Efficiently Filtering RFID Data Streams

Yijian Bai †* Fusheng Wang ‡ Peiya Liu ‡

†UCLA
bai@cs.ucla.edu

‡ Siemens Corporate Research
{fusheng.wang,peiya.liu}@siemens.com

Abstract

RFID holds the promise of real-time identifying, locating, tracking and monitoring physical objects without line of sight, and can be used for a wide range of pervasive computing applications. To achieve these goals, RFID data has to be collected, filtered, and transformed into semantic application data. RFID data, however, contains false readings and duplicates. Such data cannot be used directly by applications unless they are filtered and cleaned. While RFID data often arrives quickly and is in high volume, its detection usually demands efficient processing, especially for those real-time monitoring applications. Meanwhile, the order preservation of RFID tag observations are critical for many applications. In this paper, we propose several effective methods to filter RFID data, including both noise removal and duplicate elimination. Our performance study demonstrates the efficiency of our methods.

1 Introduction

RFID (radio frequency identification) technology uses radio-frequency waves to transfer data between readers and movable tagged objects. Thus it is possible to create a physically linked world in which every object can be numbered, identified, cataloged, and tracked. RFID is automatic and fast, and does not require line of sight or contact between readers and tagged objects. With such significant technology advantages, RFID has been gradually adopted and deployed in a wide area of applications, such as access control, library checkin and checkout, document tracking, smart box [1], highway tolls, supply chain and logistics, security, and healthcare [2].

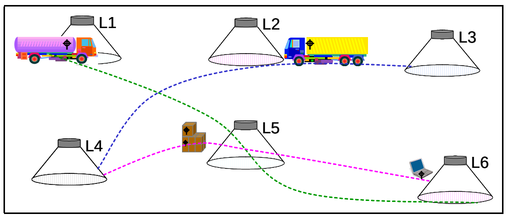

Figure 1: Pervasive Computing with RFID

One major problem to be solved in pervasive computing is to identify and track physical objects, and RFID technology is a perfect fit to solve this. By tagging objects with EPC [1] tags that virtually represent these objects, the identifications and behaviors of objects can be precisely observed and tracked. RFID readers can be deployed at different locations and networked together, which provides an RFID-based pervasive computing environment. This is illustrated in Figure 1, where L1 – L6 denote different locations mounted with readers. Tagged objects moving in this environment will then be automatically sensed and observed with their identifications, locations and movement paths.

Readers' observations, however, are raw data and can contain a lot of duplicate and false readings. Thus the first step to integrate RFID data into pervasive computing applications is to filter RFID observations.

RFID data are generated quickly and automatically, and can be used for real-time monitoring, or accumulated for object tracking. To filter the high volume real-time RFID data streams, efficient methods are essential, especially for real-time applications.

The filtered RFID data often need to preserve the original order, i.e., the first observed tagged object will be output first after filtering. Such order can be critical for many RFID applications. For example, a nurse uses a wearable reader to access RFID-tagged medical items according to medical procedures. The order the nurse accesses these medical items is critical: wrong orders may cause a medical error or even lead to fatal

*Work done while visiting Siemens Corporate Research.

CleanDB, Seoul, Korea, 2006

[1]EPC – Electronic Product Code – is an identification scheme for universally identifying physical objects, defined by EPC-Global [3].

result. Thus, the correct ordering of RFID observations together with a workflow monitoring system will minimize such errors.

In this paper, we propose effective and efficient algorithms for RFID data filtering, including noise removal and duplicate elimination.

The paper is organized as follows. We first introduce the background of RFID data filtering in Section 2. Then in Section 3 we propose algorithms to efficiently filter noise from RFID data, including the problem of order preservation in the output. Next we discuss algorithms for duplicate merging in Section 4. Performance study of these methods is discussed in Section 5, followed by Related Work and Conclusion.

2 Background

Due to the low-power and low-cost constraints of RFID tags, reliability of RFID readings is of concern in many circumstances [4, 5]. There are three typical undesired scenarios: *false negative readings*, *false positive readings* and *duplicate readings*, discussed as follows.

- *False negative readings.* In this case, RFID tags, while present to a reader, might not be read by the reader at all. This can be caused by i) When multiple tags are to be simultaneously detected, RF collisions occur and signals interfere with each other, preventing the reader from identifying any tags; ii) A tag is not detected due to water or metal shielding or RF interference.

