

# In-network Historical Data Storage and Query Processing Based on Distributed Indexing Techniques in Wireless Sensor Networks

Chunyu Ai<sup>1</sup>, Ruiying Du<sup>1,2</sup>, Minghong Zhang<sup>1,3</sup>, and Yingshu Li<sup>1</sup>

<sup>1</sup> Georgia State University, 34 Peachtree St., Atlanta, GA, USA

<sup>2</sup> Wuhan Computer School of Wuhan University, Wuhan, China

<sup>3</sup> Department of Mathematics, Graduate University, Chinese Academy of Sciences, Beijing, China

**Abstract.** Most of existing data processing approaches of wireless sensor networks are real-time. However, historical data of wireless sensor networks are also significant for various applications. No previous study has specifically addressed distributed historical data query processing. In this paper, we propose an Index based Historical Data Query Processing scheme which stores historical data locally and processes queries energy-efficiently by using a distributed index tree. The simulation study shows that our scheme achieves good performance on both query responding delay and network traffic.

**Key words:** Historical Data, Distributed Index, Historical Data Query, Wireless Sensor Networks.

## 1 Introduction

Nowadays large-scale sensor networks are widely deployed around the world for various applications. Sensor networks are used to report live weather conditions, monitor traffic on highways, detect disasters, monitor habitat of animals, etc. Tremendous volumes of useful data are generated by these deployments. Most existing applications just process real-time data generated by sensor networks (e.g., [1, 2]). However, historical data of sensor networks are also significant to us, especially statistical meaning of historical data. For instance, maximum, minimum, and average temperatures of the past two months in a specific area are concerned in the weather monitoring application. By capturing rush hours and the bottleneck of traffic according to historical data, a large quantity of useful information can be provided to improve traffic conditions. Through analysis of historical data, some knowledge, principles, and characteristics can be discovered.

One simple method to process historical data is that the base station collects all data and processes in a centralized manner. Nevertheless, sensor nodes will deplete energy rapidly for continually reporting and forwarding data. Another method is storing data locally, that is, data are stored at a sensor node where

they are generated. Intuitively, a sensor node cannot store a large quantity of historical data since its memory capacity is low. Popular motes such as Mica2 and Micaz [3] are equipped with a 512K bytes measurement flash memory, it can store more than 100,000 measurements. Another popular mote, TelosB [3], has a 1024K bytes flash memory. 512K or 1024K bytes are really small memory capacity. However, it is enough to store most of sensing data, sampling data, or statistical data during a sensor node's entire lifetime since the lifetime of a sensor node is short due to the limitation of batteries. For a Mica mote powered by 2 AA batteries, the lifetime is 5-6 days if continuously running, and it can be extended to over 20 years if staying sleep state without any activity [4]. For most applications, a sensor can live for several weeks or months [5]. Assume a Mica mote with a 512K bytes flash memory can live 3 months.  $(100,000/90) = 1111$  measurements can be saved locally every day. Suppose we have 4 sensing attributes need to be saved, frequency of sampling can be 1 per 5 minutes which is normal in wireless sensor network applications. If proper compressing techniques are applied according to data's characteristics, much more data can be stored locally. Consequently, storing data locally at a sensor is feasible. The historical data can be downloaded when the battery is replaced or recharged if possible.

The motivation of storing historical data is to support historical data queries. However, locally storing data might cause a query to be flooded in the entire network to probe data when it is processed. We propose a scheme in this paper named Historical Data Query Processing based on Distributed Indexing (HDQP) which stores historical data in network and processes queries energy-efficiently by using index techniques. Historical data is stored at each sensor. For saving memory capacity, compressing and sampling techniques can be used according to data characteristics and users' requirements. To process queries quickly and energy-efficiently, in-network distributed tree-based indexes are constructed. Using indexes to process queries can reduce the number of involved sensors as many as possible, thus conserving energy consumption. To avoid load skew, index data are partitioned on different nodes according to their time stamp. That is, there exist multiple index trees in the network. Which index trees are used depends on query conditions.

The rest of this paper is organized as follows. Section 2 discusses related work. Section 3 describes how to store historical data at a sensor node. Constructing and maintaining the distributed index trees are addressed in Section 4. Section 5 explains how to process historical data queries. Simulation results and analysis are presented in Section 6. Finally, we conclude in Section 7.

