Efficient Distributed Query Processing in Large RFID-enabled Supply Chains

Jia Liu[†‡], Bin Xiao[‡], Kai Bu[‡] and Lijun Chen[†]
[†]State Key Laboratory for Novel Software Technology, Nanjing University, China
[‡]Department of Computing, The Hong Kong Polytechnic University, China
Email: liujia@smail.nju.edu.cn, {csbxiao,cskbu}@comp.polyu.edu.hk, chenlj@nju.edu.cn

***Abstract*—Radio Frequency Identification (RFID) has dramatically streamlined supply chain management by automatically monitoring and tracking commodities. Considering the proliferation of RFID data volume, distributed storage is more applicable and scalable than centralized storage for distributed query processing. Traditional distributed RFID data storage requires each distribution center to locally store raw RFID data, leading to data redundancy, storage and query inefficiency. In this paper, we design an efficient distributed storage model by leveraging Bloom filters to save storage space and improve query efficiency. Meanwhile, we establish corresponding query processing schemes to locally support existence queries and path queries, which are two kinds of most popular queries in the supply chain management. A local query can be completed with constant time complexity regardless of data volume. Experiments demonstrate that our storage model outperforms the traditional one in terms of both space and time efficiency.**

I. Introduction

Radio Frequency Identification (RFID) plays a crucial role in improving the efficiency of business processes in supply chain management by automatically monitoring and tracking commodities [1]–[7]. In the near future, the RFID data volume tends to be enormous. For example, Venture Development Corporation, a research firm, predicts that when tags are used at the item level, Walmart will generate around 7 terabytes of data every day [8]. Along with the data explosion, many new challenges arise. One big challenge is to store big data and improve query efficiency in large RFID-enabled supply chains, resulting in the urgent need of new approach design for efficient data storage.

Applying *centralized storage* of RFID data in a supply chain can maintain a single copy of all RFID raw data. However, it is not scalable for large-scale RFID-enabled supply chains. Although this approach can simplify query processing by a global view of the entire supply chain, it causes high communication cost due to transmitting all raw data to a central server for archival and query processing. In addition, the central server is prone to be the bottleneck for high-frequency query processing.

An alternative to centralized storage is the *distributed storage* with RFID data spreading across warehouses/distribution centers in a supply chain. In the traditional distributed storage, each warehouse locally stores the raw data of commodities that move through it. This approach dramatically reduces the communication overhead as the raw RFID data no longer need to be transmitted to a central server. Thus, it is more applicable to large-scale warehouse management, considering the proliferation of RFID data and limited bandwidth resources. However, two tough challenges have to be addressed. The first challenge is how to compress massive and redundant data. Throughout the entire supply chain, products are transferred from one position to another in a flow. A simple local storage of each commodity leads to data redundancy in the flow route. The second challenge is how to support efficient query processing without a global view of RFID data. Unfortunately, most previous work focused on centralized storage [9]–[13]. Few research papers have investigated the two challenges in the distributed storage, which are, obviously, highly desirable.

In this paper, we take the first step toward efficient distributed storage to support fast query for large RFID-enabled supply chains. The major contributions of this paper are threefold.

First, we propose a novel space-efficient and time-efficient distributed RFID data storage model. The model captures the product flow property in a supply chain and incorporates the Bloom filter technique. It can provide constant storage time regardless of the number of tags to be stored, and significantly compress massive raw data. For example, a Bloom filter can store an EPC with only 9.6 bits, which reduces data by a factor of 10 compared with the traditional storage (96 bits per EPC in the C1G2 standard). The great space saving can allow the stored data to be put in the main memory of a query processing server, which in turn massively reduces the query time without resorting to slow hard disk scanning.

Second, we establish corresponding query processing schemes to locally support the existence query and path query. A local query can be completed with a constant time complexity regardless of data volume. We even can totally remove false positives caused by Bloom filters to obtain accurate queries by sending these queries to terminal nodes.

Third, we comprehensively analyze key performance indicators, such as false positive probability, storage space, storage time, query time and overhead, to optimize the model setting. Extensive simulations and experiments show that our storage model can save local storage space 10 times with 1% *local* false positive probability and the query processing can improve local query efficiency 70 times, compared with the traditional distributed RFID data storage approach.

The rest of the paper is organized as follows. Section II presents the system model. Section III proposes our RFID data storage model. Section IV establishes the corresponding query processing schemes. Section V analyzes the false positive probability, followed by the performance analysis in Section VI. Section VII evaluates our model through both

978-14799-3360-0/14/$31.00 ©2014 IEEE

simulations and experiments. Section VIII reviews the related work. Finally, Section IX concludes this paper.

II. System Model

A. System Assumption

In the supply chain, commodities flow from suppliers to distributors. Each business entity has several downstream distributors to which it sells its products. Meanwhile, it is the downstream distributor of its upstream supplier from which it purchases products. In this paper, we focus on the distribution process for one kind of commodity in the supply chain, which forms a well-accepted tree structure shown in Fig. 1.

The node A, as the source of the entire supply chain, is B, C and D's upstream supplier (parent node). By contrast, B, C and D are A's downstream distributors (children). We treat a business entity as a node of the tree in the rest of this paper. Assume that every node except the root has only one parent but is likely to have multiple children. Meanwhile, the node can communicate with all its children as well as the only parent node. Because products are completely distributed to the current node's children until leaf nodes, all commodities flow from the root to leaf nodes. Throughout the supply chain, we divide the business entities into two categories: terminal nodes (leaf nodes) and internal nodes (non-leaf nodes).

B. Problem Definition

Under above supply chain structure, our problem definition is as follows: Design an efficient distributed storage model to store a set of n tags S=$\{a_1, a_2, ..., a_n\}$ when they are distributed in a tree-structure flow, and establish corresponding query processing schemes, satisfying the following requirements:

1) Store data locally to reduce enormous communication overhead and a query can be launched at any node. 2) Compress massive raw data and improve storage efficiency to be scalable in a large-scale supply chain. 3) Complete the local query with a constant time complexity, and support the remote query involving multiple nodes to provide any query accuracy and product trajectory information.

In summary, we aim to design a space-efficient distributed storage model to support time-efficient query processing in large-scale RFID-enabled supply chains.

C. Bloom Filters

The Bloom filter [14] is an efficient data structure that can not only rapidly check set membership but also greatly save storage space. An empty Bloom filter is an array with m bits initially set to 0. There are also p hash functions H_i $(1 \leq i \leq p)$, each of which maps the input into $\{1, 2, 3, ..., m\}$. Let $C[k]$ be the k^{th} bit of the array. To store a set $S = \{a_1, a_2, a_3, ..., a_n\}$ in the Bloom filter, all $C[H_i(a_j)]$ are set to 1 for each a_j $(1 \leq j \leq n)$. Consider an item a. If all $C[H_i(a)]$ are 1, a is regarded as a member of S with an error rate. Otherwise, a is definitely not in the set S.

