Estimating Missing Values in Related Sensor Data Streams

Mihail Halatchev Le Gruenwald
The University of Oklahoma
School of Computer Science
Norman, Oklahoma 73019, U.S.A.
1-(405)-325-3498
{mhalatch@hotmail.com, ggruenwald@ou.edu}

Abstract

In wireless sensor networks, a significant amount of sensor readings sent from the sensors to the data processing point(s) may be lost or corrupted. In this research we propose a power-aware technique, called WARM (Window Association Rule Mining), to deal with such a problem. In WARM, to save battery power on sensors, instead of requesting the sensor nodes (MS), the readings of which are missing, to resend their last readings, an estimation of the missing value(s) is performed by using the values available at the sensors relating to the MS through association rule mining. The paper then presents the performance studies comparing WARM with existing techniques using the real traffic data collected by the Department of Transportation in Austin, Texas.

1. Introduction

Recent advances in Micro Electro Mechanical Systems based sensor technology, low-power analog and digital electronics, and low-power Radio Frequency (RF) design have made possible the development of relatively inexpensive and low-power wireless microsensors that can be integrated in a network [7, 9]. The purpose of such a network is to monitor, combine, analyze and probably respond to the data collected by hundreds (or even thousands) sensors distributed in the physical world in a timely manner. The possible applications are widespread – from battlefield (monitoring the movement of the enemy troops) to factory floor (monitoring motors, small robotic devices, etc.) to collecting data from an inaccessible terrain.

The continuous flow of data readings from a sensor farther into the network is called data stream. Under this model of communication all sensors push their readings to the other sensors (or sensors' proxies [12]) immediately after the data is collected or an event is detected and a packet is generated. Data streams have several important properties – the tuples exist online, i.e. they are not permanently stored; the arrival rate is not strict, i.e., the presence of the needed data for a specific query cannot be assumed; the streams are potentially unbounded in size, i.e., after a certain time, or rather after the storage limits are reached, some data must be discarded; and the tuples may get lost or corrupted.

In a wireless sensor network, it can be expected that the sensor readings sent from the sensor farther into the network (to another sensor or to a sensor's proxy) may be lost, corrupted or late. The reasons include power outage at the sensor node, sensors timers synchronization, random occurrences of local interferences (such as mobile radio devices, microwaves or broken line-of-sight path), a higher bit error rate of the wireless radio transmissions compared to the wire communication alternative, a poor performance of the implemented routing algorithm in certain situations. In an effort to provide a high quality of service for a wireless sensor network, a technique to deal with such undesirable events should be derived.

The objective of this research is to derive a technique for dealing with the case of missing, corrupted, or late reading from a particular sensor (i.e. missing tuple in a data stream) in the presence of other data streams that are possibly related to the stream with the missing tuple.

A solution that provides a good quality of service (QoS) of a monitoring application is important. Since the purpose of a sensor network is to monitor a real time phenomenon, its QoS can be defined as a function of the time needed by the queries to produce results based on the data gathered by the sensors and the accuracy of these results. There exist some applications for which the response deadlines for the queries are tight, and the accuracy of the query results is important. One such

This material is based upon work supported by the National Aeronautics and Space Administration under Grant No. NNG05GA30G issued through the Office of Space Science.

ADVANCES IN DATA MANAGEMENT 2005
Jayant Haritsa, T.M. Vijayaraman (Editors)
© CSI 2005

example is a sensor network monitoring the moving of enemy planes or missiles. In order to intercept and probably destroy them, much smaller latency and error in determining their position, direction, etc. may be allowed. For such applications, the time allowed for waiting all the sensor readings to arrive before executing a query will be much smaller. As a result a technique for estimating the late sensor readings, which are considered missing after a predefined deadline expires, is needed. Such a technique should provide an acceptable QoS for the application.

When a tuple is missing, there is no way for a continuous query to know this information. An action that may be taken is that after waiting for some predefined interval of time, the Data Stream Management System (DSMS) sends a request to the sensor with the missing tuple asking it to resend the data again. However, this approach has two major drawbacks: increased power consumptions by the sensors (they should listen for requests and resend data if needed) and increased latency of the produced result by the query (time spent for transmitting a request and waiting for a response). In addition, there is no guarantee that the requested reading will be provided because the sensor with the missing tuple may be out of power. Even if a tuple arrives well before the end of the predefined interval of time for waiting, it may be corrupted. Asking the sensor to resend the tuple may not have a significant impact on the result generation deadlines, but still poses the problem of increased power consumption.

These problems motivate us to develop an alternative approach. Instead of sending a request and waiting for a response, thus spending additional time and power, we provide a technique for estimating the missing or corrupted data. Our goal is to provide good QoS for applications running in sensor networks, where QoS is a function of the correctness of the estimated data and the achieved response time for producing the estimation, while paying close attention to the power consumption by the sensors.

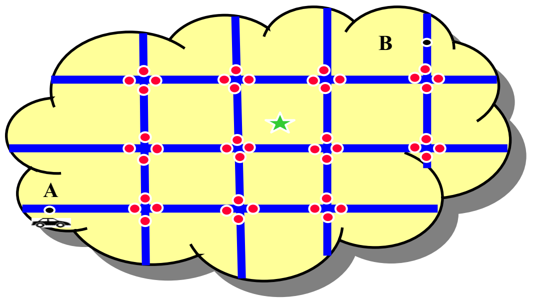
The rest of this paper is organized as follows. Section 2 provides an overview of the related work. Section 3 presents the proposed Data Stream Association Rule Mining (DSARM) framework. Section 4 contains the implementation details of the proposed data model for storing the sensor data and the proposed algorithms for updating the data model and for estimating a missing value. Section 5 presents the performance evaluations of the proposed approach by means of simulation. Finally, the conclusions and future research are provided in Section 6.

2. Related Work

In the presence of data streams the traditional Data Base Management Systems (DBMSs) show poor performance. The need for a new type of Data Stream Management Systems (DSMSs) is well justified in [6]. The issues that have to be addressed in such a DSMS include the architecture of the DSMS, the type of queries executed on streaming data, the query languages, the query operators, the query processing algorithms and the query processing optimizations [3, 6, 12]. However, there is little research done on estimating missing values in a data stream. The general problem of estimating missing values is well studied in the statistics field [13, 14], but the derived techniques lack software implementation and are rarely used in practice [15].

A possible mathematical approach to deal with missing values is proposed in [16]. The SVDimpute algorithm uses singular value decomposition for estimating a missing data in DNA microarrays. The time complexity of this algorithm is $O(n^2mi)$, where n is the number of columns (experiments), m is the number of rows (genes), and i is the number of iterations. Another technique, called *KNNimpute*, utilizes the clustering approach for estimating missing values in DNA microarrays [16]. The time complexity of the KNNimpute is $O(m^2nv)$, where m is the number of rows (genes), n is the number of columns (experiments), and v is the number of missing values in a DNA microarray. While in DNA microarray analysis there are no strict time constraints, in the case of data stream environments for monitoring or tracking applications, the response time can be of much importance. It can be expected that because of their time complexity the SVDimpute and KNNimpute algorithms may not be able to meet the time deadlines typical to some data stream applications.