- *False positive readings(or noise).* In this case, besides RFID tags to be read, additional unexpected readings are generated. This can be attributed to the following reasons. i) RFID tags outside the normal reading scope of a reader are captured by the reader. For example, while reading items from one case, a reader may read items from an adjacent case; ii) Unknown reasons from the reader or environment, for example, one of our readers periodically sends wrong IDs.

- *Duplicate Readings.* This can be caused by the following reasons: i) Tags in the scope of a reader for a long time (in multiple reading frames) are read by the reader multiple times; ii) Multiple readers are installed to cover larger area or distance, and tags in the overlapped areas are read by multiple readers; and iii) To enhance reading accuracy, multiple tags with same EPCs are attached to the same object, thus generate duplicate readings.

In practice, readings are often performed in multiple cycles to achieve higher recognition rate [5]. In this way, false negative readings can be significantly reduced. Meanwhile, noisy readings (or false positive readings) generally have a low occurrence rate compared to normal true readings. Thus only those readings that have significant repeats within certain interval are considered to be true readings. This, however, will further produce much more duplicate readings.

Based on above observations, we develop effective and efficient RFID data filtering techniques to generate clean RFID data, which can be further interpreted and integrated into RFID-based applications. In this paper, we study two types of filtering: noise is removed from RFID data (*denoising or smoothing*), and duplicates are merged into one distinct reading (*duplicate elimination, or merging*). We develop algorithms that, compared to baseline implementations, work more efficiently while requiring less buffer space for history storage for both *denoising* and *duplicate elimination*. Furthermore, we discuss the issue of output time ordering for *denoising* and show our method can address this issue efficiently.

3 Denoising in RFID Data Streams

Based on the discussion above, since multiple reading cycles are performed on tagged objects and noise readings normally have a low occurrence rate, we propose *sliding window* based approaches to solve the problem. A sliding window is a window with certain size that moves with time. Suppose the window with size `window_size` has a time coordinate of $[t_1, t_1 + \texttt{window\_size}]$, after τ, the coordinate will become $[t_1 + \tau, t_1 + \texttt{window\_size} + \tau]$.

RFID reading tuples will enter the window and get expired as time moves. Therefore, the noise readings are readings with count of distinct tag EPC values below a noise `threshold`. Denoising essentially performs the following operations: within any time window with size of `window_size` surrounding an RFID reading, if the count of the readings with same tag EPC values appears equal to or above `threshold`, then the observed EPC value is not noise and needs to be forwarded for further processing; otherwise the reading is discarded. Two parameters used here are `window_size` of a sliding time window, and a `threshold` for noise detection.

An RFID observation (reading) is in the form of: (`reader_id, tag_id, timestamp`), which refers to the EPC [3] of the RFID reader, the EPC of the tagged object, and the timestamp of this observation respectively. In the algorithms presented below, the key of a reading can be usually considered to be the pair of (`reader_id, tag_id`) in the reading. For the case where multiple readers are used to observe same tags, the key will be `tag_id`.

Baseline Denoising: A Base Approach We first show a baseline implementation of denoising as shown in Algorithm 1, which we refer to as `baseline_denoising`.

In this algorithm, intuitively, for each incoming

Algorithm 1 Baseline_denoise (params: `window_size`, `threshold`)

```
1: WINDOWBUFFER ← empty queue {FIFO queue to hold sliding window of readings}
2: loop {loop forever for next incoming reading}
3:   INCOMING ← the next reading
4:   append INCOMING to the end of WINDOWBUFFER
5:   EXPIRETIME ← INCOMING.timestamp - window_size
6:   while the head of WINDOWBUFFER is older than EXPIRETIME do
7:     remove the head of WINDOWBUFFER
8:   end while
9:   COUNT ← count of readings in WINDOWBUFFER whose key equals to INCOMING.key
10:  if COUNT ≥ threshold then
11:    for each of the reading R in WINDOWBUFFER with key equals to INCOMING.key do
12:      if R has not been output before then
13:        output R
14:        set STATE-OF-OUTPUT as true
15:      end if
16:    end for
17:  end if
18: end loop
```

reading of value `R`, we perform a full scan of the preceding sliding time window of size `window_size`. If `R` appears more than `threshold` times within the window, we know this is not a noise reading thus we output every `R` in the window. To ensure a particular reading is never output more than once, we keep a `state-of-output` with each reading in the window buffer and set it to *true* once it is output once.