## 2 Related Work

Many approaches have been proposed to describe how to store data for sensor networks. One category of such storage solutions is that the base station collects and stores all data. However, such approaches might be more applicable to answer *continuous queries*. Obviously, the mortal drawback of collecting all data is shortening the network lifetime.

For improving network lifetime, in-network storage techniques have been addressed to solve ad-hoc queries. These frameworks are primarily based on the Data-Centric Storage (DCS) concept [6]. In DCS, relevant data are stored by *name* (e.g., tiger sightings). All data with the same general name will be stored at the same sensor node. Queries for data with a particular name can be sent directly to the node storing those named data. In-network DCS schemes differ from each other based on the events-to-sensors mapping method used.

The works in [7], [8], and [9] use a distributed hashing index technique to solve range queries in a multi-dimensional space. The work in [2] proposed a distributed spatio-temporal index structure to track moving objects.

Specially, most of these approaches just store partial data, which satisfy conditions or present events and moving objects, generated by sensors. To our best knowledge, no in-network distributed historical data storage, indexing, and query processing schemes have been presented in the literature.

### 3 Historical Data Storage

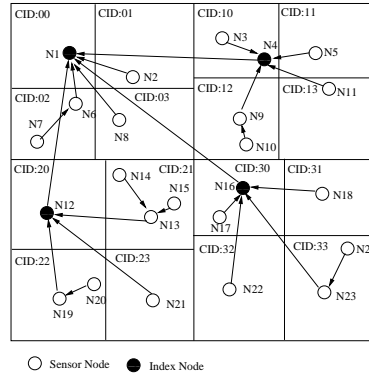
Since a sensor node's memory capacity is limited, we have to fully utilize it to store as many data as possible. The administrator of the sensor network can specify the attributes which need to be stored locally according to users' requirements. Suppose there are  $n$  attributes  $(A_1, A_2, \dots, A_n)$  need to be saved, and the sensor node senses the values  $V_1, V_2, \dots, V_n$  of  $A_1, A_2, \dots, A_n$  respectively at sensing intervals. For each sensing interval, a record with the format  $(T, V_1, V_2, \dots, V_n)$  (where  $T$  is a time-stamp) is written to the flash memory. However, if the sensing interval is very short such as 5 seconds, the flash memory will be full filled quickly. Then the rest of coming sensing data cannot be stored. To avoid this happen, whenever the flash memory is occupied more than 80%, a *weight-reducing* process is triggered. One record of every two consecutive records is erased. This weight-reducing process can make at least half of memory space available again. Applying the weight-reducing process repeatedly causes partial historical data lost. However, as we mentioned in Section 1, even though the flash memory capacity is small, it still can store 5 to 10 values per hour for each attribute. Consequently, the historical data stored in the flash memory can reflect the changing trend of each sensing attribute.

Each sensor node needs to calculate the maximum, minimum, and average for each attribute periodically. The administrator specifies an *update interval* which is much greater than the sensing interval. For instance, if the sensing interval is 5 minutes, the update interval can be 2 hours. At the end of each update interval, a sensor node sends the *index update message* including the maximum, minimum, and average values of that interval to its parent node of the index tree (we will discuss it in the next section).

## 4 Construct and Maintain Distributed Index Tree

Constructing effective distributed index structures can help process queries efficiently. In this section, we introduce how to construct and maintain distributed index trees.

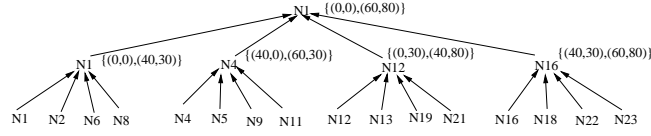
### 4.1 Construct a Hierarchical Index Tree



