some significant performance issues in large scale systems such as IoT where the number of nodes and objects could be very large. We discuss below why a centralized solution does not scale by considering the characteristics of RFID data management.

- *Frequent Updates.* Data in IoT are generated as streams and update becomes a frequent operation comparing to traditional database management systems that are optimized for frequent-read scenarios. With a centralized database (or database clusters), frequent updates to the database not only increase the storage cost, but more importantly, cause high costs in index maintenance. For example, for streaming data, a B^+ tree index could be potentially huge. The space cost for the index itself is $O(Num_{objects} * Length_{path})$ where $Num_{objects}$ is the total number of objects in the whole network, $Length_{path}$ is the average number of nodes of paths that the object moves along.
- *Row Level Security Requirement.* Business applications often require high security. Even in a federated system, the partners want to control their own data. For example, a supermarket wants to hide the buying information about the same product from one supplier to another, while the suppliers can access the data related to their own products. This requires row level authentication and the authentication overhead for space is high.
- *Archiving.* RFID data, similar to other time series data, is sensitive to time. Recent records are more interesting to the analyzers than the distant ones. For the purpose of storage efficiency, it is often necessary to archive the old data in order to make room for the new data. However, different participants may have different definitions of oldness (i.e., when the data should be archived). As a result, when old records are archived, a range query on End with a selection on $Node$ has to be performed. Furthermore, because the records for different nodes are likely to be stored in different pages, after deletion, the pages will have a lot of fragments. The rearrangement of records in these pages could be very costly and time consuming.
- *Mining Efficiency.* Stream mining techniques such as online aggregation, critical layers or popular path materialization all require a significant amount of memory in a central server.

- *Y. Wu is with Beijing Jiaotong University. E-mail: ybwu@bjtu.edu.cn*
- *Q.Z. Sheng and H. Shen are with the University of Adelaide.*
- *S. Zeadally is with the University of the District of Columbia.*

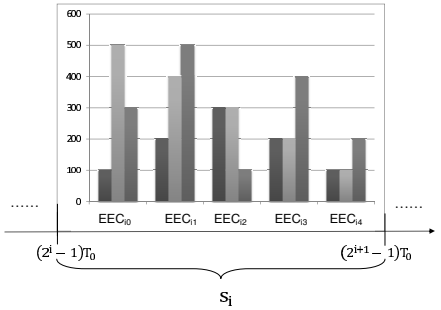


Fig. 1: Example of LTTF Slot

2 An Example of EEC Slot

Figure 1 shows an example structure of the slot s_i in LTTF. Slot s_i represents the data of the time interval $[(2^i-1)*\mathcal{T}_0, (2^{i+1}-1)*\mathcal{T}_0)$. It is split into w_s EECs. In this example, w_s is 5. For each EEC, we maintain a histogram which models the distribution of data from/to a neighbor. The current size of a slot is defined as the number of used EECs in it, so $s_i.size \leq w_s$.

The reason why we split the slots further into EECs and maintain histograms for each of them, instead of maintaining one histogram for each slot, is because we want a finer control over the memory usage. By splitting the slots, we can limit the memory usage for each slot by changing

Algorithm : Merge the Model: *merge*

Input: The slot which is being merged s_i
The model $\mathcal{M}$
Output: The merged model

1: **if** s_{i+1} does not exist in $\mathcal{M}$
2: $s_{i+1} \leftarrow$ new slot, $\mathcal{M}$.append(s_{i+1})
3: **end if**
4: **if** s_{i+1} is full, merge(s_{i+1}, $\mathcal{M}$)
5: **for** each neighbor b_j in s_i
6: $h_{i+1,j} \leftarrow s_{i+1}[b_j]$
7: **if** $h_{i+1,j}$ is nil
8: $s_{i+1}[b_j] \leftarrow h_{i+1,j} \leftarrow$ new array size of w_s
9: **end if**
10: move $h_{i+1,j}[1] \sim h_{i+1,j}[w_s/2]$
to $h_{i+1,j}[w_s/2+1] \sim h_{i+1,j}[w_s]$
11: **for** $k \leftarrow 1$ to $w_s/2$
12: $h_{i+1,j}[k] \leftarrow s_i[b_j][2*k] + s_i[b_j][2*k+1]$
13: **end for**
14: **end for**

Fig. 2: Algorithm to Merge the LTTF Model

the value of w_s. The larger w_s is, the more memory the model requires, and the more accurate the model is. In an extreme case, when w_s is 1, the slots are not split and the accuracy is the lowest.