Complexity. Assume on average there are `n` readings in the sliding window, with `k` distinct keys. Since the operations are repeated for each incoming tag reading, we analyze the time cost on a per-reading basis. The bulk of the time cost is from 4 operations: inserting the incoming reading into the window, removing expired readings from the window, computing the count of the readings with the same key, setting `state-of-output` and outputting readings of the same key if threshold condition is satisfied. Since all readings are maintained in the same FIFO (First-In, First-Out) queue, both insertion of new readings (appending to the end of queue) and removal of expired readings (removing from the head of queue) can be considered constant time ($O(1)$) operations. (Strictly speaking, expiration is amortized ($O(1)$) per incoming reading here, since on average there is only one expiration per new arrival, although individual incoming reading may trigger different number of expirations.) On the other hand, both counting and setting `state-of-output` is performed by linearly scanning the full window. Counting is always performed for each incoming reading, thus the cost is $\Theta(n)$. Setting `state-of-output` and outputting only occur when threshold condition is satisfied, thus the cost can be considered to be bounded by $O(n)$, which leaves the total cost per incoming reading to be $O(1)+O(1)+\Theta(n)+O(n) = \Theta(n)$.

Space Cost. The space cost for the *baseline_denoise* algorithm is basically the storage for the sliding window itself, thus $\Theta(n)$.

It is natural to see that, with some additional space cost, we can incrementally maintain an extra counter for each distinct tag EPC value using a hashtable (which takes $\Theta(k)$ space), thus reduce the counting cost for each incoming key value. That is, for each incoming reading we increment the counter for the corresponding key in the hashtable, and for each expired reading we decrement the counter for the corresponding key. This reduces counting to an $O(1)$ operation, although we still can not avoid the $O(n)$ operation of setting `states-of-output` and outputting readings.

3.1 Lazy Denoising with Output Order Preserving Using Hashtable

There is one problem in the *baseline_denoise* algorithm: the output readings may be out of order if we output immediately upon determining a reading is non-noise, i.e., a reading observed earlier may be output later. This affects all further RFID data processing where correct ordering of observations is critical, such as complex RFID event detections for real-time RFID applications and RFID data aggregation [6]. For example, we may need to detect a certain sequence of events, A followed by B, if the order is reversed an alert has to be raised. In this scenario, not preserving output ordering of tags will result in both false alerts and false acceptances.

The following example shows how this out-of-order problem might happen.

Example 1: out-of-order observations. Suppose two tags are being read at two readers attached to the same host computer. Each tag is repeated 10 times with an interval of 100msec, thus the window size here is 1000msec. A reading is considered to be non-noise if it appears 6 times out of any 1000msec time-window around it. Assume the two tag keys are 1 and 2, and the actual readings appear in sequences as shown in Table 1, where tag 2 arrives 100msec later after tag 1 arrives. The readings of 4, 5, 8, 9 are noise[2].

Although tag 1 and tag 2 both have 2 noise readings in this example, due to different positions of the noise, ID 2 is actually determined as a non-noise reading first (at time 700msec), while ID 1 is determined as a non-noise later (at time 800msec), although tag 1 arrives earlier than tag 2. Therefore, if we output readings

[2]This example also illustrates how to set the `window_size` parameter for the algorithm. In most cases, this parameter is dictated by the repeat count of a tag, as well as the interval between repeats. The other parameter, `threshold`, however, will need to be tuned based on error rates.

Time(msec)	Tag 1 Reading	Tag 2 Reading
100	1	
200	4	2
300	1	2
400	1	2
500	5	2
600	1	2
700	1	2*
800	1*	8
900	1	2
1000	1	9
1100		2

Table 1: Arrival Time of Readings for Tag 1 and Tag 2 (* indicates the earliest time point that the reading can be determined as non-noise)

immediately after we detect them as non-noise, as is done in the *baseline_denoise* algorithm, we will then output readings with their timestamps out of order. If we represent the output as (`id, time`), then at time 700msec and 800msec the output for this example is:

```
Time 700: (2,200) (2,300) (2,400) (2,500) (2,600) (2,700)
Time 800: (1,100) (1,300) (1,400) (1,600) (1,700) (1,800)
```

Clearly, the reading of tag 1 at time 100msec will be output later than the reading of tag 2 at time 700mec. This will present a problem for any algorithm that is dependent on correct time-ordering of readings.

To solve the out-of-order problem, one solution is, when a reading is determined as non-noise, mark the reading as non-noise but not output it yet. The output happens only if a reading marked as non-noise gets expired from the window. With the FIFO queue for the window, it is therefore very efficient to output readings in their correct order.