**Fig. 1.** Network Partition

We assume that any point in the monitored area is covered by at least one sensor node. Moreover, the sensor network is static, and absolute or relative locations (2-D cartesian coordinates) of sensor nodes are known via using GPS or location algorithms. This is necessary since users use the location as query conditions or results sometimes. At the base station, the entire network is divided into four subregions in vertical and horizontal directions, and each subregion has almost the same number of sensor nodes. Then, for each generated subregion, it is continuously divided into four subregions with almost the same number of sensor nodes. This partition process keeps going until there is no enough sensor nodes to promise that every generated subregion has at least one sensor node. A partition example is shown in Figure 1. The entire network is divided into a lot of rectangles called cells in this paper. Each cell has at least one sensor node. In each iteration of the partition, a generated subregion is assigned an ID from 0 to 3 according to its relative position to others respectively. Finally, the cell id (CID) is the combination of IDs generated by all partition steps (as shown in Figure 1). After partition, the location and CID of each cell are recorded and sent to the sensor nodes within that cell. We use 2-D coordinates of the cell's top-left and bottom-right vertices,  $\{(x_{tl}, y_{tl}), (x_{br}, y_{br})\}$ , to describe the cell's geographical position.

For processing queries efficiently, a hierarchical index tree is constructed. An index tree example is shown in Figure 1 and 2. Firstly, each cell randomly



**Fig. 2.** Index Tree

chooses a sensor node as the leader of this cell. Other nodes in that cell set the leader as their parent. Since only the leader has a chance to be the index node, to maintain energy balance, sensor nodes in the same cell serve as the leader in turn. In the example, node  $N_{13}$ ,  $N_{14}$ , and  $N_{15}$  are in the same cell, and node  $N_{13}$  is chosen to be the leader of  $N_{14}$  and  $N_{15}$ . If a cell only has one sensor node, the only node is chosen to be the leader naturally. Then, the index tree is established in a reverse manner of network partition. In other words, the index tree is constructed from bottom to top. For each group of four cells which are generated from the same subregion, the leader of the top-left cell is chosen to be the parent of these four cells' leaders including itself. Then, for each group of four generated parent nodes which belong to the same subregion in the partition process, the top-left one is chosen to be the parent. The process continues until the root is generated. In the example shown in Figure 1, Node  $N_1$  is chosen to be the parent of Node  $N_1$ ,  $N_2$ ,  $N_6$ , and  $N_8$ . Node  $N_4$ ,  $N_{12}$ , and  $N_{16}$  are chosen to be the parents of other three groups respectively. Next step, Node  $N_1$  is chosen to be the parent of  $N_1$ ,  $N_4$ ,  $N_{12}$ , and  $N_{16}$ , and the root  $N_1$  is generated. The structure of the index tree is shown in Figure 2. Obviously, the index tree is a quad-tree, and all inter nodes have four children.

If we use the same index tree during the entire network lifetime of networks, the accumulative index data will occupy most memory space quickly, especially the top layer nodes. Furthermore, index nodes will use up energy much earlier than others because the query processing frequently accesses these nodes. To balance index data distribution and energy consumption among sensor nodes, a set of index trees is generated by switching nodes' positions in the index tree periodically.

The administrator needs to specify a *tree switching period*  $P_S$ . The ideal value of  $P_S$  is the estimated network lifetime divided by the number of sensor nodes. Thus, every node has a chance to sever as an index node. Four children of the root serve as the root node in turn. For the root node (first layer), the switching period is  $P_S$ . Each node in the second layer also makes its four children serve as the parent in turn. However, the switching period of nodes in the second layer is  $4 * P_S$ . The switching period for nodes in the layer  $L$  is  $4^{L-1} * P_S$ . The lower the layer, the longer the switching period. In fact, in our index tree structure, a higher layer index node contains more data than a lower layer index node since an internal node appears at each layer of the subtree rooted at itself. For example, the root node  $N_1$  in Figure 2 appears in every layer of the index tree. Our schedule replaces the higher layer nodes more frequently than lower layer nodes, thus evenly distributing index data to all sensor nodes.

Through switching nodes in the index tree, a large number of index trees are available. However, a sensor node does not need to store all these index trees. Since we follow the rules to switch nodes, a sensor node can calculate its current parent according to the current time-stamp  $T$ . For a node in the ( $L$ )th layer of an index tree, its parent's CID at time  $T$  can be calculated by replacing its own CID's ( $L - 1$ )th bit with  $(\lfloor T / (4^{L-1} * P_S) \rfloor \% 4)$ . Geographic routing is used to generate routes among nodes of the index tree since the CID of a sensor node also indicates its relative position.