The height of the bars in the histogram of each EEC is the volume of objects flow from/to a neighbor for the time represented by that EEC. The x-axis of the histogram represents the neighbors. The *length* of the model is defined as the number of slots within it. A model of length l can store the history of $w_s * w_e * \sum_{i=0}^{l-1} 2^i$. We can calculate the length of the model which stores the history for the past t time units using:

$$l = \log_2 (\frac{t}{w_s * w_e} + 1) = \log_2 (\frac{t}{\mathcal{T}_0} + 1) \qquad (1)$$

Suppose we define the width of an event cycle as an hour and for each slot, there are 24 EECs with in it, i.e., w_e is an hour and w_s is 24, we only need $\lceil \log_2 365 \rceil = 9$ slots to store the history of one year. This efficiency in space enables the storage of these slots in main memory.

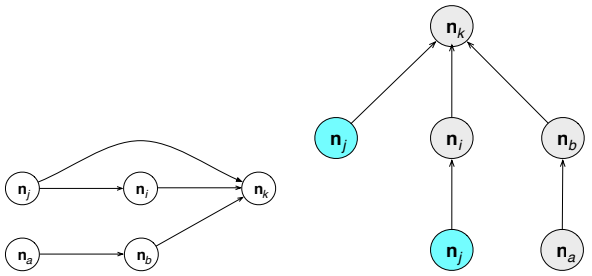
3 Merging with the Next Slot

When the new event cycle fills the most recent time slot s_0, the second most recent slot s_1 will be merged to the succeeding slots recursively.

The *merge* algorithm in Figure 2 first checks whether the next slot in the model is full. If so, it will be merged (line 4). This process is done recursively until either a non-full slot is found or all the existing slots are full. In the latter case, a new slot is created and appended to the model. Then the slot being merged will be integrated into the first non-full slot (line 5 – 14). It is compressed by merging two consecutive EECs into one (line 12). Note that in this algorithm, we assume that the width of slots w_s is even. This assumption does not affect the performance of the algorithm.

4 The Business Neighbor Tree

There are two critical issues in real-life applications. Firstly, a node may go offline with or without notification to other nodes in the network (but objects are still moving from/to that node). In this case, the information of connections relevant to that node becomes unavailable. Secondly, a node n_i and one of its source nodes n_j may both be the direct source node of n_k. We call this situation the *Sideway Problem*. Due to this, the algorithms in Section 5 of main file may fail to get the actual source node. As shown in Figure 3a, suppose object o moved from n_j to n_i, then from n_i to n_k. Both n_i and n_j are in the n_k's business neighbor set, and for the past few event cycles, n_j has more objects moving to n_k. Since n_j is the first to be queried, it will be treated as the source node of object o from which it moves to n_k. Clearly, this is not correct because the source node should be n_i.



(a) Sideway Problem (b) Source Tree

Fig. 3: Sideway Problem and the Source Node Tree

A source tree at a node n_k is a subgraph of the whole network topology, with n_k as the root, representing the topology of nodes that have either direct or indirect connections to n_k. Figure 3b depicts the source tree maintained at n_k in the sub-network shown in Figure 3a. The root of the tree is n_k. Its child nodes are the *direct* source nodes of n_k (i.e., n_i, n_j and n_b in Figure 3b).

The maintenance of the source tree is done simultaneously with the flow synopsis building. In the algorithm to gather the source node information, neighbors also include its source tree in the result set R. This source tree is then merged with the local source tree. In this way, the source tree is built recursively.

This structure adds more connectivity to the network, which increases the stability of the system in the case that some nodes may leave. However, it does not require replication of any form. It also helps to solve the *Sideway Problem*.

Essentially, the source tree gathers the nodes that are often on the same paths in the object flows. It does not store item-level information and statistics from other nodes. It is impossible for a node to get these information without permission from neighbors. As a result, although we stored a subgraph locally, we do not expose confidential information.