Algorithm 2 – *Lazy_denoising* – incorporates the above-mentioned improvements. A hashtable of counters are maintained for each distinct key value `R` that is still present in the sliding window, and the corresponding counter is incrementally updated for each incoming tuple and expiring tuple. At any point of time, if the count of `R` in the window is higher than `threshold`, we mark all readings of `R` as non-noise. To ensure the correct output order, we delay the output of all non-noise tuples till they expire from the sliding window. At this point we know for sure all non-noise tuples will be in order, since the noise readings that have already expired will never turn to non-noise to affect the order.

Complexity. With incremental counter maintenance using a hashtable, the cost of counting operation for each incoming reading is reduced from Θ(n) to O(1), at the expense of an extra Θ(k) space. With output-on-expire, it guarantees that the output is in correct time order at no extra time or space cost. The cost of hashtable maintenance (inserting and removing keys from the hashtable) is on-average upper-bounded by O(1) for each incoming reading, and due to repeating, not every incoming reading will introduce a new key.

Notice that, in general, if each key is repeated for a fair amount of time (say 10 times, which is common in practice), and the noise ratio is small (say 1%), then `k` can be considered to be an order of magnitude smaller than `n`. As the noise ratio gets higher, the difference between `k` and `n` become smaller. If we assume each tag is repeated for r times, and overall there is a p percent chance that a reading is noise, then we have the relationship that $k = n * (\frac{1}{r} + p)$.

Baseline (Ordered). In the experiments section,

Algorithm 2 Lazy_denoising (params: `window_size`, `threshold`)

```
1: WINDOWBUFFER ← empty queue {FIFO queue to hold sliding window of readings}
2: TABLE ← empty hashtable {hashtable to map each key to a counter}
3: loop {loop forever for next incoming reading}
4:   INCOMING ← the next reading
5:   mark INCOMING as noise
6:   append INCOMING to the end of WINDOWBUFFER
7:   if the counter at TABLE[INCOMING.key] does not exist then
8:     store a counter at TABLE[INCOMING.key] with value 1
9:   else
10:    increment the counter at TABLE[INCOMING.key]
11:  end if
12:  EXPIRETIME ← INCOMING.timestamp - window_size
13:  while the head of WINDOWBUFFER is older than EXPIRETIME do
14:    if the head reading is marked as non-noise then
15:      output the head of WINDOWBUFFER
16:    end if
17:    remove the head of WINDOWBUFFER
18:    decrement the counter in TABLE for the corresponding key
19:    remove the slot in TABLE if the counter for this key becomes 0
20:  end while
21:  COUNT ← counter value at TABLE[INCOMING.key]
22:  if COUNT ≥ threshold then
23:    for each of the reading R in WINDOWBUFFER with key equals to INCOMING.key, by reverse time order do
24:      if R is marked as noise then
25:        mark R as non-noise
26:      else
27:        break the for loop
28:      end if
29:    end for
30:  end if
31: end loop
```

a *Baseline (Ordered)* algorithm is used for comparison with *Baseline_denoising* and *Lazy_denoising*. This algorithm is exactly the same as *Baseline_denoising* when searching for non-noise readings, as it scans the full window each time. However, it also tries to output tuples in order by only outputting a reading when it expires from the window. The details of this algorithm are omitted here since it is a straightforward extension of *Baseline_denoising* and has exactly the same complexity bounds.

3.2 Eager Denoising: Output Data Early with Order Preservation

Although output-on-expire is efficient and straightforward, it does have a negative consequence of introducing more delay for outputting readings. Instead of being output on the fly at the time of determination to be non-noise, a reading will not be output until it is expired from the sliding window. This could be a problem if the width of the window is quite long. This indeed can be improved for situations where a reading can be output earlier while correct time order can still be preserved.

In fact, the issue of order disturbance occurs only if a reading has been output before the change of labeling on some earlier reading from noise to non-noise within the window. Therefore, for a non-noise reading that we know no other earlier noise reading is present in the sliding window, we can then safely output it without the risk of order problems. This technique is incorporated in Algorithm 3 – *Eager_denoise*.