## 4.2 Index Maintaining

**Table 1.** Index Sturcture

Child0:	Child1:	Child2:	Child3:
Location: $\{(x_{tl}, y_{tl}), (x_{br}, y_{br})\}$	Location: $\{(x_{tl}, y_{tl}), (x_{br}, y_{br})\}$	...	...
Attribute1: $T_i, \max, \min, \text{avg}$	Attribute1: $T_i, \max, \min, \text{avg}$		
$T_{i+1}, \max, \min, \text{avg}$	$T_{i+1}, \max, \min, \text{avg}$		
⋮	⋮		
Attribute2: $T_i, \max, \min, \text{avg}$	Attribute2: $T_i, \max, \min, \text{avg}$		
⋮	⋮		
⋮	⋮		
⋮	⋮		

The leader of each cell is responsible for calculating the maximum, minimum, and average values of each attribute for the cell it belongs to periodically. If there are more than one sensor node in a cell, the maximum, minimum, and average values are calculated among all sensing values of the current interval from these sensor nodes. At the end of each update interval, the leader sends the update message,  $(T, \max, \min, \text{avg})$  (where  $T$  is the beginning time stamp of the current time interval), to its parent in the index tree. The update message also includes the time stamp and node ID of the max and min values. The internal node in the index tree maintains the received index data with the structure as shown in Table 1. When an update message is received, it is inserted into the index structure. The internal node also merges four update messages for the current time interval  $T$  by calculating max, min, and avg, and sends this update message to its parent.

## 5 Historical Data Query Processing

Historical data queries can inquire max, min, and avg values for a past period of time in a specified geographic area. Also, the sensing values of specific sensor nodes or locations in the past can be retrieved. A historical query can be issued at the base station or a sensor node in the network which is named as *query node*.

**Query Example1:**

```
SELECT S.temperature, S.humidity, S.location FROM sensor S
WHERE S.location WITHIN {(10, 10), (30, 30)} AND S.time BETWEEN now()-1 days AND now()
```

**Query Example2:**

```
SELECT MAX(S.temperature), S.location FROM sensor S
WHERE S.location WITHIN {(0, 0), (50, 30)} AND S.time BETWEEN 02/01/2009 and /02/04/2009
```

If the query node receives a query like Query Example1, the query is forwarded to the query location  $\{(10, 10), (30, 30)\}$ . When a sensor node within  $\{(10, 10), (30, 30)\}$  receives this query, it forwards this query to the closest ancestor node (node  $N1$  at the second layer in Figure 1) of the index tree, whose location contains the query location. Then,  $N1$  accesses its indexes and sends the query to all its children which have intersection with the query location. Thus, the query is sent to sensor nodes which possibly satisfy the query conditions through the index tree. When a sensor node receives a query, it verifies itself. If the conditions are satisfied, it sends corresponding results back to the query node. In this query example, the index tree can bound the number of involved sensor nodes efficiently. Only cells which have intersection with location  $\{(10, 10), (30, 30)\}$  are probed.

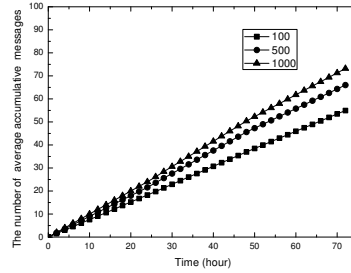
For Query Example2, the query node forwards the query to the lowest layer index node (the root  $N1$  in Figure 1), which contains the query location. Then,  $N1$  probes  $\text{MAX}(\text{S.temperature})$  within  $\{(0, 0), (40, 30)\}$  from  $N1$  and  $\text{MAX}(\text{S.temperautre})$  within  $\{(40, 0), (50, 30)\}$  from  $N4$  between 02/01/2009 and /02/04/2009 separately. For location  $\{(0, 0), (40, 30)\}$ , we can calculate the index nodes, which store the  $\text{MAX}$  value between 02/01/2009 and /02/04/2009, and merge results from these index nodes to get  $\text{MAX}(\text{S.temperautre})$  within  $\{(40, 0), (50, 30)\}$ . However, since  $\{(40, 0), (50, 30)\}$  does not perfectly match  $N4$ 's location  $\{(40, 0), (60, 30)\}$ ,  $N4$  has to further probe some of its children, then merges results returned by its children to get  $\text{MAX}(\text{S.temperautre})$ . Finally, the root  $N1$  merges results returned by  $N1$  and  $N4$  to get the  $\text{MAX}(\text{S.temperature})$  within  $\{(0, 0), (50, 30)\}$  and sends the results back to the query node. Usually, users send queries like Query Example2 to inquire statistical information from wireless sensor networks.