Solving Node Leaving Problem. With the source node tree, when a neighbor n_i leaves the network, we can query its direct source nodes $\mathbf{S}(n_i)$ to check whether an object comes from n_i, because if an object comes from a node's

direct source node n_i, it must also come from one of n_i's direct source nodes. In the source node tree, the LTTFs for all the nodes in $\mathbf{S}(n_i)$ are maintained, together with the LTTF for n_i itself. If an object comes from a node n_a in $\mathbf{S}(n_i)$, it is used in the maintenance of the LTTFs for both n_i and n_a. In this way, even when n_i is offline, its LTTF is kept accurate.

We do not remove n_i from the tree when it leaves. Instead, we mark it as "disconnected". After it is back online, we remove the "disconnected" mark. This is useful to deal with temporary departure of nodes (e.g., short server breakdown). The LTTFs for nodes in $\mathbf{S}(n_i)$ are deleted because we no longer need them to build the flow synopsis. In this way, even when some nodes in the network leave, we can still respond to tracing queries and continue to build flow synopsis.

If after a certain configurable time, n_i is still offline, it will be removed from the tree and its LTTF will be deleted or archived.

Solving Sideway Problem. To solve the *Sideway* problem, for the algorithms to answer tracing queries and building the flow synopsis, if a node has both direct and indirect connection to the root node, it will be queried regardless of the order of probabilities. For example, in Figure 3b, not only n_j is the direct source node of the root node n_k, but there is also an indirect connection ($n_j \rightarrow n_i \rightarrow n_k$) between n_j and n_k. n_j is queried regardless of its order in the sorted neighbor list.

After receiving the result set, we compare the arrival timestamps of the objects which are included in the result sets from more than one neighbors and select the node with the latest timestamp as the source node. For example, if the arrival timestamp for an object is earlier in n_j than that in n_i, it is clear that the object was sent through the path $n_j \rightarrow n_i \rightarrow n_k$.

5 Determining w_s and w_e

Generally, our model is a synopsis for the RFID stream. A generic description of such a data structure is that it supports two operations, $update$ and $computeAnswer$ [1]. In our model, the $update$ operation takes more time than $computeAnswer$, because it involves network queries (See Section 6). More importantly, it may be so slow that the histograms for the past event cycle have not yet been constructed when it starts constructing the histograms for the current event cycle. In this case, the most recent histograms in slot s_0 is actually an old one. If this happens, the most valuable data is not used because it is not ready yet. Figure 4 illustrates such a situation. At the end of the second event cycle (EC2), the construction of histograms for EC1 has not been done (which will be done at $w_e + t_u$, where t_u is the time used for update).

To avoid this problem, w_e should be large enough, i.e., w_e should be larger than the maximum possible value of t_u. But it should not be too large. If it is too large, the pattern of object flow may change during an event cycle. In this case, the change is not captured. t_u is determined by the size of the sample, the number of the business neighbors and the size of the whole network as discussed in Section 6. It is a dynamic value. A good way to determine w_e would be to make it a function of the three parameters. However maintaining the size of the whole network is not an easy task, and is costly. In this work, we propose an adaptive approach.

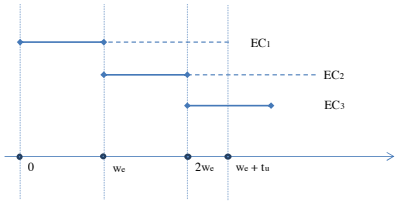


Fig. 4: When the Update Time Is Too Long

When a node joins the network, it sets w_e to a default value. During the maintenance process of the model, if t_u[1] becomes larger than w_e or smaller than $\frac{w_e}{2d}$, $w_e \leftarrow d * t_u (d > 1)$ (d is a constant between 1 and 2). In real applications, after the network becomes stable, t_u will likely become stable too, so w_e is almost a constant. Our algorithms, which are based on the assumption that w_e is a constant, are not affected. To deal with the unstable phase of model construction, each event cycle in the model is parameterized with a timestamp t_{end} for the end of the cycle.

The maximum number of EECs in a slot (i.e., w_s) has the influence on the memory usage of the LTTF model. Generally, the memory usage of the model is $O(l * w_s * n)$ (l is the length of model and n is the number of neighbors). w_s can be large, as long as the memory is sufficient. Larger w_s ensures the model with a finer granularity.