3. The DSARM Framework



● sensor nodes ★ server (sensors' proxy)

Figure 1. Sample Network

To illustrate the following discussion we have assumed a sample network (similar to [12]) as shown in Figure 1. It consists of a set of sensor nodes and a server (sensors' proxy). The sensor nodes are installed in the road system of a city. Each sensor node is equipped with a motion sensor, radio transmitter, processor, memory, and battery. The motion sensors are constantly turned on and detect passing vehicles over particular locations. The number of passing vehicles is accumulated for a predefined time interval. This number is temporarily stored in the memory. At a predefined time interval, a tuple consisting of the sensor id, time interval, and the number of vehicles detected for this time interval is generated. The processor wakes the radio transmitter up, and the radio transmitter sends the generated tuple to the server, assuming the server is within the radius of influence of each sensor node in the network (i.e., single-hop communication). Immediately after the transmission is completed, the radio unit is turned off. At the server, the data reported by the sensor nodes is stored, the estimation of a missing reading is performed when needed, and the application-specific continuous queries over the sensor data are run. One example of such a continuous query is "find the fastest route from point A to point B with respect to the current traffic conditions."

First, let us explain why we think it is reasonable to expect that there may exist relationships between the sensors integrated in a sensor network monitoring a real world phenomenon. Considering our sample network, several basic rules can be stated. For example, the sum of all outflow traffic of an intersection is equal to the sum of the inflow traffic to this intersection. Or it can be noted that the outflow traffic from an intersection is the inflow traffic for another intersection. Intuitively, it can be expected that the impact of the rush-hour traffic may be almost the same for given sets of city streets – some highways will experience significant traffic increase, while for some residential streets, the rush-hour traffic will have little or no impact.

To estimate the values of the missing tuples, our research first uses association rule data mining to identify the sensors that are related to the sensors with the missing tuples. Our research then uses the current readings of the related sensors to calculate the missing values in the current round.

The definition of the association rule problem and the association rule mining algorithm, Apriori, which we modify for our work, are presented in [2]. Here they are repeated for convenience. Let $I = \{i_1, i_2, \ldots\, i_n\}$ be a set of items. Given a set of sales transactions D, where each transaction T is a subset of I, find all association rules of the form $X \rightarrow Y$, where X (the antecedent) and Y (the consequent) are subsets of I and $X \cap Y = \varnothing$. An association rule $X \rightarrow Y$ is said to hold in the transaction set D with an *actual confidence* (*actConf*) if *actConf* percent of the transactions that contain X also contain Y, and an *actual support* (*actSup*) if *actSup* percent of the transactions in D contain both X and Y. The task of mining association rules is to find all the association rules which satisfy both the user-defined *minimum support* (*minSup*) and *minimum confidence* (*minConf*).

For Apriori, in order to find all association rules, first all *frequent itemsets* have to be determined. A set containing *k* items (called *k-itemset*) is a *k-frequent itemset* if at least *minSup* percent of the transactions in D contain the k-itemset. The sequence of finding all frequent itemsets is the following: first find all 1-frequent itemsets, next use the discovered 1-frequent itemsets to find all 2-frequent itemsets, and keep generating higher order k-frequent itemsets by using the set of the already discovered k-1-frequent itemsets until no new k+1-frequent itemset can be generated.

A direct application of the Apriori algorithm to the problem of estimating missing data in a data stream environment may not be possible. The Apriori algorithm assumes that the base data that needs to be mined to generate new knowledge in form of association rules is static while it is running. This means that the data must be stored completely before Apriori can be performed. When new data arrives, it should be added to already stored data and the updated association rules should be generated again from scratch. This approach makes two assumptions: the storage space for data is enough to store all the data to be mined, and the response time for generating all association rules, although important, is not crucial. In the case of data streams, in which sensors send data frequently, both assumptions may not be applicable. First, all of the received data from the data streams, on which a mining technique is to be performed to extract some new knowledge, cannot be stored at the server for an infinite period of time – the amount of data is potentially unbounded. As a result we must purge the older data to free some space for the newer data. Second, the sensor networks are developed to monitor real world events, and the response time is very important. If we use the Apriori algorithm for discovering association rules, every time a missing data is detected in a given round, we should start generating all frequent itemsets and all association rules from scratch, which may be prohibitively expensive in terms of time.

Another reason that Apriori cannot be applied directly to data stream mining is the format of the data to be mined. Apriori was originally derived for mining basket data which is Boolean by nature. In the case of data streams of sensor readings the data to be mined is of quantitative nature.

To adapt the basket association rule mining technique to the data stream environment we propose the Data Stream Association Rule Mining (DSARM) framework. The proposed modifications are discussed below.

Instead of generating all association rules between sensors, generate all the association rules between pairs of sensors only.

Translated into the association rules vocabulary, that means that the set will contain rules of type A $\rightarrow$ C and B

→ C, but not rules of type A, B … → C. It can be proved that in order to generate all possible association rules between pairs of sensors, it is necessary and sufficient to generate only the sets of all 1- and 2-frequent itemsets. It was observed by many researchers [2, 5] that the bottleneck in mining for association rules is exactly the task of finding the set of 3+ frequent itemsets. *The first advantage of the proposed modification is that by generating only the sets of all 1- and 2-frequent itemsets, the time needed for extraction of all applicable association rules, as well as the overall time for estimating the missing value, will be significantly reduced.*

As a result of our proposed modification the rules of type A, B → C are never generated. If a rule of type A, B → C exists in reality, the only information that we can gather from the possible generation of the A, B → C rule is that, in some cases, the calculated *actSup* and/or *actConf* for the underlying rules (A → C, B → C) may be over- or under-estimated. *The negative effect of this modification to the Apriori algorithm is that we cannot use the calculated actSup and/or actConf for the underlying rules to reliably determine the weight with which every sensor (A and B) will contribute in estimating the missing value (for sensor C).* For that reason, as an alternative approach to determine the weight with which every sensor (A and B) will contribute in estimating the missing value (for sensor C), we use the distance between the recorded histories of the sensors readings. The distance is defined as the percentage of the exact matches of the reported states in the history of the *{A, MissingSensor}* pair from all records currently stored for this pair.

