Continuous Answering Holistic Queries over Sensor Networks

Kebin Liu, *Member, IEEE*, Lei Chen, *Member, IEEE*, Yunhao Liu, *Fellow, IEEE*, Wei Gong, *Member, IEEE*, and Amiya Nayak, *Senior Member, IEEE*

Abstract—Sensor networks are widely used in various domains like the intelligent transportation systems. Users issue queries to sensors and collect sensing data. Due to the low quality sensing devices or random link failures, sensor data are often noisy. In order to increase the reliability of the query results, continuous queries are often employed. In this work we focus on continuous holistic queries like Median. Existing approaches are mainly designed for non-holistic queries like Average. However, it is not trivial to answer holistic ones due to their non-decomposable property. We first propose two schemes based on the data correlation between different rounds, with one for getting the exact answers and the other one for deriving the approximate results. We then combine the two proposed schemes into a hybrid approach, which is adaptive to the data changing speed. We evaluate this design through extensive simulations. The results show that our approach significantly reduces the traffic cost compared with previous works while maintaining the same accuracy.

Index Terms—Ubiquitous computing, sensor networks, distributed data structures

1 Introduction

Sensor networks are now widely used in many applications, from habitat monitoring to location tracking and intelligent transportation systems. In these applications, sensors are often deployed in a large area to obtain measurements of various kinds of parameters. In order to collect and analyze sensing data, users issue various queries, such as selection query, aggregate queries (*Max*, *Count*, and *Sum*) and etc. In general, aggregate queries can be classified into two different categories, *non-holistic* queries and *holistic queries* [11]. Non-holistic queries [13], such as *Count* and *Sum*, share the *decomposable* characteristic, that is, for an arbitrary query Q on data set S, there is a function f, such that for all $S = S_1 \cup S_2, Q(S) = f(Q(S_1), Q(S_2))$. If there is no such a function f, the query Q is *holistic*. For example, a *quantile* query means to find the certain value that has the user specified rank among all the values, for which we cannot find such a function f. There are many differences between these two types of queries. Non-holistic queries usually provide one single result (e.g., count value or sum value). Moreover, since these queries are decomposable, partial results from downstream sensor nodes can be combined to one single result in an intermediate node without loss of information. For example, with a tree-like routing structure [21], [23], the partial result of an *Average* query contains only two variables: the number of sensors and the summation of all values in a sub-branch rooted at the intermediate node. In a *quantile* query, however, the distribution information of all sensor values must be collected. Many efforts have been made for handling non-holistic queries, such as TAG [21], [23], Cougar [34] etc., but few of them, if any, addressed holistic queries.

Holistic queries are actually as popular as non-holistic queries. For example, when the values sensed by sensors have noise, it makes more sense to get a median result of the monitoring area than to derive an average result, since the noise may affect the average result largely. Thus, holistic queries can be used to get the approximate data distribution.

To answer holistic queries in a sensor network, many challenging issues need to be addressed. Obviously, a naïve approach requiring all sensors to deliver their values to the sink is energy inefficient. To save power, in-network aggregation [21] is introduced, in which messages from downstream nodes are merged into one, and the computation is distributed within the entire network. Due to the limitation of message size, the merging process may lose some information. Many prior attempts have been done to create sophisticated data structures and algorithms which try to keep the most useful information with a limited message size. Generally, these methods can only achieve approximate results with some error guarantee by introducing different restricts and pruning algorithms on the data structure [13], [29]. In many cases, we need the exact answers from the network. Furthermore, in many application scenarios, especially under tough environments such as coal mines and underwater, a WSN prefers continuous queries, and one-shot result is regarded as noisy and unreliable. In other words, the same query process needs to be executed periodically in

- *K. Liu, Y. Liu, and W. Gong are with the MOE Key Lab for Information System Security, School of Software, Tsinghua National Lab for Information Science and Technology, Tsinghua University, Beijing, China. E-mail: {kebin, yunhao, gongwei}@greenorbs.com.*
- *L. Chen is with the Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Hong Kong. E-mail: leichen@cse.ust.hk.*
- *A. Nayak is with the School of Electrical Engineering and Computer Science, University of Ottawa, Ottawa, ON, Canada. E-mail: nayak@uottawa.ca.*

Manuscript received 25 Apr. 2014; revised 10 Dec. 2014; accepted 6 Feb. 2015. Date of publication 26 Feb. 2015; date of current version 20 Jan. 2016. Recommended for acceptance by H. Jin.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2015.2407892

order to improve accuracy and reliability. Strong correlations (temporal correlations) exist between data in different rounds, which are often overlooked by previous approaches in answering holistic queries.