6 Performance Analysis

6.1 Model Maintenance Cost

The time complexity for maintaining the TISH model consists of three parts: i) sampling the input, ii) querying the source nodes, and iii) updating the model. Suppose t_u is the time used to update the model for an event cycle, we have:

$$t_u = t_{sample} + t_{query} + t_{update} \tag{2}$$

Using the reservoir algorithm, we only need to scan the input once, and this can be done on-the-fly. So the cost for this part is $O(x)$, where x is the total number of objects in the current event cycle.

For the worst case, the model update algorithms (Section 4 of the main file) will go through the whole model, with a complexity of $O(l)$, where l is the number of slots in the model. It should be noted that we can avoid the movement of histograms shown in Figure 2 (line 10). l is often small.

The most costly part is querying the source nodes, as the network latency is typically much longer than the

1. t_u is an observable variable.

$\mathcal{B}$

N_1	N_2	N_3	N_4	N_5
50	30	20	0	0

$\mathcal{B}'_1(\delta = 0)$

N_1	N_3	N_2	N_4	N_5
50	30	20	0	0

$\mathcal{B}'_2(\delta = 2)$

N_1	N_2	N_4	N_5	N_3
50	30	10	10	0

$\mathcal{B}'_3(\delta = 1)$

N_1	N_2	N_4	N_3	N_5
50	30	10	10	0

Fig. 5: Example in Modeling Accuracy

local processing time. Obviously, when querying the source nodes, the queries can be sent out simultaneously. However, the network traffic cost will be $O(n)$ (n is the number of neighbors). This is not economical because building the flow synopsis is not necessary to be done in real time. Indeed, the TISH model is only updated when an event cycle ends, and the maintenance time t_u is controlled to be less than an event cycle as discussed in Section 5.

With the algorithm shown in Section 5 of the main file, we can save the network traffic cost by querying the nodes which have most possibility to be the source node first. Moreover, we can also avoid the underlying P2P queries when there are no new business neighbors joining. Even if there is one, after the first event cycle, the new neighbor will be in the LTTF model with high probability. As a result, the network cost is reduced.

The accuracy of the model is very important for reducing network traffic cost to maintain the model. We represent the real distribution of objects' source nodes during a specific time period as $\mathcal{B}$, which is the neighbor list sorted in descending order by the real distributions (e.g., $\mathcal{B}$ in Figure 5). For example, in Figure 5, there are five possible source nodes (N_1 to N_5) with the possibilities of 50% for N_1, 30% for N_2, and 20% for N_3. The distributions modeled by the TISH model in the same period of time is denoted as $\mathcal{B}'$. Figure 5 depicted different scenarios for the accuracy of the TISH model.

Suppose that, the m objects (the value of m does not matter in this discussion), which we want to query their source nodes, come from y different nodes. In this case, $y \leq m$. In Figure 5, y is 3, because in the real distribution $\mathcal{B}$, the m objects can only come from N_1, N_2 and N_3.

In the best case scenario, y queries are needed. Because we have to query the first y nodes in order to find source nodes for all objects. $\mathcal{B}'_1$ is one of the best scenarios. Although the order of N_2 and N_3 are not the same as $\mathcal{B}$, 3 queries are still enough to find the source nodes. In general, the best scenario happens as long as the first y nodes in the model $\mathcal{B}'$ are the same as the first y nodes in the real distribution $\mathcal{B}$, regardless of the order.

In the worst case scenario, all the nodes in the neighbor list are queried. $\mathcal{B}'_2$ is one of these cases. The neighbor N_3 which indeed is a source node, is ranked the last in $\mathcal{B}'_2$, so we have to query all the 5 neighbors to find the source nodes for all objects. In general, the worst case happens when there exists one node in the first y nodes of $\mathcal{B}$ being ranked the last in $\mathcal{B}'$.

Suppose δ is the number of *extra* queries made in addition to the optimized value y, we can conclude that:

$$\delta = max_{i=1}^{y}(\mathcal{B}'.rank(\mathcal{B}[i])) - y \tag{3}$$

For example, in Figure 5, δ is 0 for $\mathcal{B}'_1$ because all first 3 nodes in $\mathcal{B}$ are still ranked first 3 in $\mathcal{B}'_1$, thus $max_{i=1}^{3}(\mathcal{B}'_1.rank(\mathcal{B}[i]))$ is 3. For $\mathcal{B}'_2$, N_3 in first 3 of $\mathcal{B}$ is ranked 5 in $\mathcal{B}'_2$, thus $max_{i=1}^{3}(\mathcal{B}'_2.rank(\mathcal{B}[i]))$ is 5, so δ is 2 (i.e., 5 - 3). Similarly for $\mathcal{B}'_3$, N_3 is ranked the 4^{th} in $\mathcal{B}'_3$, so δ is 1.