The fact that the generation of 1- and 2-frequent itemsets is necessary and sufficient for generating all applicable association rules between pairs of sensors has another consequence as well. In the Apriori algorithm, if the number of sensors (or number of items, in the general case) is denoted by *n*, then the maximum number of possibly existing frequent itemsets is calculated by the formula:

$$\text{Max\_Num\_Freq\_Itemsets} = \sum_{i=1}^{n} \binom{n}{i}$$

and in case n = 100, this evaluates to $\approx 10^{30}$.

On the other hand, by using the modified technique (generating only 1- and 2- frequent itemsets) the maximum number of all frequent itemsets is $\leq n^2 + n$. The significant reduction in the maximum number of possibly existing frequent itemsets makes feasible the idea of creating and maintaining the data structures that contain the metadata in the form of counters for all possibly existing frequent itemsets. These counters store the number of observed 1- and 2-itemsets that are currently stored at the server. For using the metadata structures, the cost that has to be paid is the cost of maintaining them every time a new round of readings arrives, and the additional memory space. But having up-to-date data in form of counts for all 1- and 2-itemsets will significantly reduce the time for generating the association rules because the number of needed memory access operations will be reduced substantially (for each possible frequent itemset we need only one memory read to calculate its *actSup*, instead of performing multiple memory reads in order to count it from scratch). *The second advantage of the proposed modification is that the use of the data structures containing the metadata about all possibly existing 1- and 2-frequent itemsets is now feasible, and this will lead to an additional decrease of the time needed for generating all applicable association rules and of the overall time for estimating the missing value.* As a result of the above stated facts, our next proposed modification to the Apriori algorithm can be stated as follows.

Use additional data structures that contain the metadata (in form of counters) about all possibly existing 1- and 2-frequent itemsets to reduce the number of memory access operations.

To address the quantitative nature of sensor readings as opposed to the Boolean nature of the data in the Apriori algorithm, assume the following data for a pair of sensors to be stored at the server as shown in Table 1.

Table 1. Sample transactions for a pair of sensors

Round Number	Sensor 1	Sensor 2
1	Light	Light
2	Moderate	Heavy
3	Heavy	Light
4	Moderate	Moderate
5	Light	Light

The Apriori algorithm can be modified to operate on quantitative data instead of Boolean data. For example, Apriori can be modified to check all the states reported by any two sensors stored in the data set if they are the same (round by round), and if it is true for at least the *minSup* percentage of the cases stored in the data set, then this pair of sensors is a 2-frequent itemset. In the above example there are 2 rounds in which the sensors reported the same state (*actSup* = 40%), and if *minSup* = 30%, then Sensor 1 and Sensor 2 form a 2-frequent itemset. The *actConf* of a rule Sensor 1 → Sensor 2 is 100%. This result can be misleading. If we state that there exists a rule Sensor 1 → Sensor 2 with *actSup* = 40%, *actConf* = 100%, that means it is true regardless of the possible state of the sensors. As can be seen from the sample data shown in Table 3. this is not correct. The rule Sensor 1 → Sensor 2 holds with *actSup* = 40%, *actConf* = 100% only with respect to the “Light” state of traffic. For this reason, our next proposed modification to the Apriori algorithm can be stated as follows.

Evaluate frequent itemsets and association rules between pairs of sensors always with respect to a particular state of sensors.

To address the issue of mapping potentially unbounded size of the data generated by sensors to a finite storage space, we propose the use of the sliding window concept [10]. Implementing this concept, only the last *w* rounds of sensor readings will be stored. After receiving the first *w* rounds, every time a new round arrives at the server, the data that form the oldest round should be discarded. Our next proposed modification to the Apriori algorithm can be stated as follows.

Use the sliding window concept in the data structures that store the data arriving from the sensors and in the additional data structures that store the metadata about the 1- and 2-frequent itemsets.

In summary, the proposed DSARM approach can be stated in the following way. Let $I = \{i_1, i_2, \dots i_n\}$ be a set of sensors. Let the size of the sliding window be *w* rounds. Given is the set *D* of last *w* rounds of reported sensor states, where each round *T* consists of reported states for the sensors in *I*. Find all association rules of the form $X \rightarrow Y / s$, (pronounced *X* determines *Y* w.r.t. *s*) where *s* is a sensor state out of all possible sensor states, and *X* and *Y* are subsets of *I* of size one (i.e. each *X* and *Y* is an item of *I*) and $X \cap Y = \emptyset$.

An association rule $X \rightarrow Y \mid s$ is said to hold in the set of the currently stored rounds *D* with the *actual confidence actConf* if *actConf* percent of the rounds that report *s* for *X* also report *s* for *Y*, and with the *actual support actSup* if *actSup* percent of the currently stored rounds in *D* report *s* for both *X* and *Y*.

The task of mining association rules then is to find all the association rules between pairs of sensors w.r.t. all possible sensor states which satisfy both the user-defined *minimum support minSup* and *minimum confidence minConf*.

4. The Proposed Data Model

The proposed data model for storing the rounds of sensor readings at the server consists of three major data structures – the Buffer, the Cube, and the Counter. To illustrate the use of the proposed data structures and the operation of the algorithms for updating the data model when a new round of sensor readings arrives and for estimating a missing value, a very basic instance of the sample network presented in Section 3 is taken as an example. The network consists of five sensor nodes (S0, S1, S2, S3, and S4) that send their readings to a server, where the proposed data model and corresponding algorithms are implemented. Each sensor detects the passing vehicles for a certain period of time (synchronized for all sensors) and accumulates the number of those vehicles in its memory. At the end of the time period the sensor node evaluates the collected data and generates a tuple consisting of the sensor id, the time period, and the observed state of the traffic for this time period. The different states of the observed traffic are represented as follows: Light = 1, Moderate = 2, Heavy = 3, Congestion = 4. Next, the sensor node sends the generated tuple to the server using its radio unit.

Assume the sensors have reported the following states as shown in Table 2 for the last 5 rounds of sensor readings, where round number 1 is the oldest round and round number 5 is the newest one. Assume *minSup* = 25% and *minConf* = 25%.

Table 2. Sample transactions from 5 sensors

Round Number	S0	S1	S2	S3	S4
1	1	1	1	2	2
2	2	1	1	2	3
3	4	2	2	2	2
4	1	1	1	1	4
5	1	2	2	2	3

4.1. The Buffer

The purpose of this data structure is to store the arriving readings associated with the corresponding sensors. The Buffer can be implemented as an array of size *n*, where *n* is the number of sensors. A value of ‘-1’ in the Buffer means that there is still no data received for this sensor. An example of the state of the Buffer after receiving round 5 is shown in Figure 2.

S0	S1	S2	S3	S4
1	2	2	2	3

Figure 2. The Buffer

4.2. The Cube

The purpose of the Cube is to keep track of all existing 1- and 2-itemsets in each round, which are stored in the corresponding nodes and slices. When any two sensors report the same state in a given round, then they form a 2-itemset w.r.t. the reported state for this round. Every sensor reporting a particular state forms a 1-itemset w.r.t. this state.