In this paper, we explore the correlation between data of different rounds and present two approaches to monitor continuous holistic queries, one for getting the exact answers and the other one for deriving the approximate results. Then, we propose an effective hybrid approach that adaptively selects appropriate one based on data changing speed and improves the performance of continuous queries. We take one example of the *holistic* queries, *Median*, to illustrate our proposals. *Median* is a typical as well as important type of holistic queries. For example, according to our experience in deploying the GreenOrbs [20] system, a forest surveillance sensor network with up to 330 sensor nodes, we find that it is usually of great significance to derive the general status of a monitoring area rather than single reading. This requirement can be fulfilled by fusing sensor readings from different nodes with the *Average* or *Median* query. Sensor nodes, however, are error prone that can report noisy or even error readings. In this case, the results of *Average* query can be largely affected by a small number of outliers and deviate from ground-truth. In contrast, the *Median* query is resistant to noisy readings and thus can provide robust and accurate result. In fact, the proposed methods can be naturally extended for other types of holistic queries. We will discuss the generalization in Section 7. The major contributions of this work are as follows:

1) We present a flexible bucket (F-Bucket)-based histogram approach to compute exact answer to a query. A histogram summary is used to collect the value distribution information. The query of a new round is guided by prior results so that the temporal correlations between two rounds are exploited. After several initial rounds, we can obtain the exact answers continuously.
2) We also employ a wavelet-like approximate algorithm to offer a reasonable approximate result with some error bound guarantee, especially when the sensing values in the sensor network change quickly.
3) We propose a hybrid approach, combining F-Bucket and wavelet-like approaches, to handle the continuous holistic queries. If the sensor values in network are relatively stable, we apply an exact algorithm to calculate the exact median. When the data changes quickly, our approach can adaptively switch to the approximate algorithm.

The rest sections of this paper are organized as follows. Section 2 illustrates prior works that are related to ours. Section 3 presents the overall process and refining algorithms of exact query algorithm. The wavelet-like approximate algorithm is presented in Section 4. Section 5 discusses the hybrid approach. In Section 6 we evaluate our algorithms with simulation experiments. We conclude this work in Section 7.

2 Related Work

Query processing in sensor networks has drawn significant attentions from worldwide researches. Users issue various types of queries, such as selection queries [16] and aggregate queries [28], [32]. L-PEDAPs [31] focused on routing tree construction and maintenance for query processing. Uncertainties may exist in both sensing data [1] and queries. For example, Zhang et al. [39] studied the problem of calculating the aggregate while the query location is uncertain. Ye et al. [35], [36] proposed to determine possible query results among all imprecise sensing data. Security issues [37] have been considered in query processing. By combining *homomorphic* encryption and *secret sharing*, SIES [26] achieved both confidentiality and integrity. While encryption-based aggregates provided better security, they, however, still had limitations in the usage of aggregation functions and the data integrity. RCDA [3] attempted to overcome these drawbacks by introducing "recoverability" to aggregate queries by which servers can recover all sensing data with low extra overhead. Yu et al. [38] presented a method which aiming at secure continuous aggregation querying.

Other than the approaches designed for answering aggregate queries, some data collection techniques [6], [7], [17] can be used to answer aggregate queries as well. However the focuses of those approaches are efficient data collection, not only on answering aggregate queries. Consequently we will review the work for decomposable aggregate queries and holistic aggregate queries.

As mentioned in Section 1, for the decomposable aggregate queries, there are already much works that have been done, such as TAG [21], [23], BBQ [9], Cougar [34], TiNA [30], Sketch [5], PGA [10]. TAG [21], [23] introduced the in-network aggregation scheme to minimize the amount of transmitted messages. BBQ [9] proposed using the statistical modeling techniques to answer various queries. In Cougar [34] approach, when given a user query, a query optimizer generates an efficient query plan for query processing. TiNA [30] exploited the temporal correlation between readings. Sketch [5] generalized the duplicate-insensitive characteristics to answer the aggregate queries such as Count, SUM. PGA [10] exploited range caching [10], [25] to reduce the number of exchanged messages, in which each node caches error filter for its sub-tree. They also introduced an algorithm, based on potential gains, to adaptively adjust the error thresholds.

With respect to the holistic aggregate queries that we focus on in this paper, there are not many related approaches. To generalize, there are three mainstream techniques for answering the holistic queries, which are centralized approaches, sampling approaches, and histogram-based approaches. TAG [21] presented a centralized approach for holistic in which sensor values are transmitted to the root where the aggregate result is calculated. Sampling is another type of technique for answering the holistic queries. SIA [27] proposed a framework for secure information aggregation and discussed the sampling-based computation of Median query. Histogram is a common structure for storing data distribution information in answering holistic queries. There are many types of histograms, such as equal-length histogram, equal-depth histogram and Wavelet-based histograms [24]. All histograms share the common point that they group values into buckets [11], so these methods are also regarded as

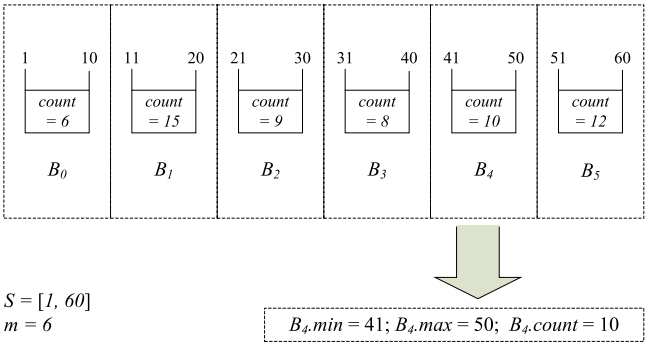


Fig. 1. An example of F-bucket.