The distributed index tree can guide queries to be processed efficiently and restrict the number of involved sensor nodes. For some queries, the results are already stored in the index structure, so accessing one or several index nodes instead of a large number of sensor nodes can acquire the results.

## 6 Simulation

In this section, we evaluate the performance of our proposed HDQP. The sensing interval of a sensor is 5 minutes, and the update interval  $P_S$  of the index tree is 2 hours. Since network traffic greatly affects energy efficiency, we use it as the metric for performance evaluation.

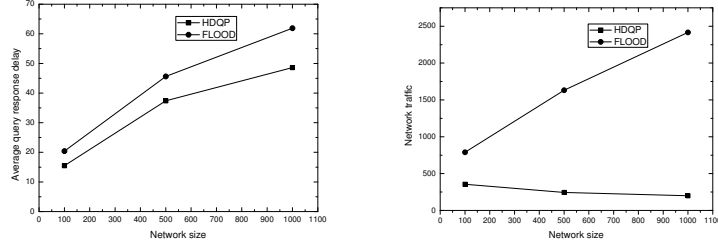
The main cost of maintaining the index tree is sending and forwarding update messages along the index tree periodically. Figure 3 shows the number of average



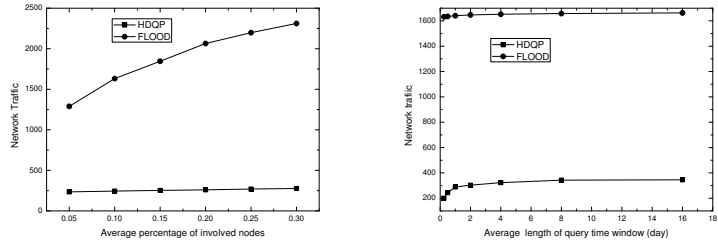
**Fig. 3.** Cost of maintaining the index tree with various network size.

accumulative messages of sensor nodes with 100, 500, and 1000 sensor nodes in the network respectively. The more sensor nodes the network has, the more cost of maintaining the index tree. The reason is that a bigger network must generate a bigger index tree structure, thus increasing the maintaining load. Since for a sensor node, one time index updating just causes about 2 messages averagely, and the index tree is not updated frequently, so the cost of maintaining the index tree is acceptable for wireless sensor networks with limited energy. Furthermore, as can be seen in Figure 3, the maintaining cost does not rapidly increase with the increasing of the network size. Therefore, the distributed index structure is also suitable for large-scale wireless sensor networks.

The performance of our HDQP is evaluated on two aspects, delay of query processing and cost of network traffic. The delay of query processing is query responding time since the query is issued until the user receives results. In our simulation, computation delay of sensor nodes is ignored, so this delay is measured by the number of hops of the longest path for sending a query and returning results. Network traffic is defined as the average number of messages sent and forwarded by all sensor nodes. Figure 4 shows the average query responding delay and network traffic (including cost of maintaining the index tree) of processing 1000 queries during 72 hours. We compare our HDQP with Flood method where the query node floods the query to entire network and all sensor nodes satisfied query conditions send corresponding data back to the query node. As seen in Figure 4(a), with the increasing of network size, query responding delay increases too since the length of paths for sending queries and returning results increases with the network size. HDQP achieves shorter delay than Flood because it does not need to probe all qualified sensor nodes, and partial or all results can be acquired from the index tree. Figure 4(b) shows that HDQP is much better than Flood on network traffic especially for large-scale networks. This is because the index tree can help HDQP to answer queries by accessing index nodes and a small number of qualified sensor nodes if necessary. However, Flood method lets all qualified sensor nodes involve in the query processing. Network traffic of HDQP decreases with the network size increasing since for processing the same



(a) Query responding delay of variant network size. (b) Average Network traffic of variant network size.