By its nature the Cube is a data cube, implementing the sliding window concept by storing data for the last *w* rounds of readings. The size of the sliding window (i.e. the depth of the Cube) is a dynamic parameter in our simulation experiments. The newest data is stored at the front of the Cube, and the oldest data is stored at the back of the Cube. For the illustrating example, the size of the sliding window is assumed to be 5 as shown in Figure 3.

We refer to the collection of all nodes at the front of the Cube as slice[0], in the next slice as slice[1], and in the slice at the back of the Cube as slice[4]. To refer to each single node for sensors S_i and S_j in a particular slice k of the Cube we use the format Cube[S_i][S_j].slice[k].

The data that we store in each node of the Cube is generated in the following way. Cube[S_i][S_j].slice[*0*] is set to be equal to the state reported by S_i if this state is the same as the state reported by S_j (2-itemset) or if $i == j$ (1-itemset); otherwise, Cube[S_i][S_j].slice[*0*] is set to -1

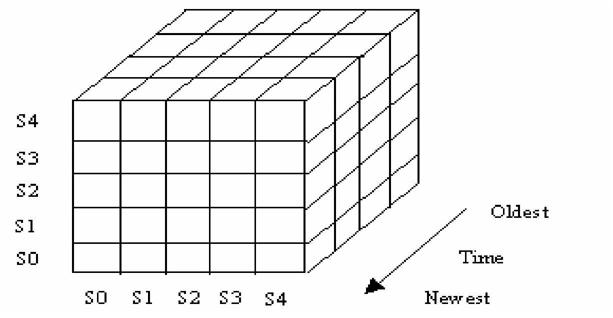


Figure 3. The Cube

4.3. The Counter

The purpose of this additional data structure is to speed up the estimation of a missing value. When estimating the missing reading from a given sensor (*MS*), first we have to check if the sensor is a 1-frequent itemset for any of the possible states. This is done by comparing the *actSup* of the sensor for all of the possible states with the user-defined *minSup*. Next, for all states (called *eligible states*) in which the missing sensor is a 1-frequent itemset, we have to check if *MS* constitutes a 2-frequent itemset with other sensors (S_i) that have no missing value in the current round. This is done by comparing the *actSup* of a {S_i, *MS*} 2-itemset w.r.t. every eligible state with the user-defined *minSup*. Last, for all {S_i, *MS*} 2-frequent itemsets (2-itemsets that have *actSup* >= *minSup*), compare the *actConf* of the rule S_i → *MS* with the user-defined *minConf* (for detailed description of the estimate algorithm, see Section 4.4.3).

To speed up the estimation, instead of going through all data stored in the Cube and performing counting for each of the 1- and 2-itemsets, we keep counters for all possible 1- and 2-itemsets. Each of these counters stores an integer representing the number of observed 1- and 2-itemsets w.r.t. a particular state that are currently stored at the server. The counters are updated every time a new round is stored in the Cube. The Counter data structure is a collection of all the counters. In this way, to check the *actSup* for an itemset in question, only one read, instead of multiple reads, has to be performed. For checking the *actConf* of a possible association rule, only two reads, instead of multiple reads, have to be performed.

The Counter can be implemented as a 3D array. Its size is equal to n x n x p, where n is the number of sensors and p is the number of possible states. Figure 4 shows the Counter for the illustrating example where p = 4 because there are four possible states, '1', '2', '3', and '4'.

We refer to the collection of all nodes at the front of the Counter as state[1], in the next state as state[2], and in the state at the back of the Counter as state[4]. We refer to each node for sensors S_i and S_j in a particular state k of the Counter as Counter[S_i][S_j][k].

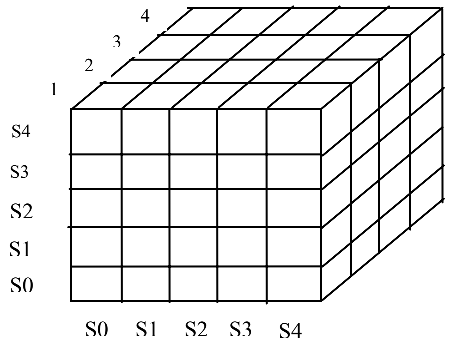


Figure 4. The Counter

4.4. The Proposed Algorithms

Three algorithms are developed to work with the proposed data model – *checkBuffer()*, *update()*, and *estimateValue()*. These algorithms are subject to two response time constraints. The first time constraint, which can be thought of as a hard deadline, is that the consequent execution of *checkBuffer()*,*estimateValue()*, and *update()* for a given round should be completed before the data from the next round start to arrive. The second time constraint, which can be thought of as a soft deadline, applies to the consequent execution of the *checkBuffer()* and *estimateValue()* algorithms. If there are missing values in the current round, their estimation should be completed as soon as possible. The overall goal of the proposed data model and algorithms is to provide an acceptable quality of service (QoS), i.e., to generate a relatively good estimation of the missing values relatively fast. On one hand, if we use some average approaches for estimating the missing values (discussed in Section 5) we can obtain the estimated values extremely fast, but the accuracy of the estimation may be poor, thus resulting in poor QoS. On the other hand, if an alternative approach for estimating the missing values such as the one we are proposing is used, even if it provides excellent accuracy of the estimated values, if it is slow in producing the estimations, the achieved QoS also will be poor. Indeed, if the estimation of the missing values completes shortly before the next round of data starts to arrive, no matter how accurate the estimation is, the useful life of the results produced by the continuous queries operating on the sensor data will be extremely short.

4.4.1. The checkBuffer() Algorithm

The *checkBuffer()* algorithm is shown in Figure 5.

```
1. checkBuffer()    // This algorithm checks the Buffer at a
// predefined time interval for a presence of missing sensors
// readings in the current round. If missing values were
// found, it invokes the estimateValue()
// algorithm, else it invokes the update() algorithm
2.    boolean missingValue;
3.    int missingSensorID;
4.  while(true) // repeat this process indefinitely
5.  while(time window for current session with the
    sensors is still open)
      listen to sensors and record the data received from
      a particular sensor (S0,S1,S2,S3,S4)
      to corresponding field in the Buffer;
6.  end inner while;
 // check if there is a missing value after the current session
 // is closed
7.   for(int i = 0; i < numberOfSensors; i++)
8.   if(Buffer[i] == -1)
9.     missingValue = true;
10.     missingSensorID = i;
11.  end if;
12. end for;
13. if(missingValue) // there is a missing value
14.    invoke estimateValue(missingSensorID);
15.  else
16.    send OK signal to queries;
17.    invoke update();
18.  end if;
19.  // upon completion of update() algorithm
20.  set missingValue = false;
21. set values in the Buffer to -1 // meaning "no new data
                                   // stored"
22.  end outer while;
```