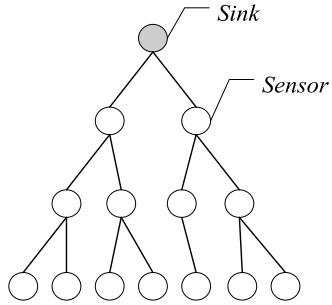


Fig. 2. Network topology.

buckets-based approaches. Hellerstein [14] introduced a wavelet-based aggregation scheme which can also produce results of increasing resolution over time. Q-digest [29] is another kind of buckets-based approach which allows overlapping buckets. Besides, Q-digest guarantees an error of $O(\log(\sigma)/m)$ with message size of m. Greenwald et al. [12] designed an online algorithm for computing ε-approximate quantile summary for large data stream with a worst-case space requirement of $O(log(\varepsilon N)/\varepsilon)$. Cormode et al. [4] proposed distributed-tracking schemes for finding accurate quantiles with error guarantees. They assumed each remote site maintains a local data stream. Greenwald and Khanna [13] presented algorithms which can provide ε-approximate answers for quantile with message size $O(log^2(n)/\varepsilon)$. Some other methods such as Manku etc. [22] presented hybrid approximate algorithms for computing frequency counts over data streams. ASP [15] presented tree-based space-efficient scheme for calculating the frequent items and quantile.

Other than query processing over sensor networks, there are many other interesting works in the literature [18], [33], such as energy efficient coverage problem in sensor networks [2], moving object tracking [8], event detection [19] and etc.

3 Exact Query Scheme

We assume that a *Median* query is executed upon a dataset S within the range [1, σ] and the total number of sensors in the network is n. The range here refers to integer values, the real value range can be converted into integer range by multiple a large enough integer. The output of Q is a data element $d \in [1, \sigma]$ such that the number of sensor values which are smaller (or larger) than d is $n/2$. A straightforward approach to get answers for Q is to retrieve all the values from sensors, however, as mentioned before, this is not energy efficient. Thus, in this paper, we propose a histogram-based approach to get exact answers for continuous queries. Specifically, we use the histogram summary structure to store the value distribution of the network. Each bucket in the histogram counts the number of values in a certain range. The exact query algorithm works when data values in sensor network are relatively stable over time. Since the sensor values are relatively stable and highly correlated among different rounds, the Median result can be searched in multiple rounds. In each round, we adjust the ranges of buckets, that is, subdividing the range where the median is located and assigning the subdivided small ranges to all intermediate buckets in the next round. After a few rounds, the ranges of all intermediate buckets will be of length 1, thus, we can get the exact median result continuously. When the median value runs out the ranges handled by these intermediate buckets, we need to adjust the range again. The range refining is conducted at the sink and the schema is flooded to the network. The detailed steps of this approach are discussed in following sections.

3.1 Data Structure

In the exact query approach, we apply the bucket histogram, which is referred as *F*(*lexible*)-*Bucket*, to get the exact answer for the query. The F-Bucket schema is assigned at the sink and then flooded to the whole network. One F-Bucket F consists of a list of triples which are denoted as buckets, $F = \{B_0, B_1, \ldots, B_i, \ldots B_{m-1}\}$ and $B_i = (min, max, count)$, where integer m denotes the maximum number of the buckets determined by the message size, each bucket B_i has a range $[B_i.min, B_i.max]$ defined by the *min* and *max*, and $B_i.count$ counts the number sensor values in this range. Note that the ranges associated to bucket B_i and B_{i+1} are consecutive, thus, $B_i.max + 1 = B_{i+1}.min$. F-Bucket is flexible because the ranges corresponding to buckets are not fixed and can be refined periodically according to recent query results. An example of the F-Bucket with six buckets is illustrated in Fig. 1.

In order to reduce the transmitted bytes, messages do not carry full F-Buckets but only buckets that are not empty ($count > 0$). We apply the in-network aggregation at intermediate nodes to merge all the partial F-Buckets received from children.

3.2 Query Processing

Although our query algorithm can work over arbitrary topology structures, to simplify the discussion we assume a tree-like routing topology [17], [19]. All sensor nodes in the network are organized into a spanning tree rooted by a special node called sink as illustrated in Fig. 2. Take the *Median* query as an example, each round the leaf node builds an F-Bucket of only one bucket with its own value and transmits to its parent. An intermediate node receives F-Buckets from children and merges them with its own F-Bucket to an integrated one. The intermediate node then sends the new F-Bucket to its parent as well. In the end of the round, the sink merges all the received F-Buckets to a final one and

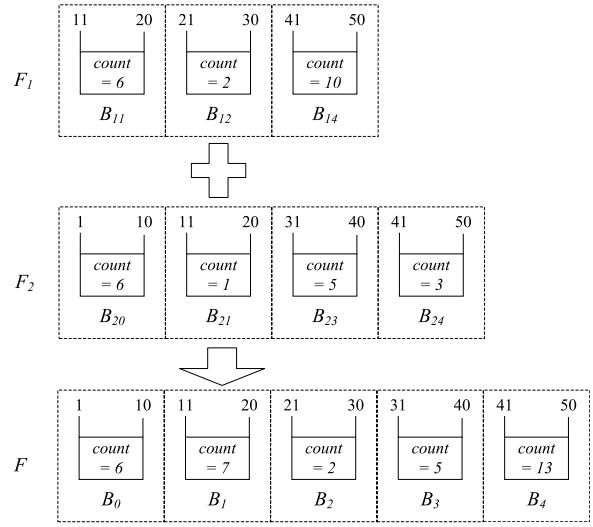


Fig. 3. Merge two F-buckets.