Given a Bloom filter with m bits and p hash functions, both insertion and query time are $O(p)$. Specifically, to add an element or check membership, it only needs to run the element through the p hash functions and update or check corresponding bits, regardless of the data volume. In addition, Bloom filters can significantly compress data at the expense of a small error rate. For example, a Bloom filter with 1% error rate and an optimal value of p, requires only about 9.6 bits per item. Therefore, the Bloom filter is not only time-efficient but also space-efficient, which is the main reason why we incorporate it into our data storage model.

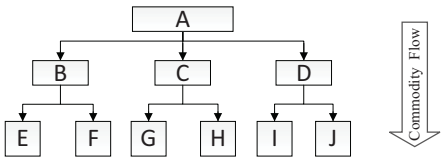


Fig. 1: The supply chain structure

III. RFID Data Storage Model

In this section, we design an efficient distributed storage model leveraging Bloom filters to save storage space and improve storage efficiency. In traditional distributed storage, when tagged commodities are transferred from one location to another, the tags are scanned automatically by RFID readers. Then, the tag information, such as EPCs, is stored locally to avoid high communication overhead. However, this kind of local storage causes data redundancy from a global view of the supply chain. In addition, the huge local data volume in upstream suppliers will cause high storage cost. For instance, the root is supposed to store all tags' information. Therefore, our primary performance objective is to reduce storage space and improve storage efficiency.

As aforementioned, there are two kinds of business entities: terminal nodes and internal nodes. In our storage model, every node maintains a space-efficient Bloom filter to store EPCs of all tags that move through it. The storage operation in the Bloom filter is an irreversible hash process that cannot restore raw information, so one copy of raw data are reserved in terminal nodes for minimizing redundancy as well as avoiding information loss. In other words, every terminal node not only holds a Bloom filter for storing EPCs, but also saves raw tag information in a database table like epc_table (epc, $product\ name$, ...). As a result, internal nodes can efficiently compress RFID EPCs while terminal nodes can offer accurate queries. We will analyze the space and storage efficiency of the model in Section VI.

We present an example to illustrate our storage model in Fig. 2. The internal nodes A, B and C use only Bloom filters to store tags' EPCs. With the whole tag set S=$\{a_1, a_2, a_3, ..., a_8\}$ distributed from A, A inserts S into its Bloom filter. Similarly, after purchasing related commodities, B and C insert $\{a_1, a_3, a_5, a_7\}$ and $\{a_2, a_4, a_6, a_8\}$ into their Bloom filters respectively. Apart from Bloom filters, the terminal nodes D, E, F and G also save tag information with epc_tables. For example, when the tag set S_{11}=$\{a_1, a_3\}$ is distributed to D, D stores raw information of $\{a_1, a_3\}$ in its epc_table and also inserts their EPCs into its Bloom filter. So do E, F and G.

IV. Query Processing

According to our proposed distributed storage model, this section designs the corresponding query processing to support

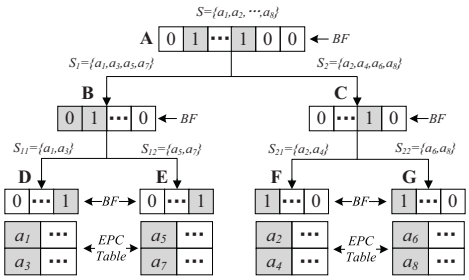


Fig. 2: An example of distributed storage model

two kinds of most popular queries in the supply chain management, existence queries and path queries.

A. Existence Query

Existence queries check whether a tag has been in a certain location (the granularity is the node). We define ***local existence query*** (local query for short) as a kind of existence query asking for information from only a node's local storage without communicating with other nodes. Specifically, when a local query launched at a node L, L executes the query based on its local Bloom filter. After that, the query is likely to return $false$ with certainty or $true$ with an error rate caused by the inherent false positives of Bloom filters (query accuracy will be discussed in Section V). If the query result is $false$, the query will finally be terminated with $false$ result. Otherwise (the query returns $true$), L will continue to inquire its epc_table for achieving higher query accuracy only when L is a terminal node, without further querying other nodes.

When an existence query at the node L requires high query accuracy like 99.9% but the local query with 99% accuracy cannot meet this requirement, it is necessary for L to cooperate with other nodes for accomplishing this confined existence query. In this case, L first executes the local query based on its local storage. If the query returns $false$ or L is a terminal node, L will stop the query immediately since the query accuracy reaches up to 100%. Otherwise (the result is $true$ and L is an internal node), L will inform all its children to execute the query and then wait for their replies. Among these replies, the final result is $true$ as long as there is a $true$ reply. Otherwise, the query returns $false$ as the final result. Upon receiving the query, all L's children do the same operations as L does. This process repeats until it satisfies the suitable terminating condition, such as the expected query accuracy.

In light of this, we introduce an important definition: ***N-layer existence query***. For an existence query launched at the node L, if it is allowed, at most, to be forwarded to the N^{th} layer node in the subtree rooted from L, then we refer to this kind of existence query as N-layer existence query (N-layer query for short) issued at L. Note that the number of layers starts counting from 1. That is, L as the root of the subtree is the first layer and the local query is actually the 1-layer query. Algorithm 1 describes the specific query and communication process of the N-layer query. Take Fig. 3 for example. A 2-layer query is issued at B. If B's local query returns $true$ (errors may occur), B will send the updated query to its children and wait for their responses after changing the 2-layer query into the 1-layer query. Upon receiving the 1-layer query, B's children, D and E execute the query and reply to

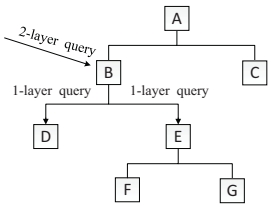


Fig. 3: A 2-layer existence query

B. As a result, B returns the final result according to their responses. On the other hand, if B's local query returns $false$ with certainty, the 2-layer query with the $false$ result will be terminated immediately without further querying. Therefore, the 2-layer query at the node B, at most, is forwarded to the second layer, D and E, in the subtree rooted from B.