Figure 5. The *checkBuffer()* Algorithm

4.4.2. The update() Algorithm

The purpose of the *update()* algorithm is to update both the Cube and the Counter when a reading without missing values is received in the Buffer, or in case of missing values present in the current round, after their estimation has been completed. The data that we store in each node when updating the Cube is generated as discussed in Section 4.2. The updating of the Counter is done in the following way. For every 1- and 2-itemset discovered in the sensor readings currently stored in the Buffer, the value in corresponding node in the Counter is incremented by one because a new 1- or 2-itemset is now stored in the Cube. On the other hand, after the Cube is initially filled, every time the data from a new round is stored at the front of the Cube, the data for the oldest round is discarded from the back of the Cube. When discarding the oldest round, the content of each discarded node is examined. If the value stored at the node being discarded is different than -1, i.e., to be discarded is an existing 1- or 2-itemset from the Cube, then the count for this 1- or 2-itemset, which is stored in the corresponding node in the Counter, should be decremented by one. The *update()* algorithm is shown in Figure 6 and Figure 7.

```
1. update()// The purpose of this algorithm is to update the Cube
and the Counter every time a new round  (without missing
values) of sensor readings is stored in the Buffer.
// Start a loop that traverses sensor readings in the Buffer
2.for (int j = 0; j < bufferSize; j++)
// first update 1-itemsets
3.   if(Buffer[j] == 'x')  // x represents a possible sensor state
4.      Cube[$S_j$][$S_j$].refresh('1'); // the refresh(state) method
 // is presented in Figure 7.
5.   end if;
// Start another loop to generate 2-itemsets between the
// sensor readings in the particular round,
// also traverses the Buffer.
6.for (int k = j+1; k < bufferSize; k++)
// If both sensors report the same event (denoted in our example
// by ('1' || '2' || '3' || '4'))
// set Cube[$S_j$][$S_k$].slice[0] = '1' || '2' || '3' || '4'
7.   if(Buffer[j] == Buffer[k] == 'x')
8.      Cube[$S_j$][$S_k$].refresh ('x');
9.    Cube[$S_j$][$S_k$].refresh ('x');
10.  else //these two sensors do not constitute 2-itemset
11.    Cube[$S_j$][$S_k$].refresh('-1');          //' -1'  meaning no
//  relation between this pair of sensor w.r.t. a
// particular state is detected for the current round
12.    Cube[$S_j$][$S_k$].refresh('-1');
13.    end if;
14.  end inner for;
15.end outer for;
```

Figure 6. The update() Algorithm

```
1. refresh(possibleState)// This method is invoked by the
update() algorithm. The purpose of the refresh(state) method
// is to add new nodes at the front of the Cube every time an
// update of the data model is performed and to discard the
// oldest nodes (at the back of the Cube). The refresh(state)
// method also maintains the counts of the observed 1- and 2-
// itemsets currently stored in the Cube by updating the Counter.
2.Insert a new node at the front of the Cube with the data field
value set to possibleState;
3.if (the value stored in the inserted new node is not == -1)
// i.e. the inserted node contains information about a 1- or 2-
// itemset that was detected in the current round of readings
// stored in the Buffer
      increment the value stored in the corresponding node in
        the Counter by 1;
   end if;
// implement the sliding window concept
4.discard the node at the back of the Cube;
5.if (the value stored in the discarded node is not == -1)
// i.e. the discarded node contains information about a 1- or 2-
itemset which no longer will be stored in the Cube
      decrement the value stored in the corresponding node in
      the Counter by 1;
6. end if;
```

Figure 7. The *refresh(state)* Method

4.4.3.The estimateValue(missingSensorID) Algorithm

The purpose of the *estimateValue(missingSensorID)* algorithm is to estimate a missing sensor reading given its sensor ID by accessing both the Counter and the Cube. The algorithm is shown in Figures 8 (a-d).

The *estimateValue(missingSensorID)* algorithm proceeds in the following manner. First, the *actSup* for *MS* for every possible state is obtained from the Counter and is compared with the *minSup* (Step 3). If for a given state the *actSup* is equal to or greater than the *minSup*, then this state is declared to be an *eligible state*. Only for eligible states, *MS* can be a consequent of an association rule between a pair of sensors. For every discovered eligible state a temporary data structure is generated, called *StateSet* (Step 4). Every *StateSet* is associated with a particular *eligible state*. Next, in every *StateSet*, by examining the content of the Buffer, distributed are the sensor IDs of the sensors that report the same state in the current round as the eligible state, with which the particular *StateSet* is associated (Steps 7 to 11). As a next step, the algorithm checks every sensor S_i in a particular *StateSet* (for every *StateSet*) if it can produce an association rule of the type $S_i \rightarrow MS$ w.r.t. an *eligible state*. This is done in two steps. First, the algorithm checks if the 2-itemset $\{S_i , MS\}$ is a 2-frequent itemset w.r.t. an eligible state (Steps 14 to 16). This is done by obtaining the *actSup* for this 2-itemset (by using the count for this 2-itemset stored in the Counter) and comparing it with the *minSup*. If this 2-itemset is not a 2-frequent itemset (i.e. *actSup* < *minSup*), S_i is deleted from the particular *StateSet*. Else, the *actConf* of the possible rule $S_i \rightarrow MS$ w.r.t. an *eligible state* is obtained (using the data stored in the Counter) and compared with the *minConf* (Steps 17 to 19). If *actConf* is equal to or greater than the *minConf*, this sensor is declared to be an *eligible sensor*, and will participate in the estimation of the missing value for the MS. Otherwise, this sensor is discarded from the *StateSet*. At this point all association rules between pairs of sensors (where the consequent is the *MS*) w.r.t. all *eligible states* are generated. The *eligible states* for the *MS* are known, and all the antecedents (i.e. the *eligible sensors*) are collected in the corresponding *StateSets*. It should be noted that this was achieved without accessing the Cube, thus reducing the number of memory accesses significantly. To determine the weighted contribution of each eligible sensor towards the missing value being estimated, the distance between the histories (reported states stored in the Cube) of this *eligible sensor* and the *MS* is calculated (Steps 26 to 34). The weighted contribution of each *eligible sensor* towards a given state is accumulated in a variable associated with this state (Step 35). The estimated missing value is calculated as shown in Step 39. The estimated value is stored in the Buffer (Step 40), and the Buffer is checked for other missing values (Steps 41 to 46). If other missing values are found in the Buffer, the *estimateValue()* algorithm is called again; otherwise an OK signal is sent to the application queries, and the *update()* algorithm is invoked.

```
1. estimateValue(missingSensorID)
// This algorithm is invoked by the checkBuffer() algorithm
// when a missing value is detected. The purpose of this
// algorithm is to estimate the missing value(s), to store it
// in the Buffer, and when the estimation is completed, to call
// the update() algorithm.
// Determine all eligible states for MS and create StateSets
// for them
2. for(all possible states)
//check if the actSup for this state is >= minSup
3.  if((Counter[MS][MS][state] / slid_win_size) >= minSup)
4.          create a StateSet [e] = ∅;
            //e is the name of the eligible state, in our
            //example, e can be = 1, 2, 3, or 4
5.  end if;
6.  end for;
// Distribute the sensors without missing values (S_i) to the
// corresponding StateSets based on their current reading
// in the Buffer
7. for(i = 0; i < bufferSize; i++)
//get the reading for Buffer[i]
8. .if(reported reading for this sensor == any existing e)
9.          add the sensor id to StateSet [e]
10.end if;
11.end for;
```