Algorithm *Eager_denoise* (Algorithm 3) improves over Algorithm *Lazy_denoise* (Algorithm 2) by outputting non-noise readings more eagerly: as soon as there is no more noise before the non-noise reading within the sliding window, the non-noise reading is output. To achieve this, the algorithm keeps track of the first noise reading (FIRSTNOISE) inside the window at all times. Then an invariant is kept at the end of processing each incoming reading, such that all the non-noise readings before FIRSTNOISE are output, and all the non-noise readings after FIRSTNOISE are not. (In the case of no presence of noise, everything is output at the end of the processing of the incoming reading). To maintain this invariant, each time FIRSTNOISE changes – either by expiring the reading out of the window, or due to setting of non-noise when its key appearance is more frequent than the threshold – we output all non-noise readings by time order until we find the next FIRSTNOISE in the window.

Therefore in this algorithm, in a nutshell, for each incoming reading and each expiring reading we incrementally update the corresponding counter for each distinct tag EPC value in the hashtable. Once the counter for value `R` is higher than `threshold`, we set all readings of R in the window to be non-noise. We immediately output the non-noise reading of value `R`

Algorithm 3 Eager_denoise (params: `window_size`, `threshold`)

```
1:  WINDOWBUFFER ← empty queue
2:  TABLE ← empty hashtable
3:  FIRSTNOISE ← null {keep earliest noise in window}
4:  loop {loop forever for next incoming reading}
5:    INCOMING ← the next reading
6:    mark INCOMING as noise
7:    if FIRSTNOISE = null then
8:      FIRSTNOISE ← INCOMING
9:    end if
10:   append INCOMING to end of WINDOWBUFFER
11:   if the counter at TABLE[INCOMING.key] does not exist then
12:     initiate TABLE[INCOMING.key] with counter 1
13:   else
14:     increment TABLE[INCOMING.key]
15:   end if
16:   EXPIRETIME ← INCOMING.timestamp - window_size
17:   SEARCHFIRST ← false
18:   while the head of WINDOWBUFFER is older than EXPIRETIME do
19:     if SEARCHFIRST = false ∧ the head reading is marked as noise then
20:       SEARCHFIRST ← true
21:       FIRSTNOISE ← null
22:     else if SEARCHFIRST = true ∧ the head reading is marked as non-noise then
23:       output the head of WINDOWBUFFER {this is a non-noise reading after the previous expired FIRSTNOISE}
24:     end if
25:     remove the head of WINDOWBUFFER
26:     decrement the counter in TABLE for the corresponding key
27:     remove the slot in TABLE for 0-counts
28:   end while
29:   COUNT ← counter value at TABLE[INCOMING.key]
30:   if COUNT ≥ threshold ∨ SEARCHFIRST = true then {If either the threshold condition is met, or we need a new FIRSTNOISE, scan the window}
31:     for each of the reading R still in WINDOWBUFFER according to time order do
32:       if COUNT ≥ threshold ∧ R.key = INCOMING.key ∧ R is marked as noise then
33:         if SEARCHFIRST = false ∧ R = FIRSTNOISE then
34:           SEARCHFIRST ← true
35:           FIRSTNOISE ← null
36:         end if
37:         mark R as non-noise
38:         if SEARCHFIRST = true ∨ R.timestamp < FIRSTNOISE.timestamp then
39:           output R {output the newly-determined non-noise reading, if either the next FIRSTNOISE is unknown, or it is earlier than the known FIRSTNOISE}
40:         end if
41:       else if R is non-noise ∧ SEARCHFIRST = true then
42:         output R {output the existing non-noise reading, only if the next FIRSTNOISE is not determined yet}
43:       else if SEARCHFIRST = true ∧ R is marked as noise then
44:         SEARCHFIRST ← false
45:         FIRSTNOISE ← R
46:         if COUNT < threshold then
47:           break the while loop
48:         end if
49:       end if
50:     end for
51:   end if
52: end loop
```

once we can determine that there are no more noise readings before this reading in the sliding window.

Complexity. Compared to *Lazy_denoise*, *Eager_denoise* performs one more operation: the maintenance of FIRSTNOISE. An extra linear search on the window is performed whenever FIRSTNOISE changes, and the search is obviously less frequent than one time per incoming reading. Therefore the bound of O(n) processing time per incoming reading still remains the same.

4 Duplicate Elimination (Merging)

When noise in the readings is eliminated, duplicate readings for the same tag have to be recognized and only the first (or the earliest) one among all duplicates should be retained. Our duplicate-elimination (or merging) algorithms take one parameter – `max_distance`. If a reading is within `max_distance` in time from the previous reading with the same key, then this reading is considered a duplicate. Otherwise, it is considered a new reading and is output.