Algorithm 1: N-layer Existence Query

Input: t: tag's EPC, n: layer limitation, L: current node
Output: $true/false$

```
re = local existence query(t, L);
if re == false || L is terminal node || n ≤ 1 then
    send re to the querier; return;
else
    send the query with parameters (t,n − 1) to L's children;
    RP = the set of replies from all L's children;
    count = |RP|;
    for (i = 1; i ≤ count; i++) do
        if RP[i] == true then
            send true to the querier; return;
        end if
    end for
    send false to the querier; return;
end if
```

B. Path Query

Besides existence queries, the path query as another kind of common query is widely available for tracing the tagged item's lifetime trajectory in practical applications. If a path query for a tag t is issued at the node L and t went though L before, then t's trajectory will be split into three parts: L, $upstream\ path$ composed of L's ancestor nodes and $downstream\ path$ composed of L's descendant nodes. Thus, the complete path is supposed to be $upstream\ path \rightarrow L \rightarrow downstream\ path$. Consider the path query. L first carries out the local query to check whether t has stayed in L. If the query result is $false$, the path query finally returns $null$ without further querying. Otherwise, the trajectory will consist of only $upstream\ path \rightarrow L$ when L is a terminal node, but $upstream\ path \rightarrow L \rightarrow downstream\ path$ when L is an internal node.

For the upstream path, L is supposed to inquire its parent node which executes the same operation as L does until the root node, as shown in Algorithm 2. Clearly, once a tag went through L, it must stay in L's parent node, due to the fact that all products in L are distributed from L's ancestor nodes. Therefore, the upstream path is related to only the supply chain structure, instead of the queried tag. Further, Algorithm 2 is required to be executed only once for a fixed supply chain structure. On the other hand, consider the downstream path query. L sends the query to all its children and waits for their replies. Then, any child node C receiving the query executes

the local query firstly. If it returns $false$, C will send $null$ to its parent node immediately without other operations. Otherwise, C also forwards the query to its children and waits for their replies as L does. After receiving a non-null $path$, C assembles and sends the new path $C{\rightarrow}path$ to its parent node. However, if all $path$s tend to be $null$, C will send $null$ as the final result to its parent node. The process repeats until terminal nodes, as shown in Algorithm 3.

V. False Positive Probability

Although Bloom filters incorporated in our storage model are space-efficient, they are likely to cause query errors. In order to clarify the query accuracy, we analyze the error rate of different queries in this section. In general, our storage model has no false negatives (FNs) but false positives (FPs). An FN means the query result is $false$ but it is in fact $true$. In our model, once a tag goes through a node, the node must store the tag's EPC in its Bloom filter, resulting in corresponding bits are all set to 1. When a new query about this tag is issued, it is definite that the EPC has been in the previous Bloom filter. Therefore, there are no FNs occurring in our storage model. Unfortunately, an FP is likely to occur when a Bloom filter suggests that a tag is in the Bloom filter but it is actually not. We next analyze the false positive probability (FPP) of two kinds of queries, existence queries and path queries.

Algorithm 2: Upstream Path Query

Input: L:node who launches the query, M:current node
1: **if** M is the root **then**
2: **if** L == M **then**
3: return $null$;
4: **end if**
5: send "M" to L; return;
6: **else**
7: send the query with parameter "M" to M's parent;
8: **end if**
9: $path$ = the reply from M's parent node;
10: **if** L==M **then**
11: return $path$;
12: **else**
13: new_path=$path{\rightarrow}M$;
14: send new_path to L; return;
15: **end if**

Algorithm 3: Downstream Path Query

Input: t: tag EPC, M: current node
1: re = local existence query(t);
2: **if** re == $false$ **then**
3: send $null$ to M's parent node; return;
4: **end if**
5: **if** M is the terminal node **then**
6: send "M" to M's parent node; return;
7: **end if**
8: send the query with parameters(M,t) to M's children;
9: $paths$ = the set of replies from all M's children;
10: count = $|paths|$
11: **for** $(i = 1; i \leq count; i++)$ **do**
12: **if** $paths[i] \neq null$ **then**
13: break;
14: **end if**
15: **end for**
16: **if** $i > count$ **then**
17: send $null$ to M's parent node;
18: **else**
19: new_path = $M{\rightarrow}paths[i]$;
20: send new_path to M's parent node; return;
21: **end if**

According to the process of the path query, the query may be terminated by $false$ result with certainty or be forwarded to terminal nodes with epc_tables guaranteeing 100% query accuracy. Thus, there are no FPs happening for path queries. As a result, we focus on analyzing the FPP of the existence query. Table I lists the notations to be used below. We first discuss FPPs of local queries and then FPPs of N-layer queries.

TABLE I: NOTATIONS OF PARAMETERS

Parameters	Definitions
n_i	the number of tags stored at the node i
m_i	the length of the bit array of i's Bloom filter
p_i	the number of hash functions of i's Bloom filter
f_i	the FPP of the local query at the node i
f_i^n	the FPP of the n-layer query at the node i
$H(i)$	height of the tree rooted from i (count from 1)
$D(i)$	the set of the node i's children

A. FPP of the Local Query

Consider a local query checking the membership based on only the local storage. Terminal nodes do not incur FPs due to their epc_tables offering 100% query accuracy. However, an internal node which only holds a Bloom filter is likely to cause FPs. Assume that the Bloom filter consists of m bits and p hash functions for storing n tags.

Theorem 1: The FPP of the local existence query at the internal node is:

$$f = \left(1 - (1 - \frac{1}{m})^{pn}\right)^p \approx \left(1 - e^{\frac{-pn}{m}}\right)^p \quad (1)$$

Proof: The probability that a certain bit is set to 1 by a hash function is $\frac{1}{m}$. Thus, the probability that the bit is not set by a hash function is $(1 - \frac{1}{m})$. After inserting n elements with p hash functions, the probability that the bit is still 0 is $(1 - \frac{1}{m})^{np} \approx e^{\frac{-pn}{m}}$. Hence, the probability that the bit is 1 is $(1 - (1 - \frac{1}{m})^{np}) \approx (1 - e^{\frac{-pn}{m}})$. The FP occurs when the p bits mapped by the p hash functions are all 1. Therefore, the FPP is $f = (1 - (1 - \frac{1}{m})^{pn})^p \approx (1 - e^{\frac{-pn}{m}})^p$. ■

The FPP decreases with m but increases with n. Table II presents different FPPs under different optimized parameters in Bloom filters. When $m/n = 10$, the FPP is 1%. In other words, if each EPC (96 bits in the C1G2 standard) takes up 10 bits in a Bloom filter, the query accuracy will reach up to 99%. It indicates that the Bloom filter can greatly compress raw data at the expense of a small error rate.