Figure 8.a The estimateValue() Algorithm: Determining the eligible states for the MS and distributing the sensors to corresponding StateSets

```
// Clean the StateSets from sensors that do not constitute
// association rules of type S_i → MS | e, for any e
12.for(every StateSet)
13.  for(every sensor S_i in this set)
// Check if S_i constitutes a 2-frequent itemset with MS by
//checking if actSup of {S_i , MS}| e is >= minSup. If not, a rule
//S_i → MS | e, for any e, cannot be generated, then drop this
//sensor from the StateSet
14.    if((Counter[S_i][MS][state] / slid_win_size) < minSup)
15.        discard S_i from the StateSet;
16.    end if;
// Check if S_i can produce a rule of type S_i → MS | e by
// checking if the actConf of such rule,
// (calculated as actSup of {S_i, MS}| e divided by the
// actSup{S_i}| e ) is greater than or equal to
// minConf. If not, a rule S_i → MS | e, for any e,
// cannot be generated, then drop this sensor from
// the StateSet
17.    if((Counter[S_i][MS][state] / Counter[S_i][ S_i][state]) <
                                                     minConf)
18.        discard S_i from the StateSet;
19.    end if;
20.  end for;
21.end for;
```

Figure 8.b The estimateValue() Algorithm: Determining the eligible sensors for the MS

```
// Calculate the weighted contribution of each eligible sensor,
// with which it will participate in the estimated  value of
// the missing sensor reading, by calculating the distance
// between the histories of Si and MS. Create variables to hold
// the accumulated weighted contribution for each eligible state.
22.eligibleStateValue[e] = 0;// where e is any eligible state
// Create a variable that will hold the total of all weighted
// contributions of the eligible sensors
23.totWeightCont = 0;
24.for(every e)
25.  for(every eligible sensor in StateSet [e])
// Calculate the distance between the histories of Si and MS
26.    distance = 0;
27.    fingerMS = Cube[MS][MS].slice[0];
28.    fingerSi = Cube[Si][Si].slice[0];
29.    while(fingerMS != null) //traverse the DLLs
// The distance is measured as the number of exact matches of
// the sensor readings for the pair of sensors
30.      if(fingerMS == fingerSi)
31.        distance ++;
32.      end if;
33.    end while;
// Calculate the weighted contribution from this sensor
34.    wCont = distance / sliding window size;
35.    eligibleStateValue[e] = eligibleStateValue[e] + wCont;
36.    totWeightCont = totWeightCont + wCont;
37.  end for;
38.end for;
// Now we can calculate the missing value
39 .missVal =
```

$$\sum_{e=1}^{all\_eligible\_states} (eligibleStateValue[e] * e) / totWeightCont$$

```
// Record the estimated value in the Buffer
40. Buffer[missingSensorID] = missVal;
```

Figure 8.c The estimateValue() Algorithm: Calculating the value for the *MS* using the eligible sensors

```
// Check for other missing values in the Buffer
41.for(int i = 0; i < numberOfSensors; i++)
42.  if(Buffer[i] == -1)
43.    missingValue = true;
44.    missingSensorID = i;
45.  end if;
46. end for;
47. if(missingValue) // There is a missing value
48.    invoke estimateValue(missingSensorID);
49. else
50.   send OK signal to queries;
51.   invoke update(); // To update the Cube and the Counter
52. end if;
```

Figure 8.d The estimateValue() Algorithm: Checking for other missing values in the Buffer and calling the appropriate algorithms.

5. Simulation Experiments

The performance of our proposed approach (WARM) is studied by means of simulation. Several different simulation experiments are conducted in order to evaluate the behavior of WARM and to compare WARM with some existing techniques.

Our simulation model consists of 108 sensor nodes, combining a vibration detector and a RF transmitter. All sensor nodes report to a single server. The sensors are deployed on city streets and collect and store the number of the vehicles detected for a given time interval. The actual vehicle counts taken as sensor readings that are used as input for our simulation experiments are traffic data provided by [1]. The data was collected in year 2000 at various locations throughout the city of Austin, Texas. The data represents the current location, the time interval, and the number of vehicles detected during this interval. From this set we generated four different input data sets corresponding to the different numbers of the possible sensor states used for the simulation experiments.

The dynamic parameters in the simulation experiments include the size of sliding window (winSize), which is the depth of the Cube, taking values 6, 18, 30, and 42 rounds; the minimum support and the minimum confidence (MSMC), taking values 0, 1, 4, 7, and 10%; the number of possible sensor states (numStates), taking values 10, 20, 40, and 80; and the error rate of a single-hop wireless link (errRate), which represent the probability that a sensor reading sent to the server by a RF transmission is lost or corrupted, taking values 10^{-3}, 10^{-2}, 10^{-1} [4]. The size of the sliding window, the MSMC, and the number of possible sensor states are studied because of the following reasons. All the three parameters affect the accuracy of the estimation and the percentage of cases in which the estimateValue() algorithm cannot produce an estimation. In addition, the first and third parameters affect the memory size WARM would need, while the first and second parameters affect the time that WARM would take to generate the estimation.

The static parameters in the simulation experiments are adopted from the literature. They include the main memory access time - 60 nsec per word [11]; the sensor battery power - 560 mAh [8]; the current consumption in transmit mode – 25mA, in receive mode – 17mA, and in idle mode – 10mA [8]; and the transmit data rate 0.4kbit/s [8].

5.1. Evaluation of Estimation Accuracy

The evaluation of the achieved accuracy of an estimation of the missing values is done by using the average root mean square error (RMSE).

$$RMSE = \frac{1}{numStates} \sqrt{\frac{\sum_{i=1}^{\#estimations} (Xa_i - Xe_i)^2}{\#estimations}},$$

where Xa_i and Xe_i are the actual value and the estimated value, respectively; *#estimations* is the number of estimations performed in a simulation run and *numStates* is the number of subsets, in which the actual readings are distributed.