Algorithm 4 – *baseline_merge* – performs duplicate elimination by simply keeping a sliding-window of size `max_distance`. For each incoming reading, if there exists another reading in the window with the same key, then it is considered a duplicate, otherwise it is output as a new reading.

Algorithm 4 Baseline_merge (param: *max_distance*)

```
1: WINDOWBUFFER ← empty queue {FIFO queue to hold sliding window of readings}
2: loop {loop forever for next incoming reading}
3:   INCOMING ← the next reading
4:   EXPIRETIME ← INCOMING.timestamp - max_distance
5:   while the head of WINDOWBUFFER is older than EXPIRETIME do
6:     remove the head of WINDOWBUFFER
7:   end while
8:   go through WINDOWBUFFER to look for another reading with the same key as INCOMING
9:   if nothing is found then
10:    output INCOMING
11:  end if
12:  append INCOMING to the end of WINDOWBUFFER
13: end loop
```

Complexity In *baseline_merge*, a linear scan is performed on the full window for each incoming reading, therefore the time cost is Θ(n). The space cost is simply the window itself in a FIFO queue, at Θ(n).

Baseline_merge is intuitive and can be also easily realized in some systems that support the concept of sliding windows. For example, a SQL-based DSMS(Data Stream Management System) can code *baseline_merge* as the following continuous query, assuming a data stream schema of `Readings(key, time)`:

```
SELECT key, time
FROM Readings R1
WHERE NOT EXISTS
    ( SELECT *
        FROM Readings R2
        OVER(max_distance milliseconds PRECEDING R1)
        WHERE R2.key = key
        AND R2.time <> time)
```

Algorithm 5 Hash_merge (param: `max_distance`)

```
1: TABLE ← empty hashtable {hashtable to store the last appearance time for each key}
2: loop {loop forever for next incoming reading}
3:   INCOMING ← the next reading
4:   if INCOMING.timestamp - TABLE[INCOMING.key] > max_distance then
5:     output INCOMING
6:   end if
7:   update TABLE[INCOMING.key] to be INCOMING.timestamp
8: end loop
```

Baseline_merge carries a Θ(n) time cost per incoming reading, and a Θ(n) space cost, both of which can be further improved. In fact, it is straightforward to see that it is not necessary to keep a *max_distance* window worth of readings in order to determine whether an incoming reading is a duplicate. All that needs to be maintained is a timestamp to indicate the last time a reading with the same key as the incoming reading appears. If the distance between the incoming timestamp and the last timestamp is larger than *max_distance*, then we treat it as a new reading and output it.

Algorithm 5 uses a hashtable to keep the last appearance timestamp for each distinct key value. For each incoming reading, its timestamp is compared to the corresponding entry for this key in the hashtable, and the reading is determined to be a new tag reading if the key does not appear in the table, or the time distance is larger than `threshold`.

Complexity. Since the hashtable keeps one entry per distinct key value, the average space cost is now Θ(k), compared to Θ(n) of *base_merge*. Furthermore, the time cost per incoming reading is now reduced to O(1) for hashtable lookup, instead of a full scan of Θ(n). The cost of maintaining the hashtable is less than O(1) on-average for each incoming tuple, since not every incoming/expiring tuple will cause insertion/deletion of keys from the hashtable.

5 Performance Study

For experiments, a random RFID reading generator was created, which generates RFID tag reading according to a Poisson process. The Poisson process generates tag readings with random arrival time, while the arrival time conforms to a Poisson distribution with a chosen average tag arrival rate. Each generated tag reading repeats for 10 times, with some chosen noise level (a certain percentage of the reading are noise).