TABLE II: FPPS UNDER DIFFERENT PARAMETERS

FPP	10%	1%	0.1%	0.01%	0.001%
m/n	5	10	15	20	24
p	3	7	10	14	17

To sum up, the FPP of the local query at a node L is:

$$f_L = \begin{cases} 0 & if\ L\ is\ a\ terminal\ node \\ \left(1 - e^{\frac{-p_L n_L}{m_L}}\right)^{p_L} & otherwise \end{cases} \quad (2)$$

B. FPP of the N-Layer Query

Having considered various tree structures, we find that it is too complicated to directly deduce the FPP of the N-layer query. Instead, we resort to recursive analysis. The FP of the N-layer query at the node L occurs, only when both L's local query and any i's ($i \in D(L)$) (N-1)-layer query incur FPs.

Theorem 2: The FPP of the N-layer query at L is:

$$f_L^N = f_L\Big(1 - \prod_{i \in D(L)} (1 - f_i^{N-1})\Big) \tag{3}$$

where, $f_i^j = 0$ ($j \geq 1$) when i is a terminal node.

Proof: The N-layer query will be updated to the (N-1)-layer query when being sent to L's children. For a single node i ($i \in D(L)$), the probability that it does not incur an FP of the (N-1)-layer query is $(1 - f_i^{N-1})$. Thus, the probability that there are no FPs of (N-1)-layer queries at all $D(L)$ is $\prod_{i \in D(L)}(1 - f_i^{N-1})$. The probability that at least one node i incurs an FP of the (N-1)-layer query is $(1 - \prod_{i \in D(L)}(1 - f_i^{N-1}))$. Therefore, the FPP of the N-layer query is the probability that both the FP of L's local query and FP of any i's ($i \in D(L)$) (N-1)-layer query occur: $f_L^N = f_L\big(1 - \prod_{i \in D(L)}(1 - f_i^{N-1})\big)$. ■

With more information obtained from other nodes, the N-layer query ($N \geq 2$) dramatically improves query accuracy compared with the local query. Fig. 4 simulates the FPP of a N-layer query ($1 \leq N \leq 6$), where the supply chain is comprised of a full binary tree. It is clear that the FPP decreases with N. When the FPP of the local query is 0.1, the FPP of the 3-layer query tends to be just under 0.01. Similarly, the 4-layer FPP falls to less than 0.05 with the local FPP being 0.3.

Note that the computation process of f_L^N is bottom-up throughout the whole supply chain structure. Meanwhile, the initial N in f_L^N is 1 and it increases by 1 each time. This process continues until $f_L^N = 0$ (f_L^m must be 0 when $m > N$). Take Fig. 3 as an example. C, D, F and G are all terminal nodes, so $f_C^1 = f_D^1 = f_F^1 = f_G^1 = 0$. C, D, F and G stop computing. Then for the node E, $f_E^1 = f_E$, $f_E^2 = f_E(1 - \prod_{i \in \{F,G\}}(1 - f_i)) = 0$. E stops computing. After that, B has $f_B^1 = f_B$, $f_B^2 = f_B(1 - \prod_{i \in \{D,E\}}(1 - f_i)) = f_B f_E$, $f_B^3 = f_B(1 - \prod_{i \in \{D,E\}}(1 - f_i^2)) = 0$. B stops computing. Finally, A has $f_A^1 = f_A$, $f_A^2 = f_A(1 - \prod_{i \in \{B,C\}}(1 - f_i)) = f_A f_B$, $f_A^3 = f_A(1 - \prod_{i \in \{B,C\}}(1 - f_i^2)) = f_A f_B^2 = f_A f_B f_E$ and $f_A^4 = f_A(1 - \prod_{i \in \{B,C\}}(1 - f_i^3)) = 0$. A stops computing.

Once an existence query is issued at the node L with expectant query accuracy F ($0 < F \leq 1$), we can minimize i satisfying the following inequation:

$$f_L^i \leq (1 - F) < f_L^{i-1} \tag{4}$$

where, $i \geq 1$ and $f_L^0 = 1$. Moreover, if $i = H(L)$, then $f_L^i = 0$. In other words, we can definitely guarantee 100% query accuracy when the query is allowed to be forwarded to terminal nodes.

VI. Performance Analysis

In this section, we analyze key performance metrics for our storage model and query processing.

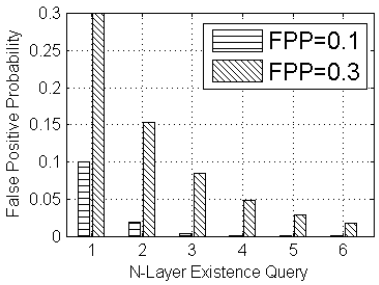


Fig. 4: FPP of the N-layer existence query

A. Storage Space

In our storage model, every node is required to construct a Bloom filter with m bits and p hash functions for storing tags' EPCs. As a result, m is supposed to be minimized as it acts as major storage space of the Bloom filter. Assume that there are n tags[1] to be stored at a node L and the EPC is 96 bits. According to Theorem 1, we have $m = \frac{-np}{ln(1 - {f_L}^{1/p})}$. Let the first derivative of m be zero, we have the minimum m when $p = \frac{-lnf_L}{ln2}$. Because p is an integer, we set $p = \lceil \frac{-lnf_L}{ln2} \rceil$ in our model. Thus, we have

$$min(m) = \Big\lceil \frac{-n\lceil \frac{-lnf_L}{ln2} \rceil}{ln(1 - {f_L}^{1/\lceil \frac{-lnf_L}{ln2} \rceil})} \Big\rceil \tag{5}$$

Apart from the Bloom filter, L will also be required to reserve raw RFID data in its epc_table if it is a terminal node. Therefore, the storage space tends to be $(96 + k)n$ (k bits for storing product information expect for the EPC, if desired). In summary, if L is an internal node, the storage space will be $\lceil \frac{-np}{ln(1 - {f_L}^{1/p})} \rceil$. Otherwise, the storage space will be $\lceil \frac{-np}{ln(1 - {f_L}^{1/p})} \rceil + (96 + k)n$, where $p = \lceil \frac{-lnf_L}{ln2} \rceil$.

B. Storage Efficiency

Storage efficiency is defined as the average storage time for locally storing each tag's information. Given a Bloom filter consisting of an m-bit array and p hash functions, the storage efficiency remains stable at $O(p)$ regardless of tag cardinality. Therefore, for a node L, the storage efficiency will be $O(p_L)$ if L is an internal node but tends to be $(O(p_L)+S_L)$ (S_L changing over tag cardinality is the average storage time for saving a tag's information in L's epc_table) otherwise.