The expression $\sqrt{\frac{\sum_{i=1}^{\#estimations}(Xa_i - Xe_i)^2}{\#estimations}}$ represents the standard error and is an estimate of the standard deviation under the assumption that the errors in the estimated values (i.e. Xa_i - Xe_i) are normally distributed. Thus, the RMSE allows the construction of confidence intervals describing the performance of different candidate missing value estimators. The smaller the RMSE (the standard deviation), the better the estimator. The calculated RMSE for each different set of input data (e.g. Set10 means that the sensor readings are split into 10 subsets) is divided by the number of subsets and the result is the average standard deviation in each case. This is done to keep the measure comparable across experiments.

This experiment is conducted for four different approaches for estimating the missing value:

- The Previous Value (PV) approach: using the previous value of the sensor with the missing value in the current round as an estimate;
- The Average Round (AR) approach: using the average of the available sensor readings in the current round as an estimate of the missing value;
- The Average Window Size (AWS) approach: using the average of all the readings stored in the Cube for the sensor with the missing value in the current round as an estimate;
- The combined (WARM) approach: using the estimateValue() algorithm when it can estimate the missing value and the AWS approach when the estimateValue() algorithm cannot estimate the missing value.

The resulting RMSEs for the different approaches are presented in Figure 9. It can be seen from Figure 9 that out of the three alternative approaches (PV, AR, AWS), the best estimator in the case of our sample application is the approach which calculates the missing value as an average of all the readings stored in the Cube for the sensor with the missing value (the AWS approach). For that reason the AWS approach was chosen to be used to estimate the missing value for the cases in which the estimateValue() algorithm cannot produce an estimation. The best accuracy of estimation using the WARM approach for the given input data is achieved for winSize = 42, MSMC = 1%, and numStates = 40. For these values of the dynamic parameters, the set of discovered related sensors to the sensor with the missing value is the most representative of the true relations between the locations in the real-world road system. This means that the number of the discovered truly existing relations between pairs of sensors is maximized, while the number of fake relations between the sensors that were able to pass the MSMC test in the estimateValue() algorithm is minimized.

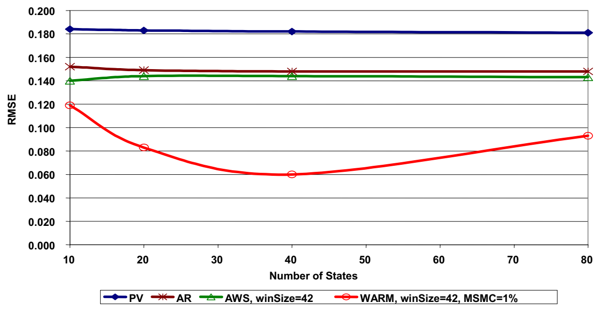


Figure 9. RMSE for PV, AR, AWS, and WARM

5.2. Evaluation of Main Memory Access Time per round (TMMAT)

As an approximation of the response times for performing an update and for estimating the missing value(s) we use the total time needed for performing memory accesses for each of the algorithms. The TMMAT is defined as the time for performing all main memory accesses required for update and estimate calls, per round of sensor readings. The probability of having *k* (*k* between 0 and 108) missing sensor readings in a round (out of 108 sensors total) is given by the Poisson distribution for each value of errRate. The TMMAT (in milliseconds) for WARM for the values of the dynamic parameters with which the best accuracy of the estimation is achieved, and as a function of the quality of the wireless link (errRate) is shown in Figure 10. As can be seen from Figure 10 even for the worst quality of the wireless communication link (10% of the sensor readings are lost or corruptted), and the maximum percentage of missing readings possibly occurring in a round (100% of all possible cases are considered), the TMMAT for the WARM approach is less than 35 milliseconds.

The TMMAT is an approximation of the response time - the time elapsed from receiving the readings from the sensors to the moment at which all the missing values in this round are estimated and the data model (the Cube and the Counter) is updated with the data in the completed round. The results shown in Figure 10 lead us to believe that the response time of WARM will provide an acceptable QoS (a function of the achieved accuracy of the estimated values and the response time) for a wide range of monitoring applications.

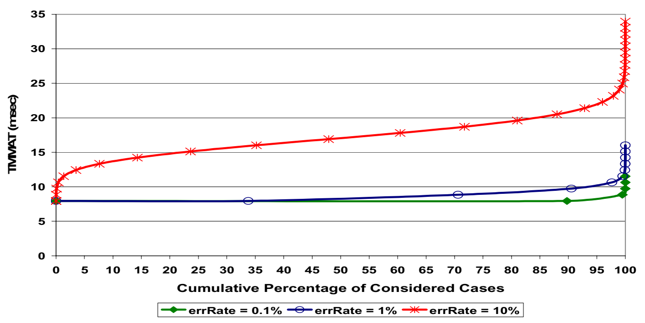


Figure 10. TMMAT (in miliseconds) per round of sensor readings for different error rates of the single-hop wireless link for WARM

5.3. Evaluation of Memory Space

The results of the simulation experiments show that the implementation of the WARM approach is feasible - the needed memory space (≈ 9MB) for 108 sensors, winSize equal to 42 rounds and numStates equal to 80, is less than the RAM memory provided in a contemporary computer.

5.4. Evaluation of Overall Power Consumption

The overall power consumption (OPC) amounts at the sensor nodes for WARM and for an alternative approach called MultiSend are compared. Under the MultiSend approach no estimation of the missing value(s) at the server is performed. In the case of a missing reading from a sensor, the sensor is explicitly pulled by the server to resend its reading. The comparison is done for different error rates of the single-hop wireless link between the sensor nodes and the server. For both approaches, the amount of power consumed (measured the mAh) by all sensor nodes for processing a round of readings is calculated. Then the maximum number of rounds that can be processed under the two different approaches using the same initial amount of battery power distributed at the sensor nodes is calculated. The experimental results for the OPC for the WARM and MultiSend approaches show that for the chosen values of the parameters, the number of rounds which the network can process using the WARM approach is 2.5 times bigger than the case when using the MultiSend approach. The main reason for this difference is the fact that for the MultiSend based approaches all the sensors in the network should spend an additional RF power waiting for a possible request from the server to resubmit their readings again.

5.5. Evaluation of the Percentage of Cases in Which a Missing Value Cannot be Estimated by the estimateValue() Algorithm Alone (PCE)

There is a possibility that the estimateValue() algorithm alone will not be able to estimate a missing value. This is due to the following reasons: 1) the sensor with the missing value (MS) does not have the actual support greater than the minimum support for any state, i.e., MS is not a 1-frequent itemset; 2) the MS is a 1-frequent itemset, i.e., its actual support is greater than the minimum support for at least one of the states, but no 2-frequent itemset consisting of the MS and any of the other sensors can be generated (the actual support of such 2-frequent itemset is less than the minimum support); and 3) the MS constitutes one or more 2-frequent itemsets with other sensors (S_j), but no association rule of the type $S_j \rightarrow$ MS can be generated because the actual confidence of such association rule is less than the minimum confidence for all S_j.