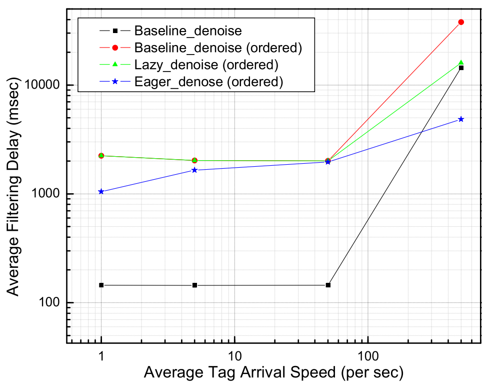

Figure 2: Noise Elimination: Delay under Different Arrival Rates

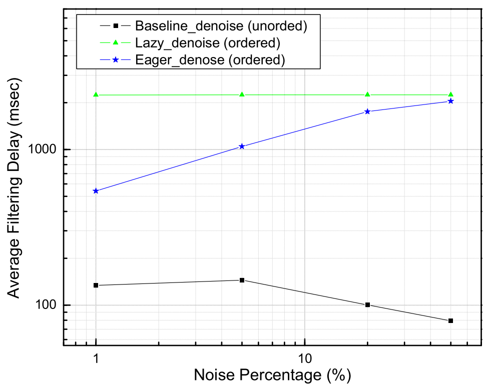

Figure 3: Noise Elimination: Delay under Different Noise Percentage

5.1 Performance of Denoising under Different Arrival Rates

In the first experiment, we study the performance of the various algorithms under different tag rates. The random generator is fixed with the following parameters: each tag reading repeats 10 times, with 200 milliseconds gap between the repeats, and a 5 percent of tag readings are noise. The average tag arrival rates tested include: 1 tag/sec, 5 tags/sec, 50 tags/sec and 500 tags/sec. (With repeats set to 10/tag, the total reading arrival rates are 10/sec, 50/sec, 500/sec and 5000/sec, respectively.) Average filtering delay over all output readings is used to measure the performance of the algorithms.

In Figure 2, four algorithms are used to filter the reading to perform denoising. *Baseline (Unordered)* corresponds to the *Baseline_denoise* algorithm presented above, which performs denoising without any optimization, and output the readings in incorrect timestamp order. *Baseline (Ordered)* is a modified version of the *Baseline_denoise* algorithm, which also performs denoising without any optimization, but outputs the readings in correct orders by outputting at the time of expiring from sliding window. *Lazy_denoise* and *Eager_denoise* are exactly as described above, and both output readings in correct time order.

All four algorithms function correctly to filter out the noise readings, and the three ordered-output algorithms also proved to maintain the correct ordering. Figure 2 shows the performance of the algorithms in terms of average delay of readings. *Baseline (Unordered)* works well with low tag rates, because it completely ignores the output time order issue and therefore has the advantage of output immediately on detection. Its performance degrades under high tag rate situations due to large overhead of linear scanning of the large sliding window under high rates. *Baseline (Ordered)* has the worst performance of all, since it has no optimization, while it still tries to maintain the timestamp ordering. *Lazy_denoise* performs better than *Baseline (Ordered)* under high loads because it utilizes hashtables to reduce the overhead. *Eager_denoise* has the best performance of all, since it not only utilizes the hashtable optimization, but also outputs readings as soon as they are safe to output. Overall, *Eager_denoise* has the best performance under all load conditions.

5.2 Performance of Denoising under Different Noise Ratio

The *Baseline (Unordered)*, *Lazy_denoise* and *Eager_denoise* algorithms are studied for the performance under different noise ratio. The random generator is fixed with the following parameters: each tag reading repeat 10 times, with 200 milliseconds gap between the repeats, and overall tag arrival rate is 1/second. Then different noise ratios are tested, including 1%, 5%, 20% and 50%.

Again, from Figure 3, *Baseline (Unordered)* works well in terms of performance since it ignores the ordering issue and outputs immediately upon detection, but its output readings are in incorrect time order. *Lazy_denoise* has to wait until the readings get expired from the sliding window, therefore it has the largest delay. The interesting observation is that, under low noise ratio, *Eager_denoise* works almost as well as *Baseline (Unordered)*, although it maintains the correct output time order. That is because when noise ratio is low, it is more likely for a non-noise reading to be output early under *Eager_denoise*, when there is no more noise preceding it in the sliding window. However, as noise ratio gets higher, *Eager_denoise* gets closer to *Lazy_denoise* since there are more and more noise readings present to prevent early outputting. Nonetheless, overall *Eager_denoise* always works better than *Lazy_denoise*.

5.3 Performance of Duplicate Elimination

We study the performance of the two duplicate elimination algorithms (*Baseline_merge* and *Hash_merge*) under different tag arrival rates. The random generator is fixed with the following parameters: each

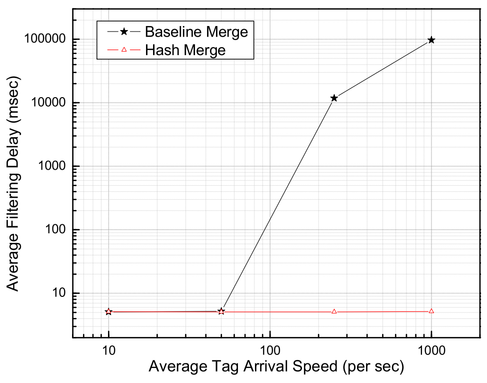

Figure 4: Duplicate Elimination: Delay under Different Arrival Rates

tag reading repeat 10 times, with 200 milliseconds gap between the repeats, and a 0 percent noise (since here we are testing duplication elimination only, noise are presumed already removed by previous filtering). Then the performance is tested under different average tag arrival rates, including 10 tags/sec, 50 tags/sec, 250 tags/sec and 1000 tags/sec. (With repeats set to 10/tag, the total reading arrival rates are 100/sec, 500/sec, 2500/sec and 10000/sec, respectively.)