calculates the median result. During a continuous *Median* query, the above process repeats round by round. The range refining algorithms finds the range where the median value is located and subdivides it to all intermediate buckets. After a few rounds all the intermediate buckets handle a range of length 1 and the rest ranges are handled by two boundary buckets. The median value is located in the range of intermediate buckets and exact median results can be calculated continuously. When the median runs out the range of intermediate buckets, the range refining algorithm will adjust the ranges of buckets.

3.2.1 Intermediate Node Operation

The processing of an intermediate node is illustrated in Algorithm 1. Basically, the intermediate node collects the F-Buckets, merges them and sends the merged one to upper level. The merging of two F-Buckets is simple because the range of each bucket is fixed, we only need to merge their *count* of corresponding buckets that are associated to the same range. As shown in Fig. 3, F_1 and F_2 have three and four nonempty ($count > 0$) buckets respectively, in the merged F-Buckets F, there are five buckets and the *count*s in corresponding buckets of F_1, F_2 are summed. For example, the buckets in F_1 and F_2 with same range [11], [20] have *count* of 6 and 1 respectively, and then the *count* of bucket [11], [20] in F is 7 $(6 + 1)$.

Algorithm 1. Intermediate Node Processing

1: *Generate value v;* //start processing
2: *Build F-Bucket Fv;*
3: *Receive F-Buckets from children*;
4: **for all** *Fi in received F-Bucket list* **do**
5: *Fv = Merge (Fv, Fi);*
6: **end for**;
7: *Transmit Fv;* //transmit message to parent node;

3.2.2 Result Calculation

With the received F-Bucket in sink, we can answer a *Median* query. Algorithm 2 illustrates the steps to compute a *Median* result. According to Algorithm 2, the return value is a range that covers the query result. If the returned range is of length 1, we can get the exact median.

Algorithm 2. Calculating_Median (F, n)

1: $rank = n/2$;
2: $sum = 0$;
3: $i = 0$;
4: **while** $sum < rank$ **do**
5: $sum = sum + F.bucket[i].count$;
6: $i{+}{+}$;
7: **end while**
8: **Return** range of $F.bucket[i]$;

3.3 Range Refining

At the very beginning, all buckets are equal length; during the continuous query, the range assignment will be adjusted to fit the value distribution better. The refining process assigns more buckets to the range where the queried median is located and adjust this assignment while the value distribution changes. As the historical data accumulated at the sink, we have an anterior prediction about the real median value. Based on this prediction, we can adjust the range granularity associated to each bucket. The refining algorithm will assign a fine granularity monitoring to the range which covers the median value with high probability and a coarse granularity monitoring to the rest. Most buckets are assigned to the candidate range (called *focused window*) and each bucket handles a range of length 1 after a few rounds.

With the refining algorithms, we can guarantee the exact result for arbitrary query. The basic idea of a refining algorithm is multi-pass search. The continuous query rounds are divided to three different types. Considering median, in *initial rounds* we try to find a narrow range contains the median value. This is achieved by subdividing the current range where median is located. At last, most of buckets are assigned to monitor this range and each of them is corresponding to a value interval of length 1. Then in the following *continuous rounds*, querying will focus on this range (*focused window*) and extract the accurate median value. When the median value runs out this range due to value changing, one or more *refresh rounds* will rediscover the *focused window*.

1) *Initial rounds*. At the very beginning, *Initial rounds* are applied to detect the *focused window*, as shown in Fig. 4, assuming that the sensor values are among $S = [0, 174]$ and there are at most seven buckets in one F-Bucket.

In the first round, the monitoring range S is divided in to seven equal parts and each bucket counts the number of values that lie in its range. After this round, as shown in Fig. 4 the median value is located in bucket A. The range associated to A ([50, 74]) is subdivided in the second round and handle by bucket A_1 to A_5. The rest ranges are combined and handled by B ([0, 49]) and C ([75, 174]). After the second round, the range covering median shrinks to [60, 64] (A_3) which is narrow enough and regarded as the *focused window*. The process of finding the *focused window* may continue for several rounds and it will end in at most $log(\sigma)$ rounds. In the following *continuous rounds*, bucket A_1 to A_5 are

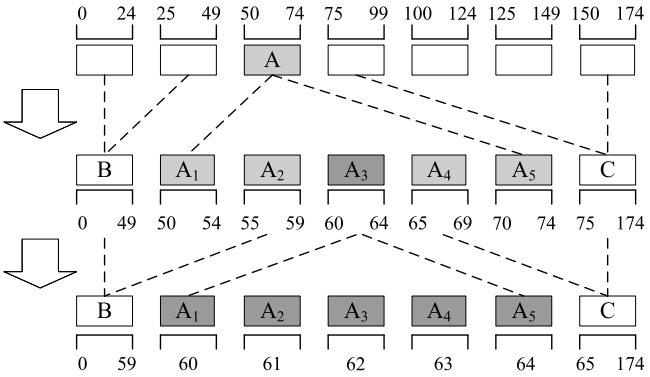


Fig. 4. Range refining in initial rounds.