The goal of our model is to keep the average value of δ as small as possible. We will demonstrate through extensive simulation experiments (described in Section 7) that our proposed TISH model is very accurate in this sense.

6.2 Tracing and Tracking Cost

The cost of tracking and tracing consists of two parts. If the initiating node of the query is not on the movement path of the object, the query should first be redirected to a certain node on the path. The cost of this step is the cost of the underlying P2P overlay lookup cost. However, in real applications, it is rare that the query is initiated by a third party. So in most cases, this part of the cost does not exist.

The second part is the query cost. The total cost of a trace query (t_{trace}) is a function of the length of the object's moving path. In general, $t_{trace} = \sum_{i=1}^{l} t_i$, where t_i is the time used to establish the i^{th} segment in the path and l is the length of the path. We assume that t_i is proportional to the number of queries (q) made, so $t_i = q * \mathbf{T}$ where $\mathbf{T}$ is a constant representing normal remote query processing time. Clearly, the cost is determined by q. It is easy to infer that if the object comes from the j^{th} neighbor in $\mathcal{B}$, then j queries are needed to find it. Thus, the optimized average value of q is (in this section, we reuse the definition of notations in Section 6.1.):

$$q = \sum_{j=1}^{y}(j * \mathcal{B}[j].probability) \tag{4}$$

i.e., the order of the querying follows the *actual* order of the neighbors sorted by the probability that the object comes from them.

The number of queries q' by using the TISH model is:

$$q' = \sum_{j=1}^{y}(\mathcal{B}'.rank(\mathcal{B}[j]) * \mathcal{B}[j].probability) \tag{5}$$

Figure 6 shows an example of the optimized cost and the cost with the TISH model by using the examples in Figure 5. If the actual order ($\mathcal{B}$) is followed when querying the source node, we only need, on average, $0.5*1+0.3*2+0.2*3 = 1.7$ queries. But in a different order, for example, in $\mathcal{B}'_1$, the order of N_2 and N_3 is reverted. In this case, we have to use $0.5*\mathcal{B}'_1.rank(N_1)+0.3*\mathcal{B}'_1.rank(N_2)+0.2*\mathcal{B}'_1.rank(N_3) = 0.5*1+0.3*3+0.2*2 = 1.8$ queries.

The goal of tracing query processing is to make q' as close to q as possible. We will show in Section 7 that the

$\mathcal{B}[j]$	probability	Number of Queries			
		$\mathcal{B}$	$\mathcal{B}'_1$	$\mathcal{B}'_2$	$\mathcal{B}'_3$
N_1	0.5	1	1	1	1
N_2	0.3	2	3	2	2
N_3	0.2	3	2	5	4
q or q'		1.7	1.8	2.1	1.9

Fig. 6: Examples of Tracing Efficiency

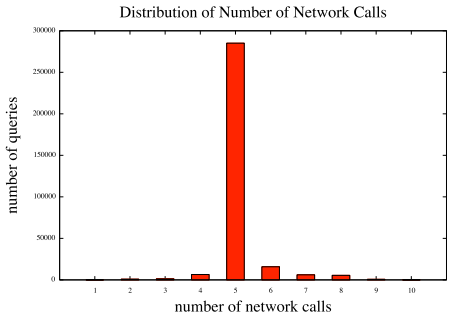


Fig. 7: Distribution of Number of Network Calls for Model Maintenance

TISH model can make the difference less than 0.1, i.e., in most cases, $\mathcal{B}'$ contains the same neighbors with the same order as $\mathcal{B}$.

7 Supplemental Experiments

7.1 Model Maintenance Cost

An interesting performance evaluation is to investigate how many network calls are used per query for the tracking and tracing algorithm, and the type of distribution we obtain. We illustrate the result in Figure 7, by using a histogram. We note that the majority of the queries use less than 10 (the average number of fan-out) network calls in order to maintain the TISH model. According to the analysis in Section 6.1, we expect that the average number of network calls for model maintenance is close to the *effective fan-outs*, which is the average number of active (non-idle) connections for all the nodes (5 in our experiments). The average number of network calls in this experiment is 5.11, which is close to what we expect.