C. Query Time & Query Overhead

Query time and query overhead are both important performance indicators to evaluate a query. As mentioned, our query processing supports two types of queries, existence queries and path queries. A path query launched at L comprises three parts: L, $upstream\ path$ and $downstream\ path$. However, execution time and overhead of the path query are mainly composed of the downstream path query as the upstream path query is required to be executed only once. Furthermore, according to the query processing in Section IV, the downstream path query at L can be actually treated as $H(L)$-layer query

[1]If a priori knowledge of n is unknown, we can construct the Bloom filter using Scalable Bloom Filters [15], which cost a little extra space to accommodate dynamic tag cardinality. The Scalable Bloom Filter is actually a variant of Bloom filters and its theoretical analysis can be found in [15], we therefore focus on analyzing the general case with a prior n in this paper.

without considering negligible differences of communication packages. Hence, this subsection focuses on analyzing query time and overhead for only the N-layer query. Table III lists the notations to be used below.

TABLE III: NOTATIONS OF PARAMETERS

Parameters	Definitions
t_c	communication time between two nodes
$p(t,i)$	probability that the tag t is the node i's member
$T_E(i)$	time of local existence query at the node i
$T_E(i,n)$	time of n-layer existence query at the node i
$T_{E1}(i,n)$	time of n-layer query for a member tag at i
$T_{E0}(i,n)$	time of n-layer query for a non-member tag at i

1) Query Time: Query time is defined as the average time for executing a query. We first analyze query time of a local query (1-layer query). If L is an internal node, the query time $T_E(L)$ will be $O(p)$. By contrast, if L is a terminal node and the query returns $true$ (an FP may occur), the query will continue to be executed in L's epc_table. In order to improve query efficiency, we suppose that a B-tree index, one of the most common indexes, is created, so the query time tends to be $O(log(n))$. Therefore, we have the local query time:

$$T_E(L) = \begin{cases} O(p) & if\ L\ is\ an\ internal\ node \\ O(p) + \big(\mu + (1-\mu)f_L\big)O\big(log(n)\big) & otherwise \end{cases}$$

where, $\mu = p(t,L)$.

We then analyze query time of an N-layer query ($2 \le N \le H(L)$) according to whether t is L's member. If t is in L's Bloom filter, the query will definitely be forwarded along t's trajectory, like $L_1 \to L_2 \to ... \to L_k$ ($1 \le k \le N$). However, k relying on both t and the FP varies a lot under different queries. We instead explore the upper bound of expected query time, letting $k = N$. Suppose that the query goes through one of the longest communication links, like $L_1(L{=}L_1) \to L_2 \to ... \to L_N$ and the query result is feed back from L_{i+1} to L_i ($1 \le i < N$). The maximal query time for a member tag at L is:

$$\begin{aligned} T_{E1}(L,N) &= \sum_{i=1}^{N-1}(T_E(L_i)+t_c) + T_E(L_N) + (N-1)t_c \\ &= \sum_{i=1}^{N} T_E(L_i) + 2(N-1)t_c \end{aligned} \quad (6)$$

On the other hand, if t does not belong to L's Bloom filter (it cannot be the member of L's descendant nodes), the query will go down the next layer only when FPs occur. Thus, an N-layer query for t may be completed at the k^{th} ($1 \le k \le N$) layer nodes (the k^{th} layer nodes participating the query all return $false$). Suppose that the communication link is $L_1 \to L_2 \to ... \to L_k$ ($1 \le k \le N$). Theorem 3 shows the expected query time.

Theorem 3: The expected query time of an N-layer query for the non-member tag t at the node L is:

$$\begin{aligned} T_{E0}(L,N) &= T_E(L)(1-f_L) + f_L^{N-1}\Big(\sum_{i=1}^{N} T_E(L_i) + 2(N-1)t_c\Big) \\ &\quad + \sum_{k=2}^{N-1}\left[(f_L^{k-1} - f_L^k)\Big(\sum_{i=1}^{k} T_E(L_i) + 2(k-1)t_c\Big)\right] \\ &= T_E(L) + \sum_{i=1}^{N-1} f_L^i\big(T_E(L_{i+1}) + 2t_c\big) \end{aligned} \quad (7)$$

Proof: L first executes the local query. If it returns $false$ (the probability is $1 - f_L$), the query terminates immediately. Thus, we have the first term $T_E(L)(1-f_L)$. If the FP occurs, the query will go down the next layer. Then, the probability that the N-layer query is terminated at the k^{th} ($2 \le k \le N-1$) layer nodes (the k^{th} layer nodes participating the query all return $false$) is $(f_L^{k-1} - f_L^k)$. That is because, f_L^{k-1} as the probability that the $(k-1)$-layer query returns $true$, means the query will continue to go to the k^{th} layer. Similarly, f_L^k indicates the probability that the query goes to the $(k+1)^{th}$ layer at least. Hence, $(f_L^{k-1} - f_L^k)$ is the probability that the N-layer query is terminated exactly at the k^{th} layer. The corresponding query time is $(\sum_{i=1}^{k} T_E(L_i) + 2(k-1)t_c)$. Hence, we have the expected query time by multiplying the two terms, that is, $(f_L^{k-1} - f_L^k)(\sum_{i=1}^{k} T_E(L_i) + 2(k-1)t_c)$. Note that the probability that the query goes to the N^{th} layer is f_L^{N-1}, since it must be terminated no matter what the N^{th} layer nodes return. Finally, we have $T_{E0}(L,N)$ by summing up all the terms. ■

The expected query time of the N-layer query for a tag t at the node L is:

$$T_E(L,N) = T_{E1}(L,N)p(t,L) + T_{E0}(L,N)(1-p(t,L)) \quad (8)$$

2) Query Overhead: In general, the query overhead is affected by a range of factors, such as communication and computation resources. For simplicity, we measure the query overhead with the number of nodes involved in the query as they are positively correlated. Table IV lists the notations will be used below.

TABLE IV: NOTATIONS OF PARAMETERS

Parameters	Definitions
$C(i,n)$	overhead of n-layer query at the node i
$C_1(i,n)$	overhead of n-layer query for member tag at i
$C_0(i,n)$	overhead of n-layer query for non-member tag at i

Similar to query time, the query overhead depends on a tag's membership. If an N-layer query for a non-member tag t is issued at L, the query will go down the next layer only when the local FP occurs. Thus, the probability that only L participates in the query is $(1-f_L)$. Otherwise, L's children will be involved in the (N-1)-layer query. Therefore, we have the recursion formula:

$$\begin{aligned} C_0(L,N) &= (1-f_L) + f_L\left[1 + \sum\nolimits_{i \in D(L)} C_0(i,N-1)\right] \\ &= 1 + f_L \sum\nolimits_{i \in D(L)} C_0(i,N-1) \end{aligned} \quad (9)$$

where $C_0(i,n) = 1$, when $n = 1$ or i is a terminal node.