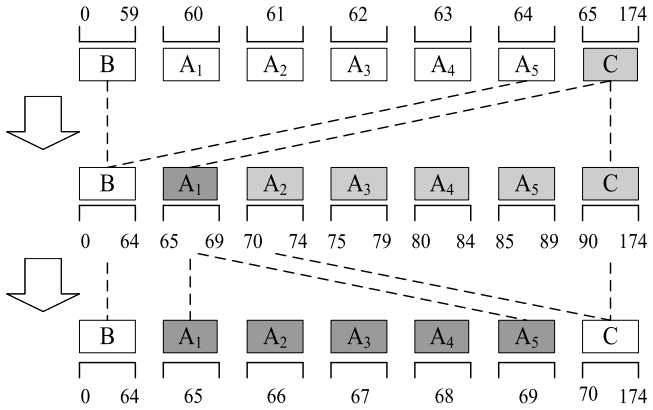


Fig. 6. Hierarchical refining algorithm.

assigned to the *focused window* and each bucket handles a range of length 1.

2) *Continuous rounds*. Each *Continuous round* can provide an accurate answer for the median query till the median value runs out the *focused window*. Process of each node in *continuous rounds* is similar to that in *initial rounds*.

3) *Refresh rounds*. When the median value runs out the *focused window* due to value changing, the *refresh round* will relocate the *focused window* to adapt the new data distribution. We propose two algorithms for refining the range assignment, *Slip refining* and *Hierarchical refining*.

Slip refining. This algorithm slips the *focused window* to the same direction which the median value moved towards. The slipping distance is equal to the length of *focused window*.

As shown in Fig. 5, if the median has run into the range of bucket *C* ([65, 174]), then in the *refresh round* the ranges handled by *B* and A_1 to A_6 are combined to a new range [0, 64] which is assigned to *B* and the *focused window* slips to [65, 69] and is assigned to A_1 to A_6 again. Bucket *C* charges the rest range. This process will continue till the *focused window* catches up the median again.

Hierarchical refining. The second refining algorithm is Hierarchical refining. In the Slip refining, we guess that the median lies in the range adjacent to current *focused window* and slip the window to the adjacent range. However, if the median changes fast, we need multiple rounds to catch up with it. In the Hierarchical refining, we first use an additional round to relocate the rough range covering median. Then slip the *focused window* to it. A running example is illustrated in Fig. 6. In this example, the median value runs out the current *focused window* [60, 64] to range [65, 174] handled by bucket *C*. The range of bucket *C* is subdivided and assigned to buckets A_1 to *C* and the current *focused window* is merged to bucket *B*. Buckets A_1 to A_5 each handles a range with equal length to the prior *focused window* which is 5 in this example. Since we assume that the data changing is slow, median value will be located in A_1 to A_5 with high probability. As shown in Fig. 6, the *focused window* is relocated to range [65, 69] and new *continuous rounds* begin.

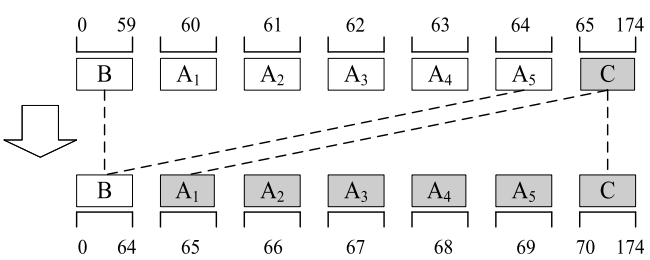


Fig. 5. Slip refining algorithm.

4 Wavelet-Like Approximate Query Algorithm

When sensor values change quickly, the range refining process will be called frequently which leads to very high communication cost. In fact, under such a situation, the temporal correlation over time is weak and it is not appropriate to use F-Bucket any more. Thus, we propose a wavelet-like approximate algorithm to get approximate answers to the query. In this algorithm, queries in different rounds are independent, so the data changing over time will not affect the query processing. We still use the histogram summary to store the information. Different from the exact query, each bucket in this structure stores the average of a group of values and buckets are sorted by their values, which is similar to the formulation of Haar wavelet [24]. Intermediate sensors combine data structures from downstream nodes, insert their own values and forward the structure to upper nodes. Finally, at the sink we get an aggregated data structure, with which we can run varying queries. For example, if we want the median value, we can choose the value in the middle position of this list. In the following sections we will show details of this algorithm as well as the error bound in the worst case.