7.2 Effects of Constant c

We are interested in how the constant c affects the performance of the model. In this experiment, we examined the performance of the model maintenance with various values of c from 1 to 5. It is clear that when c is big, the accuracy becomes worse. This is because by using larger c, more old data are included and they biases the model. So we limit the test to a reasonable range. We ran the tests with both "Constant Patterns"-only and "Mixed Without Random" settings as defined in Section 6 of the main file. Figure 8 shows the result for this experiment.

With "Mixed Without Random" setting, the performance is the best when c is 3. This is because that when c is

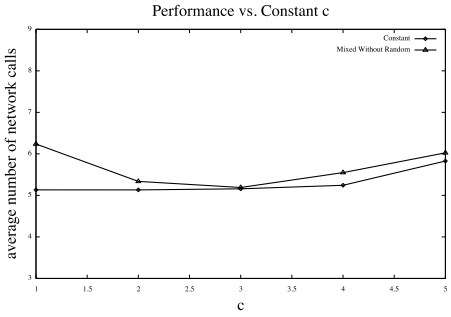


Fig. 8: Number of Network Calls against c

smaller than 3, only two of the most recent event cycles are used. The model becomes too sensitive to the changes of the patterns. The little bias in the sampling will be expanded in the model. When c is greater than 3, more historical data are used to calculate the probabilities. This makes the model not sensitive to the changes of the patterns.

However, with "Constant Pattern", when c is small (less than 4 in our result), it does not affect the performance. This is because the network topology does not change. The only dynamic change is the *idle/active* state switch of connections. If c is too large, the TISH model becomes insensitive to these dynamic changes and the performance becomes worse.

8 Additional Literature Review

8.1 Data Structures and Data Transformation

A raw RFID record is a triple tuple $(ID, time, location)$ [2], which presents little value until it is transformed into a form suitable for application-level interactions. In addition, such data has implicit meanings and associated relationships with other RFID records where applications have to make appropriate inferences [15].

In one of the earliest efforts on RFID data modeling, Wang et al. propose an RFID data model that abstracts static and dynamic entities including object, reader, location and transaction [16], [17]. It models the interaction between them as either *state* or *event* based relationships. The data model also provides a rule-based data filter engine. In [12], RFID applications are classified into a set of typical scenarios and a generalized data modeling framework with constructs for each typical scenario is proposed. In [6], Hu et al. focus on the path encoding by using a Bitmap data type and they demonstrated that significant storage savings can be achieved. In [7], a novel data structure and algorithms to efficiently encode, decode and query an object's moving path in an RFID database are proposed. The approach has been further improved very recently in [8] to cope with the situation where the moving path is long. Finally, Lin et al. propose in [11] a "Multi-Table" model in which *path* and *containment* relationships can be defined.

8.2 Knowledge Discovery

For an RFID data warehousing approach proposed by Gonzalez et al. in [4], the authors observe that individual objects tend to move and stay together (i.e., bulky object movements in supply chain). They propose a novel model to compress RFID data without information loss. The model consists of a hierarchy of highly compact summaries of data aggregated at different abstraction levels where analysis takes place. These summaries are represented as RFID-cuboids. Each RFID-cuboid records object movements and stores product information for each RFID object, information on objects that stay together at a location, and path information necessary to link multiple stay records. This work is further improved in [3] by discovering the *Gateway* nodes that have either high fan-in or high fan-out edges. The RFID cuboids are created based on this discovery to save more space. This method is restricted to process static data in centralized environments. It is not suitable for processing RFID data streams.

In [9], Lee and Park propose a dynamic tracing model for supply chain management, which considers not only the movements, but also the combination and splitting of RFID-attached objects. In [13], an algorithm to extract frequent sequential patterns from RFID data streams is proposed. The authors also introduce a decision tree model to determine the prime movements of tagged objects by using the extracted patterns. TMS-RFID [10] is a system to manage temporal RFID data, which extracts complex temporal event patterns over RFID streams. In [5], the authors propose a probability evaluation model and algorithms for moving range query processing. These models focus on different aspects of modeling RFID data, but they all have the same limitations as RFID-cuboid.