On the other hand, suppose that there is a member tag t whose trajectory starting from L is $L_1(L) \to L_2 \to ... \to L_k$ ($1 \le k \le N$). Without FNs occurring in our model, the query must be transmitted along the trajectory, leading to L_i's participation ($1 \le i \le k$). Meanwhile, L_i's siblings (nodes with same parent node) also participate the query due to the fact that the query from L_{i-1} is sent to all its children including L_i. Moreover, the query will be forwarded if FPs occur at these siblings, which is equivalent to the query for a non-member tag t at these nodes. Therefore, we have the recursion formula

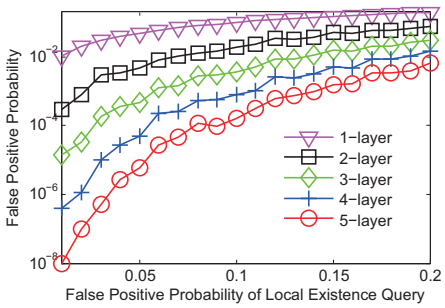


Fig. 5: False positive probability of the N-layer query

of query overhead for a member tag at the node L:

$$C_1(L_i, N) = \sum_{\substack{j \neq L_{i+1} \\ j \in D(L_i)}} C_0(j, N-1) + C_1(L_{i+1}, N-1) + 1 \tag{10}$$

where, $1 \leq i < k$ and $C_1(L_k, 1) = 1$.

In the end, we have the expected query overhead for a tag t at the node L:

$$C(L, N) = C_1(L, N)p(t, L) + C_0(L, N)(1 - p(t, L)) \tag{11}$$

VII. Evaluation

In this section, we first simulate the key performance indicators, i.e. FPP, query time and query overhead. We then compare our storage model with the traditional distributed approach by real experiments.

A. Simulation

In the simulation, the supply chain structure is comprised of a full binary tree with 2^{18} tags flowing from the root to terminal nodes. Consider any internal node L in the tree. Half of the products at L are distributed to L's left child, and the remaining products are distributed to L's right child. Every node stores related information of tags according to our storage model. In each simulation, the FPPs of local queries at all nodes are set to be the same. Along with above settings, we simulate the FPP, query time and query overhead.

1) FPP of the Ouery: To simulate FPPs, 10^8 non-member tags generated randomly are treated as our test set. We use the variable $total$ to count the $ture$ returns (FPs occur) and then the FPP is calculated by $total/10^8$ in each simulatioin. Fig. 5 shows FPPs of N-layer queries ($1 \leq N \leq 5$) under different local FPPs. The FPP of the N-layer query experiences a steady increase with the local FPP, which is in accordance with our intuition and theoretical derivation. In contrast, the FPP of the N-layer query decreases as N increases, confirming that the N-layer query can efficiently improve the query accuracy. Table V contrasts the theoretical value with the simulative FPPs of N-layer queries ($1 \leq N \leq 5$) under different local FPPs. The data show that the corresponding FPPs are close. Note that the simulative value is supposed to fluctuate up and down around theoretical value. However, the figure for simulation is always a little bigger than theory. There are two reasons for this case. First, all hash functions do not distribute uniformly, resulting in deviating ideal expectation. Second, the total number of hash functions is limited in the simulation, so the parent and children are likely to share same hash functions, leading to the increase of FPPs.

TABLE V: COMPARISON OF FPPS

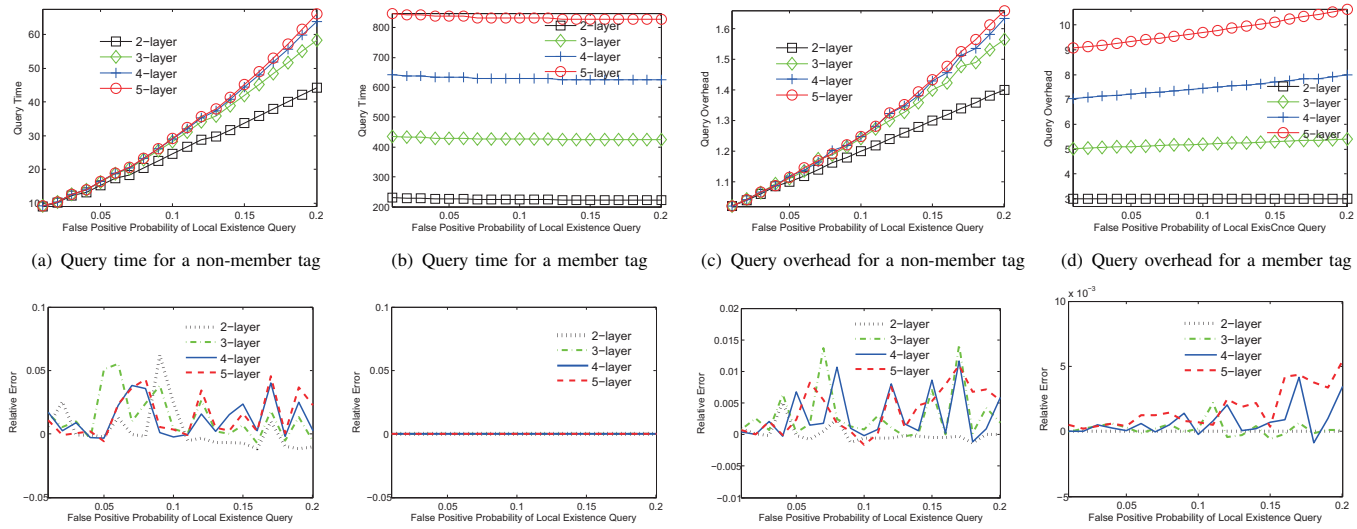
		FPPs of N-layer existence queries				
1-L	theory	0.04	0.08	0.12	0.16	0.20
	simulation	0.0402	0.081	0.12	0.16	0.199
2-L	theory	0.0031	0.012	0.027	0.0471	0.072
	simulation	0.0033	0.013	0.033	0.0474	0.076
3-L	theory	2.5e-4	2.0e-3	6.4e-3	0.015	0.028
	simulation	3.5e-4	2.7e-3	8.4e-3	0.015	0.03
4-L	theory	2.0e-5	3.1e-4	1.5e-3	0.0047	0.011
	simulation	2.8e-5	5.2e-4	2.5e-3	0.0048	0.014
5-L	theory	1.6e-6	5.0e-5	3.7e-4	1.5e-3	4.4e-3
	simulation	2.7e-6	1.2e-4	6.3e-4	1.6e-3	6.8e-3

2) Query Time: Query time consists of computing time as well as communicating time. Generally, remote communication takes longer time than local computation due to the network latency. Assume that the average peer-to-peer communication time is $O(100)$ and querying the epc_table and Bloom filter takes $O(log(n))$ (n is the number of tags) and $O(p)$ (p is the number of hash functions) respectively. In Fig. 6(a), we query 10^6 non-member tags in each simulation and calculate average query time of the N-layer query ($2 \leq N \leq 5$). As it can be seen, the time is positively relevant to the local FPP and N. First, the bigger the local FPP is, the more likely an FP incurs. Thus, it is easier for the query to go down the next layer, increasing query time. Second, the query is allowed to be forwarded to more layers with the growth of N, leading to the rise of execution time. Fig. 6(e) contrasts the simulative time with corresponding theoretical time. The error rate is controlled within 7% that verifies our theoretical deduction about query time for non-member tags. By contrast, in Fig. 6(b), we query 2^{18} member tags in each simulation and obtain the average query time. The query time decreases with the local FPP since the bigger the local FPP is, the less local query time is. Fig. 6(f) shows that the simulative time is equal to theoretical time, resulting from the fact that the query always goes to terminal nodes (no FNs), which definitely reaches the maximal query time (theoretical value).