4.1 Data Structure in Approximate Query

In the approximate query algorithm, the data structure is different from F-Bucket and we will refer to the approximate structure as AF-Bucket, where each AF-Bucket is represented as: $F' = \{[b_0, b_1, \ldots, b_i, \ldots, b_{m-1}], count(F'), loglen(F')\}$; $(b_i < b_{i+1})$.F' consists of m sorted values and an integer *count* which denotes the number of values rolled in to this structure. Variable *loglen* indicates the number of values rolled in one bucket $loglen = log(count/m)$. In this structure, b_i is the approximate value along with function $rank(b_i) = count \times (i/m)$. For example an AF-Bucket $F' = \{[3, 6, 10, 15], 8, 1\}, count(F') = 8$ means there are eight values in this structure and $loglen(F') = 1$ means each bucket represents $2^1 = 2$ values.

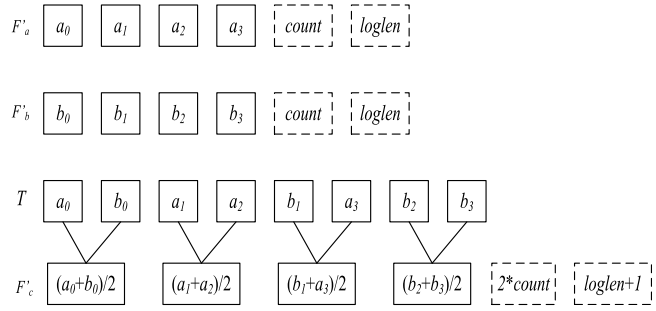


Fig. 7. Example of merging two AF-buckets.

Value 3 is the average of the zeroth and first values and 6 is the average of the second and third values, etc.

4.2 Query Processing

The query processing in approximate query has no range refining steps. Each round is an independent one-shot query. In each round, the leaf nodes construct new AF-Buckets and insert their values in. Then the AF-Buckets are delivered to their parents. In an intermediate node, the received AF-Buckets are merged together to be an integrated one. This merging process reduces transmissions at the cost of losing some information. Finally, the sink aggregates all received AF-Buckets to an integrated one and calculates the query results. The following sections shows details of the merging process in the intermediate nodes and the result calculation in the sink.

Algorithm 3. Intermediate Node Processing

1: *Generate value v;* //start processing
2: *Receive AF-Buckets* $\{F'_i\}$ *from children;*
3: *Insert v to* $\{F'_i\}$
4: ***while*** *exist* $F'_iF'_j$ *in* $\{F'_i\}$ *and* $loglen(F'_i) = loglen(F'_j)$ ***do***
5: *Merge* (F'_i, F'_j);
6: ***end while***
7: *Transmit the merged* $\{F'_i\}$; //transmit to parent;

4.2.1 Intermediate Node Operation

The processing in an intermediate node is illustrated in Algorithm 3. During the process of inserting the own value v to the AF-Bucket list, two cases have to be dealt with. If there is one AF-Bucket F' in the list with count less than m, we can insert v to the value list of F' and resort the list. If all AF-Buckets' count are no less than m, we construct a new AF-Bucket with only one value v and add it to AF-Bucket list. The merging process of two AF-Buckets $F'_a = \{[a_0, a_1, \ldots, a_m{-}1], count(F'_a), loglen(F'_a)\}$ and $F'_b = \{[b_0, b_1, \ldots, b_m{-}1], count(F'_b), loglen(F'_b)\}$ is discussed as follows:

1) If $count(F'_a) = count(F'_b) > m$ (in other words $loglen(F'_a) = loglen(F'_b) > 0$), we put all values in F'_a and F'_b into a double *size* (2*m) temp list T and these values are resorted while being inserted to T. Then, the average of every two values in list T is calculated to construct the merged AF-Bucket F'_c. Finally, double the variable count and have the *loglen* plus one. This process is shown in Algorithm 4. Fig. 7 shows an example of the merging of two AF-Buckets. In this example, F'_a and F'_b are two input AF-Buckets and F'_c is the merged AF-Bucket. As described in Algorithm 4, the values in F'_a and F'_b are put into the temp list T. The values in T are sorted. As shown in Fig. 7, in this example we assume that the sorted list T is $[a_0, b_0, a_1, a_2, b_1, a_3, b_2, b_3]$. Then the average of every pair of values is used to construct a new AF-Bucket F'_c with the *count* doubled and *loglen* equals to one plus that in F'_a (F'_b).

2) If $count(F'_a) < mandcount(F'_b) < m$, the merging is simpler. Values in F'_a and F'_b are put together to a temp array and resorted. Then the first m values are selected to store in F'_a and the rest values are put to F'_b. If $count(F'_a) + count(F'_b) \leq m$, then F'_b is empty and can be removed from the AF-Bucket list.

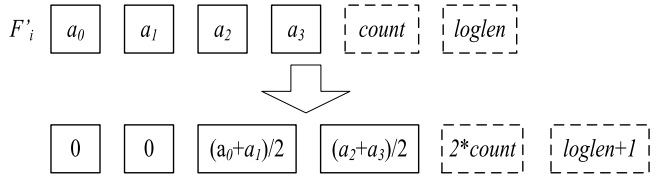


Fig. 8. Example of zero-padding.