SPIRE [14] is a novel RFID data interpretation and compression system. It extracts location and containment relationships over RFID streams by applying a probabilistic algorithm over a time-varying graph model. This model can be deployed in either centralized or distributed environments. However, it still requires full access to all the data.

The BRIDGE project[2] is an implementation of the EPCglobal framework. It includes a probabilistic model to predict future location of RFID-attached objects using Markov Chain. However, this model is designed specifically for supply chain management systems. It requires synchronization with static supply chain model, which makes it inflexible.

REFERENCES

[1] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and Issues in Data Stream Systems. In *Proceedings of the 21th ACM Symposium on Principles of Database Systems (PODS'02)*, Madison, Wisconsin, 2002.

[2] S. S. Chawathe, V. Krishnamurthy, S. Ramachandran, and S. Sarma. Managing RFID Data. In *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB'04)*, Toronto, Canada, September 2004.

2. http://bridge-project.eu

[3] H. Gonzalez, J. Han, H. Cheng, X. Li, D. Klabjan, and T. Wu. Modeling Massive RFID Data Sets: A Gateway-Based Movement Graph Approach. *TKDE*, 22:90–104, 2010.

[4] H. Gonzalez, J. Han, X. Li, and D. Klabjan. Warehousing and Analyzing Massive RFID Data Sets. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE'06)*, Atlanta, Georgia, USA, April 2006.

[5] Y. Gu, G. Yu, N. Guo, and Y. Chen. Probabilistic Moving Range Query over RFID Spatio-temporal Data Streams. In *Proceeding of the 18th ACM Conference on Information and Knowledge Management (CIKM'09)*, Hong Kong, China, 2009.

[6] Y. Hu, S. Sundara, T. Chorma, and J. Srinivasan. Supporting RFID-Based Item Tracking Applications in Oracle DBMS Using a Bitmap Datatype. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB'05)*, Trondheim, Norway, September 2005.

[7] C.-H. Lee and C.-W. Chung. Efficient Storage Scheme and Query Processing for Supply Chain Management Using RFID. In *Proceedings of the 2008 ACM International Conference on Management of Data (SIGMOD'08)*, Vancouver, Canada, 2008.

[8] C.-H. Lee and C.-W. Chung. RFID Data Processing in Supply Chain Management Using a Path Encoding Scheme. *IEEE Transactions on Knowledge and Data Engineering*, 23(5):742–758, 2011.

[9] D. Lee and J. Park. RFID-based Traceability in the Supply Chain. *Industrial Management and Data Systems*, 108(6):713–725, 2008.

[10] X. Li, J. Liu, Q. Z. Sheng, S. Zeadally, and W. Zhong. TMS-RFID: Temporal Management of Large-scale RFID Applications. *Information Systems Frontiers*, 13(4):481–500, 2011.

[11] D. Lin, H. G. Elmongui, E. Bertino, and B. C. Ooi. Data Management in RFID Applications. In *Proceedings of the 18th International Conference on Database and Expert Systems Applications (DEXA'07)*, Regensburg, Germany, 2007.

[12] S. Liu, F. Wang, and P. Liu. Integrated RFID Data Modeling: An Approach for Querying Physical Objects in Pervasive Computing. In *Proceedings of the 15th ACM International Conference on Information and Knowledge Management*, Arlington, Virginia, USA, 2006.

[13] T. Nakahara, T. Uno, and K. Yada. Extracting Promising Sequential Patterns from RFID Data Using the LCM Sequence. In *Proceedings of the 14th International Conference on Knowledge-Based and Intelligent Information and Engineering Systems (KES'10)*, Cardiff, UK, 2010.

[14] Y. Nie, R. Cocci, Z. Cao, Y. Diao, and P. Shenoy. SPIRE: Efficient Data Interpretation and Compression over RFID Streams. *IEEE Transactions on Knowledge and Data Engineering*, 24(1):141–155, 2012.

[15] Q. Z. Sheng, X. Li, and S. Zeadally. Enabling Next-Generation RFID Applications: Solutions and Challenges. *IEEE Computer*, 41(9):21–28, September 2008.

[16] F. Wang and P. Liu. Temporal Management of RFID Data. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB'05)*, Trondheim, Norway, September 2005.

[17] F. Wang, S. Liu, and P. Liu. A Temporal RFID Data Model for Querying Physical Objects. *Pervasive and Mobile Computing*, 6(3):382–397, 2010.