Both algorithms are able to eliminate duplicate readings and only output the corresponding reading once. However, it is clear from Figure 4 that *Hash_merge* is far-superior than the baseline implementation. The delay is basically negligible even under an arrival rate of 10,000 readings/sec (1000 tags/sec) for *Hash_merge*, while *Baseline_merge* starts to cause large delays after tag rate reaches 500 readings/sec(50 tags/sec).

6 Related Work

RFID data filtering needs to remove noise and duplicate from continuous high volume RFID data streams generated from RFID readers. Such filtering is essential to provide accurate data used for RFID-enabled pervasive applications. While RFID data filtering is supported in RFID Middleware systems such as [7, 8, 9], large volume real-time RFID data streams demand more efficient approaches for filtering these data.

RFID data processing is a hybrid of event processing and stream processing. Past work on event detection and processing – such as [10, 11] – is not concerned with processing speed and memory management issues, where events are normally generated from databases and different from events from high-speed event streams. On the other hand, past work on data stream processing and continuous query optimization [12, 13, 14] assumes accurate stream sources and is not concerned with RFID application-specific issues, such as the existence of noisy and duplicate readings.

In [15], a probability-based approach is provided to detect duplicate in web click streams. This approach can not be applied to RFID data, since accuracy is among the top priority for RFID data processing.

7 Conclusion

In this paper, we identify the problem of RFID data filtering and develop efficient methods to eliminate noise and duplicate from RFID observations. Specially, for noise filtering (denoising or smoothing), we propose an approach for more efficiently maintaining the original time order of observations in the output; and for duplicate elimination, the approach that we formulate can minimize memory requirement for history buffering. We then perform experiments to validate our approaches through simulated RFID data generator and demonstrate that our approaches are effective and efficient. Our approach of data filtering is essential to provide clean and correct RFID data before they can be further processed, transformed, and integrated for RFID-enabled pervasive applications. The techniques also provide an important reference for building RFID Middleware [7, 8, 9] where filtering is a critical component.

References

[1] M. Lampe and C. Flrkemeier. The Smart Box Application Model. In *PerCom*, 2004.

[2] Siemens to Pilot RFID Bracelets for Health Care. http://www.infoworld.com/article/04/07/23/HNrfid implants_1.html, July 2004.

[3] EPC Tag Data Standards Version 1.1. Technical report, EPCGlobal Inc, April 2004.

[4] J. Brusey et. al. Reasoning About Uncertainty in Location Identification with RFID. In *RUR at IJCAI*, August 2003.

[5] H. Vogt. Efficient Object Identification with Passive RFID Tags. In *Pervasive*, 2002.

[6] F. Wang and P. Liu. Temporal Management of RFID Data. In *VLDB*, 2005.

[7] C. Bornhoevd et. al. Integrating Automatic Data Acquisition with Business Processes - Experiences with SAP's Auto-ID Infrastructure. In *VLDB*, 2004.

[8] Oracle Sensor Edge Server. http://www.oracle.com /technology/products/iaswe/edge_server.

[9] Sybase RFID Solutions. http://www.sybase.com/rfid, 2005.

[10] S. Chakravarthy et. al. Composite Events for Active Databases: Semantics, Contexts and Detection. In *VLDB*, 1994.

[11] N. H. Gehani et. al. Composite Event Specification in Active Databases: Model & Implementation. In *VLDB*, 1992.

[12] R. Motwani et. al. Query processing, approximation, and resource management in a data stream management system. In *CIDR*, 2003.

[13] Sam Madden et. al. Continuously adaptive continuous queries over streams. In *SIGMOD*, 2002.

[14] D. Abadi et. al. Aurora: A new model and architecture for data stream management. *VLDB Journal*, 12(2), 2003.

[15] A.Metwally et. al. Duplicate Detection in Click Streams. In *WWW*, 2005.