Algorithm 4. Merge(F'_a, F'_b)

1: *indexA = 0;*
2: *indexB = 0;*
3: **for** *i* **from** 0 **to** 2×*m* - 1 **do**
4: **if** $indexB >= m$ **or** ($indexA < m$ **and** F'_a[*indexA*] < F'_b[*indexB*])
5: *T*[*i*] = F'_a[*indexA*];
6: *indexA* ++;
7: **else**
8: *T[i]* = F'_b *[indexB];*
9: *indexB* ++;
10: **end if**
11: **end for**
12: **for** *j* **from** 0 **to** *m*-1 **do**
13: $F'_c[i] = (T[2i] + T[2i+1])/2$;
14: **end for**
15: $count(F'_c) = 2 \times count(F'_a)$;
16: $loglen(F'_c) = loglen(F'_a) + 1$;
17: *Return* F'_c;

Note that after the merging process, there may be more than one AF-Bucket in the list with different *loglen*, because only the AF-Buckets with same *loglen* can be merged. All of them are transmitted to upstream nodes and the number of AF-Buckets is less than $log(n/m)$.

4.2.2 Result Calculation

After the sink receives AF-Buckets, it merges them in the same way as discussed above. If there is still more than one AF-Bucket in the list that cannot be merged, the zero-padding merging algorithm is applied. If the *loglen* (*count*) of AF-Bucket F'_i is smaller than that of F'_{i+1}, we pad zeros to F'_i till F'_i and F'_{i+1} are equal length. Fig. 8 shows an example of double an AF-Bucket by zero-padding. By this way, all the AF-Buckets are merged pair after pair. At last we get an integrated AF-Bucket and calculate the query result with it. For example in the Median query, we just remove the prior

zeros in the final value list and use the value in the middle position of the remaining values as the approximation of the real median value in sensor network.

4.3 Error Bound

Merging of two AF-Buckets will lose information and cause error to the query results, take the *Median* query as an example, the relative percentage error is defined as

$$\text{err} = \frac{realRank(c) - n/2}{n}. \tag{1}$$

Though merging introduces errors, we can show that the error is bounded in the worst case, as stated in the following Theorem.

Theorem 1. *The error introduced by merging of two AF-Buckets is within* O(log(n)/m).

Proof. First we consider the case of merging of two AF-Buckets F'_a and F'_b with $count = m$ and thus $loglen(count/m) = 0$. The merged result is denoted as F'_c. Note that in this situation value a_j and b_k have the exact rank. □

If $c_i = (a_j + b_k)/2, a_j < b_k$, minimum rank of c_i is $lowrank(c_i) = rank(a_j) + rank(b_{k-1})$ and the maximum rank of c_i is $highrank(c_i) = rank(a_{j+1}) + rank(b_k)$. So the maximum rank error of c_i is:

$$\begin{aligned}\text{errorRank}(c_i) = \text{rank}(a_{j+1}) - \text{rank}(a_j) + \text{rank}(b_k) \\ - \text{rank}(b_{k-1}) = 2\end{aligned} \tag{2}$$

Note that if $k = 0$, maximum rank error of c_i will be less, so in the following discussion, we just consider the situation that $k > 0$. The situation of $c_i = (b_k + a_j)/2, a_j > b_k$ is the same to discussion above.

If $c_i = (a_j + a_{j+1})/2, a_j < a_{j+1}$, the minimum rank of average value c_i is $lowrank(c_i) = rank(a_j) + rank(b_k)$ in which b_k is the nearest value that smaller than a_j and the maximum rank of c_i is $highrank(c_i) = rank(a_{j+1}) + rank(b_{k+1})$ in which b_{k+1} is the nearest value bigger than a_{j+1}.

$$\begin{aligned}\text{errorRank}(c_i) = \text{rank}(a_{j+1}) - \text{rank}(a_j) + \text{rank}(b_{k+1}) \\ - \text{rank}(b_k) = 2\end{aligned} \tag{3}$$

At the end, the rank error of c_i is at most

$$\text{errorRank}(c_i) = 2. \tag{4}$$

Second, we consider the case of merging of two AF-Buckets A and B with $count = m \times 2^{loglen}$ and thus $loglen > 0$.

If $c_i = (a_j + b_k)/2, a_j < b_k$, minimum rank of c_i is $lowrank(c_i) = lowrank(a_j) + lowrank(b_{k-1})$ and the maximum rank of c_i is $highrank(c_i) = highrank(a_{j+1}) + highrank(b_k)$. So the maximum rank error of c_i is

$$\begin{aligned}\text{errorRank}(c_i) &= \text{highRank}(c_i) - lowRank(c_i) \\ &= highRank(a_{j+1}) - lowRank(a_j) \\ &\quad + highRank(b_k) - lowRank(b_{k-1}) \\ &= errorRank(a_{j+1}) + 2^{loglen} + errorRank(b_k) \\ &\quad + 2^{loglen}.\end{aligned} \tag{5}$$

Because AF-Buckets A and B have the same *count* and *loglen*, the rank error of a_i and b_i are the same, then

$$\text{errorRank}(c_i) = 2 \times (\text{errorRank}(a_j) + 2^{loglen(A)}). \tag{6}$$

Similarly we can prove that in other cases, the maximum rank error is the same to function (6).