The PCE is computed for each simulation run using the formula:

$$\%ValuesCann\ otBeEstima\ ted = \frac{\#casesValue\ CannotBeEs\ timated}{totalNumbe\ rOfAttempt\ sToEstimat\ e} * 100\%,$$

where *#casesValueCannotBeEstimated* is the number of cases that a missing value cannot be estimated using the estimateValue() algorithm alone, and *totalNumberOfAttemptsToEstimate* is the total number of attempts to estimate a missing value in a given simulation run.

The effect of the sliding window size (winSize) and the MSMC parameters can be explained in the following way. A change of winSize and/or MSMC may lead to a change in the required number of occurrences (RNO) in which a pair of sensors must report the same state in order to pass the MSMC test. The reason for this is that the actual support (which should be greater than or equal to minSup in order to pass the MSMC test) for a pair of sensors is calculated as the ratio of the number of occurrences in which a pair of sensors reports the same state and the window size, i.e., actSup = (reqNumOccurr/winSize)*100%. When increasing the window size in order to achieve the same actSup (needed to pass the MSMC test), the RNO may increase as well. On the other hand, a change of MSMC sometimes leads to a change to the RNO as well.

The experiment results show that for an increase of the winSize that does not lead to an increase of the RNO, there is a decrease in the PCE. This result should be expected since increasing the window size is equivalent to having a longer history of sensor readings stored in the data model. On the contrary, when an increase of the window size leads to an increase of the RNO, then there is an increase in the PCE. This is because fewer pairs of sensors will be able to pass the MSMC test, regardless of the fact that there is longer history stored in the data model.

Increasing the MSMC value leads to an increase in the PCE in general. The reason for this is that by increasing the MSMC value we pose stricter requirements to pairs of sensors in the MSMC test. In some cases, the PCE remains the same, regardless of the increase of the MSMC value. This can be explained by the fact that even we increase the MSMC value, the RNO in which a pair of sensors must report the same state remains unchanged.

Another dynamic parameter that affects the PCE is the number of possible sensor states. The experiment results show that by increasing the number of states the PCE also increases. The reason for this behavior is the fact that by increasing the number of states (i.e. decreasing the range for each subset of possible sensor states), we get a smaller number of the sensor readings that will be assigned to a particular subset. Having the sensor readings distributed among more subsets' results in discovering fewer relations between subsets, and consequently, between pairs of sensors.

6. Conclusions and Future Research

This research has proposed an approach called WARM (Window Association Rule Mining) for estimating missing values in related data streams. WARM uses association rule mining in order to determine the sensor nodes that are related to the sensor with the missing reading. The readings of of the related sensors in the current round participate in estimating the missing value. When a missing value cannot be estimated by using association rule mining, it is estimated using the average of all available readings for the sensor with the missing value.

Performance evaluations by means of simulation were conducted to compare WARM and alternative approaches in terms of estimation accuracy, memory access time for estimation, sensors power consumption, memory space required, and percentage of cases in which a missing value cannot be estimated using association rule mining alone. The test data is the real traffic data collected by the Department of Transportation in Austin, Texas [1]. The simulation results show that although WARM requires more memory space and take longer to produce an estimation than the considered alternative approaches, it achieves better accuracy of the estimated value than the alternative approaches do. The memory space needed by WARM is feasible for a contemporary computer, and the time needed for producing the estimation of a missing value by WARM is acceptable for many applications.

For future research, the DSARM framework should be enriched so that it will support mining for association rules of type $S_i \mid stateA \rightarrow MS \mid stateB$ in order to reduce the percentage of cases in which a missing value cannot be estimated by the estimateValue() algorithm and to achieve a better accuracy of the estimated value. A weight assignment strategy assigning more weight to the events that happened sooner to the present moment than to the events that happened further in the past should be investigated. Also, the case of multiple sensor failure of co-related sensors should be considered.

References

[1] Austin Freeway ITS Data Archive, http://austindata.tamu.edu/default.asp, accessed January, 2003

[2] R. Agrawal, T. Imielinski, A. Swami. "Mining Association Rules between Sets of Items in Large Databases". *ACM SIGMOD Conference, pp. 207-216*, May 1993.

[3] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. "Models and Issues in Data Stream Systems". *ACM SIGMOD/PODS 2002 Conference*

[4] S. Banerjee and A. Misra. "Minimum Energy Paths for Reliable Communication in Multi-hop Wireless Networks", *ACM Mobihoc 2002*, June 2002.

[5] S. Brin, R. Motwani, J. Ullman, S. Tsur. "Dynamic Itemset Counting and Implication Rules for Market Basket Data". *ACM SIGMOD Conference, pp. 255-264*, May 1997.

[6] D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. "Monitoring Streams - A New Class of Data Management Applications". *the 28th VLDB Conference*, 2002

[7] A. Chandrakasan, R. Amirtharajah, S. Cho, J. Goodman, G. Konduri, J. Kulik, W. Rabiner, and A. Wang. "Design Considerations for Distributed Microsensor Systems". *IEEE 1999 Custom Integrated Circuits Conf., pp. 279-286*, May 1999

[8] Chipcon Inc., http://www.chipcon.com/, accessed Feb., 2003

[9] L. Clare, G. Pottie, and J. Agre. "Self-Organizing Distributed Sensor Networks". *SPIE Conference on Unattended Ground Sensor Technologies and Applications, pp. 229-237*, Apr. 1999

[10] M. Datar, A. Gionis, P. Indyk, and R. Motwani. "Maintaining Stream Statistics over Sliding Windows". *ACM Symposium on Discrete Algorithms, pages 635-644*, 2002

[11] C. Kozierok, "The PC Guide", http://www.pcguide.com/ref/ram/timingRatings-c.html, accessed February, 2003.

[12] S. Madden and M. Franklin. "Fjording the Stream: An Architecture for Queries over Streaming Sensor Data". *International Conference on Data Engineering*, 2002

[13] G. McLachlan and K. Thriyambakam. "The EM Algorithm and Extensions", *John Wiley & Sons*, 1997

[14] D.Rubin. "Multiple Imputations after 18 Years". *Journal of the American Statistical Association, 91, pp. 473-478*, 1996

[15] H. Tirri and T. Silander. "Stochastic Complexity Based Estimation of Missing Elements in Questionnaire Data". *the Annual American Educational Research Association Meeting, SIG Educational Statisticians,* 1998.

[16] O. Troyanskaya, M. Cantor, G. Sherlock, P. Brown, T. Hastie, R. Tibshirani, D. Botstein, and R. Altman. "Missing Value Estimation Methods for DNA Microarrays", *Bioinformatics, 17, pp. 520-525*, 2001.