3) Query Overhead: The overhead of the N-layer query at the node L relies on whether the tag has stayed in L. Fig. 6(c) queries 10^6 non-member tags in each simulation, and gets the average query overhead of the N-layer query ($2 \leq N \leq 5$). In contrast, Fig. 6(d) queries 2^{18} member tags and obtains the average number of nodes involved in the query. The overhead increases with the local FPP as well as N. For one thing, the high local FPP increases the chance for a query to go down the next layer. For another, the query is allowed to be forwarded to more nodes with the increase of n. Note that the overhead of the 2-layer query in Fig. 6(d) remains unchanged, since the query for a member tag at L will definitely be forwarded to all L's children regardless of the local FPP. Fig. 6(g) and Fig. 6(h) contrast the simulated overhead with the corresponding theoretical overhead for non-member and member tags respectively. Both small relative errors verify that our theoretical deduction about query overhead.

B. Experiment

This subsection compares our storage model and query processing with the traditional distributed approach in terms of storage space, storage time and local query time. We experiment on an Intel Dual Core 2.5G CPU with 12G memory



(a) Query time for a non-member tag (b) Query time for a member tag (c) Query overhead for a non-member tag (d) Query overhead for a member tag

(e) Relative error of query time for a non-member tag (f) Relative error of query time for a member tag (g) Relative error of query overhead for a non-member tag (h) Relative error of query overhead for a member tag

Fig. 6: Verification of query time and query overhead

and 7200 RPM hard drive using Java. Meanwhile, considering the traditional distributed storage, we use Oracle Database 11g Release 2 to locally store raw EPCs that a node captures and create a default index based on the attribute *epc* in the database. Due to the lack of a well known commercial RFID data set, we generate 96-bit EPCs randomly. The scale of our data set ranges from 100,000 to 10,000,000.

1) Storage Space: The space of a Bloom filter depends on the expected error rate. As seen from Fig. 7(a), the storage space per EPC varies a lot under different local FPPs. The horizontal ordinate is FPP=$2^i \times 10^{-4}$ ($0 \leq i \leq 11$) ranging from 10^{-4} to 10^{-1}. The traditional storage remains stable at 96 bits, whereas storage space using Bloom filters decreases with the local FPP. For example, when FPP=10^{-4} (i=0), our model needs 20 bits per EPC. When FPP increases to 10^{-1}, it needs no more than 5 bits per EPC, which compresses the data about 20 times. Therefore, our storage model is space-efficient at the cost of a small error rate.

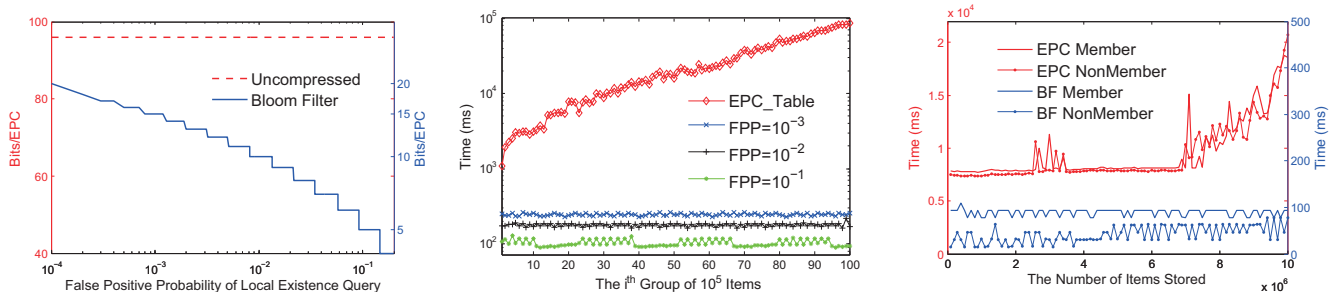
2) Storage Time & Local Query Time: Fig. 7(b) compares our model with traditional one under different data volume in terms of storage time. In each simulation, we insert 10^5 EPCs (the horizontal axis indicates the i^{th} group of 10^5 EPCs) into the Bloom filter and epc_table respectively. Due to the great space saving, the Bloom filter is put in the main memory compared with epc_table stored in the hard disk. In the figure, the storage time almost increases exponentially with the data scale in traditional storage (vertical coordinate is exponential), because the database has to maintain indexes for storing new EPCs, leading to great storage overhead. In contrast, the storage time of Bloom filters with different FPPs (10^{-3}, 10^{-2}, 10^{-1}) fluctuates around different constants respectively, regardless of the data volume. The FPP=0.01 is 8 times more efficient than traditional scheme at least. Moreover, the storage time decreases with the local FPP, resulting from the reason that the higher FPP is, the less hash functions are required.

Fig. 7(c) compares query time between our model and the traditional one. In each simulation, we query 10^5 member and non-member EPCs based on epc_tables and BFs and then obtain total query time. The query time of traditional storage (left vertical coordinate) for member tags (EPC Member) and non-member tags (EPC NonMember) varies a lot with different number of items. When the number is less than 7×10^6, the time almost remains stable, with the exception of a peak at around 3×10^6. The steady trend is due to the fact that the difference of computation time is negligible facing small-scale EPCs, while the peak is likely to result from the index adjustment executed by oracle database. When the number is greater than 7×10^6, the time generally sees a sharp rise. In contrast, FPP=0.01 (right vertical coordinate) for member tags (BF Member) and non-member tags (BF NonMember) does not change with the data scale. The former one is longer than the latter, since the query for a member tag must check all hash functions. Instead, it will return immediately if any bit is 0 for a non-member tag. Our queries are faster than traditional queries more than 70 times. In summary, our storage model and query processing with Bloom filters are time-efficient.