(c) Average Network traffic of variant average involved nodes percentage of queries. (d) Average Network traffic of variant average length of query time window.

**Fig. 4.** Query Processing Performance, 1000 queries during 72 hours.

number of queries, a large-scale network has much more nodes to share work load. Figure 4(c) shows the average network traffic with variant involved sensor nodes percentage of queries. This parameter is decided by the location condition in a query. As shown in the figure, with the increasing of involved node percentage, the network traffic of Flood increases obviously since all involved nodes need to report results. However, HDQP is not affected by this parameter obviously, since HDQP acquire most of data from the index trees. The average network traffic with variant length of query time window is shown in Figure 4(d). With the increasing of time window length, the network traffic of HDQP increases too. Since index data are partitioned by time stamp, more index nodes are accessed for processing queries with a longer time window. When the length of time window reaches a certain value, the network traffic does not evidently increase any more. The reason is that when the time window is long than 4 times of partition period of the accessed layer, the number of accessed index nodes does not increase any more. However, an index node may include index data of several time slices.

In summary, our HDQP scheme can process historical data query quickly and energy-efficiently, and it is also suitable for large-scale networks.

## 7 Conclusion

In this paper, we have proposed a scheme, HDQP, to process historical data queries of wireless sensor networks. Our approach is the first work to study distributed historical data query processing by using an effective distributed index tree. The indexes can help process queries energy-efficiently and reduce the query responding delay. Index tree switching mechanism also can balance the load among sensor nodes to avoid data distribution and energy consumption skew.

## Acknowledgment

This work is supported by the NSF under grant No. CCF-0545667 and CCF 0844829.

## References

1. M. Aly, A. Gopalan, J. Zhao, and A. Youssef, "Stdcs: A spatio-temporal data-centric storage scheme for real-time sensornet applications," in *Sensor, Mesh and Ad Hoc Communications and Networks, 2008. SECON '08. 5th Annual IEEE Communications Society Conference on*, June 2008, pp. 377–385.
2. A. Meka and A. Singh, "Dist: a distributed spatio-temporal index structure for sensor networks," in *CIKM '05: Proceedings of the 14th ACM international conference on Information and knowledge management*. New York, NY, USA: ACM, 2005, pp. 139–146.
3. "Crossbow - wireless sensor networks - products - wireless modules," <http://www.xbow.com/Products/productdetails.aspx?sid=156>. [Accessed February 03, 2009].
4. D. Malan, T. Fulford-jones, M. Welsh, and S. Moulton, "Codeblue: An ad hoc sensor network infrastructure for emergency medical care," in *International Workshop on Wearable and Implantable Body Sensor Networks*, 2004.
5. S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "Tinydb: an acquisitional query processing system for sensor networks," *ACM Trans. Database Syst.*, vol. 30, no. 1, pp. 122–173, 2005.
6. S. Shenker, S. Ratnasamy, B. Karp, R. Govindan, and D. Estrin, "Data-centric storage in sensornets," *SIGCOMM Comput. Commun. Rev.*, vol. 33, no. 1, pp. 137–142, 2003.
7. B. Greenstein, D. Estrin, R. Govindan, S. Ratnasamy, and S. Shenker, "Difs: a distributed index for features in sensor networks," in *Sensor Network Protocols and Applications, 2003. Proceedings of the First IEEE. 2003 IEEE International Workshop on*, May 2003, pp. 163–173.
8. X. Li, Y. J. Kim, R. Govindan, and W. Hong, "Multi-dimensional range queries in sensor networks," in *SenSys '03: Proceedings of the 1st international conference on Embedded networked sensor systems*. New York, NY, USA: ACM, 2003, pp. 63–75.
9. M. Aly, K. Pruhs, and P. K. Chrysanthis, "Kddcs: a load-balanced in-network data-centric storage scheme for sensor networks," in *CIKM '06: Proceedings of the 15th ACM international conference on Information and knowledge management*. New York, NY, USA: ACM, 2006, pp. 317–326.