VIII. Related Work

Existing research work on RFID data management falls into two areas. One is concerned with data processing. An adaptive smoothing filter SMURF for RFID data cleaning was proposed to provide accurate RFID data to applications in [16]. Rao et al. [17] designed a deferred approach for detecting and correcting RFID data anomalies by utilizing declarative sequenced-based rules. Chen et al. [18] proposed a Bayesian inference based approach for cleaning RFID raw data with data redundancy and prior knowledge. Mahdin et al. [19] presented a data filter method that efficiently detected and removed duplicate readings.

The other area is about data storage and query processing. A warehousing model was introduced in [8] that preserved object transitions while providing significant compression and path-dependent aggregates. Wang et al. [13] established Dynamic Relationship ER model (DRER) to track and monitor RFID data. Lee et al. [11] proposed an effective path encoding



(a) Comparison of storage space (b) Comparison of storage time (c) Comparison of query time

Fig. 7: Comparison of the performance indicators

scheme to encode the flow information for products. The VG-curve combined with Multidimensional Dynamic Clustering Primary Index was used to efficiently access multidimensional data in [20]. Chawathe et al. [10] suggested a layered architecture for managing RFID data. All these approaches assumed that RFID data were stored within a single data repository. The traceability networks with a new architecture and algorithms for processing traceability queries were introduced in [21]. Cao [22] designed a scalable and distributed stream processing system for RFID tracking and monitoring by combining location and containment inference with stream query processing. These work [21], [22], however, do not consider distributed RFID data storage and fast query.

IX. Conclusions

The big data in RFID applications has posed new requests for distributed data storage and query processing support. In this paper, we design an efficient distributed storage model leveraging Bloom filters and establish corresponding query processing schemes. Our primary objective is to make large-scale RFID data management more space-efficient and time-efficient under any query accuracy requirement. Two kinds of most popular queries, i.e., existence query and path query, can be efficiently supported in the newly proposed model. Theoretical analysis and experiments validate the significant improvement of our model over the traditional distributed storage approach. With the expense of 1% local false positive probability, our model can reduce the storage space by a factor of 10 and improve the local query efficiency 70 times.

Acknowledgment

This research is supported in part by National Natural Science Foundation of China (No. 60873026, 61272418, 61373181), the National Science and Technology Support Program of China (No. 2012BAK26B02), Industrialization of Science Program for University of Jiangsu Province (No. JH10-3) and HK RGC PolyU 5281/13E.

References

[1] R. Angeles, "RFID technologies: Supply-chain applications and implementation issues," *Information Systems Management*, vol. 22, no. 1, pp. 51–65, 2005.

[2] W. Luo, S. Chen, T. Li, and S. Chen, "Efficient missing tag detection in RFID systems," in *Proc. of IEEE INFOCOM*, 2011, pp. 356–360.

[3] S. Chen, M. Zhang, and B. Xiao, "Efficient information collection protocols for sensor-augmented RFID networks," in *Proc. of IEEE INFOCOM*, 2011, pp. 3101–3109.

[4] H. Yue, C. Zhang, M. Pan, Y. Fang, and S. Chen, "A time-efficient information collection protocol for large-scale RFID systems," in *Proc. of IEEE INFOCOM*, 2012, pp. 2158–2166.

[5] K. Bu, B. Xiao, Q. Xiao, and S. Chen, "Efficient misplaced-tag pinpointing in large RFID systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 11, pp. 2094–2106, 2012.

[6] Y. Zheng and M. Li, "ZOE: Fast cardinality estimation for large-scale RFID systems," in *Proc. of IEEE INFOCOM*, 2013, pp. 908–916.

[7] M. Chen, W. Luo, Z. Mo, S. Chen, and Y. Fang, "An efficient tag search protocol in large-scale RFID systems," in *Proc. of IEEE INFOCOM*, 2013.

[8] H. Gonzalez, J. Han, X. Li, and D. Klabjan, "Warehousing and analyzing massive RFID data sets," in *Proc. of ICDE*, 2006, pp. 83–92.

[9] Y. Bai, F. Wang, P. Liu, Z. Carlo., and S. Liu, "RFID data processing with a data stream query language," in *Proc. of ICDE*, 2007, pp. 1184–1193.

[10] S. S. Chawathe, V. Krishnamurthy, S. Ramachandran, and S. Sarma, "Managing RFID data," in *Proc. of VLDB*, 2004, pp. 1189–1195.

[11] C.-H. Lee and C.-W. Chung, "RFID data processing in supply chain management using a path encoding scheme," *IEEE Trans. on Knowl. and Data Eng. (TKDE)*, vol. 23, no. 5, pp. 742–758, 2011.

[12] D. Lin, H. Elmongui, E. Bertino, and B. Ooi, "Data management in RFID applications," in *Database and Expert Systems Applications*, vol. 4653, 2007, pp. 434–444.

[13] F. Wang and P. Liu, "Temporal management of RFID data," in *Proc. of VLDB*, 2005, pp. 1128–1139.

[14] B. Xiao and Y. Hua, "Using parallel bloom filters for multiattribute representation on network services," *IEEE Transactions on Parallel and Distributed Systems*, vol. 21, pp. 20–32, 2010.

[15] P. S. Almeida, C. Baquero, N. Preguica, and D. Hutchison, "Scalable bloom filters," *Information Processing Letters*, vol. 101, no. 6, pp. 255 – 261, 2007.

[16] S. R. Jeffery, M. Garofalakis, and M. J. Franklin, "Adaptive cleaning for RFID data streams," in *Proc. of VLDB*, 2006, pp. 163–174.

[17] J. Rao, S. Doraiswamy, H. Thakkar, and L. S. Colby, "A deferred cleansing method for RFID data analytics," in *Proc. of VLDB*, 2006, pp. 175–186.

[18] H. Chen, W.-S. Ku, H. Wang, and M.-T. Sun, "Leveraging spatio-temporal redundancy for RFID data cleansing," in *Proc. of ACM SIGMOD*, 2010, pp. 51–62.

[19] H. Mahdin and J. Abawajy, "An approach for removing redundant data from RFID data streams," *Sensors (Basel, Switzerland)*, vol. 11, no. 10, pp. 9863–9877, 2011.

[20] J. Terry, B. Stantic, and A. Sattar, "Indexing RFID data using the vg-curve," in *Australasian Database Conference*, 2012, pp. 117–126.

[21] R. Agrawal, A. Cheung, K. Kailing, and S. Schönauer, "Towards traceability across sovereign distributed RFID databases," in *Proc. of IDEAS*, 2006, pp. 174–184.

[22] Z. Cao, C. Sutton, Y. Diao, and P. Shenoy, "Distributed inference and query processing for RFID tracking and monitoring," *Proceedings of the VLDB Endowment*, vol. 4, no. 5, pp. 326–337, 2011.