Then, according to the epagoge

$$\begin{aligned}&\text{loglen} = 1 : \text{errorRank}(C0) = 2^1 \\ &\text{loglen} = 2 : \text{errorRank}(C1) \\ &\qquad = 2 \times (\text{errorRank}(C0) + 2^1) = 2 \times 2^2 \\ &\text{Assume :} \\ &\text{loglen} = i : \text{errorRank}(Ci) = i \times 2^i \\ &\text{then :} \\ &\text{loglen} = i + 1 : \text{errorRank}(Ci + 1) \\ &\qquad = 2 \times (i \times 2^i + 2^i) = (i + 1) \times 2^{i+1}.\end{aligned} \tag{7}$$

In function (7), *errorRank*(*Ci*) denotes the maximum rank errors in an AF-Buckets with *loglen i*. If there are n nodes in the sensor network, the maximum value of *loglen* in the final AF-Bucket is $log(n/m)$, so the maximum percentage rank error is

$$\frac{\log(\frac{n}{m}) \times \frac{n}{m}}{n} < \frac{\log(\frac{n}{m})}{m}. \tag{8}$$

Finally, during the result calculating, we use the zero-padding algorithm which may bring some errors in to query results. However, since our zero-padding and merging are from the AF-Bucket with smaller *count* to the AF-Bucket with bigger *count* pair after pair, so the biggest AF-Bucket does not need to pad zeros. Then the *count* in final integrated AF-Bucket is at most $2n$ and the maximum percentage rank error is $log(2n/m)/m$. Thus the error bound is $O(log(n)/m)$.

5 Hybrid Approach

According to prior discussion, it is clear that when the data changing rate is low, the exact query scheme achieves very high efficiency. However, when sensor values change quickly, the range refining process will be called frequently which leads to very high communication cost. In this case, the approximate approach becomes more stable to answer the queries. However, how to adaptively select the appropriate scheme is non-trivial. To fully leverage the advantages of both methods, in this work, we propose a novel metric named *efficiency* which models how quickly the sensor value changes and its effect on query processing. Using this metric, our hybrid scheme adaptively applies different query algorithms under different circumstances during the query processing. The definition of *efficiency* metric will be described in detail in the following paragraph.

At the very beginning of that query, we use the exact query algorithm. Due to the data changing, the exact answer can not be obtained each round. The median will run out the *focused window* in some rounds and some other rounds are used to refine the range assignment. We denote all these

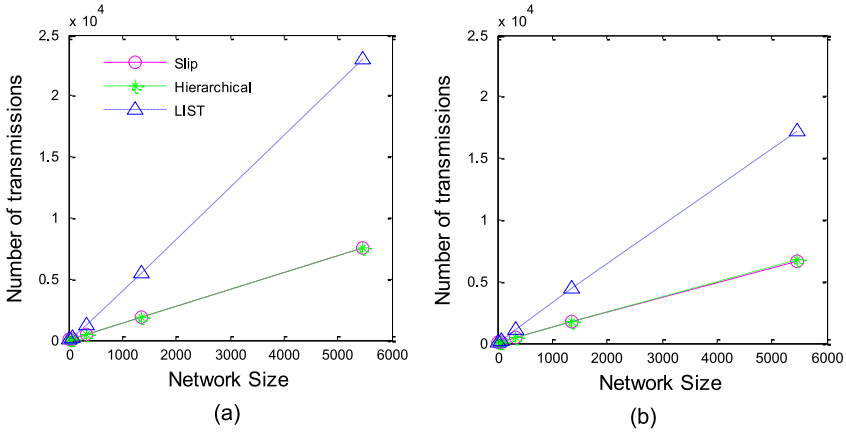


Fig. 9. (a) Traffic on random data. (b) Traffic on correlated data.

rounds as *no-answer* round. The rounds that can provide an exact result as exact answer rounds. Then the ratio between the numbers of these two types of rounds specifies the *efficiency* of current query, *efficiency* = *number*(*no-answer*) / *number*(*answer*). When the *efficiency* parameter rise above a certain user specified threshold, it will be switched to the approximate algorithm automatically. In the experiment of this paper, the *efficiency* parameter is set to 0.5 which indicates that there is at most one no-answer round per two exact answer rounds.

Our approach can also switch to the exact query algorithm when the data changing is slow. The switching parameter dependens on the *efficiency* parameter. Assume that when the median runs out the *focused window*, one additional round is used to refine the range assignment. Thus there are two *no-answer* rounds each time when the *median* run out *focused window*. So the number of exact answer rounds between two refinements is at least 2/*efficiency*. In other words, the median values will stay in the *focused window* for at least 2/*efficiency* rounds, and then the change of the median value is less than $\frac{(m-2)\times efficiency}{2}$ (m denotes the number of buckets in exact query and then m-2 denotes the width of *focused window*) per round. When applying the approximate algorithm, we cache the approximate results of recent several rounds. When the results' change is less than $\frac{(m-2)\times efficiency}{2}$ per round, we can switch to the exact query algorithm.

6 Evaluation

In this section, we evaluate our hybrid algorithms for *Median* query in varying conditions. We also compare our algorithm with the LIST and Q-digest [29] approach. LIST is a simple un-aggregated data summarization scheme in which the summary is a list of distinct sensor values and a count for each value. Intermediate nodes receive summaries from its children and then form a list of all distinct values with their counts in the sub-tree. All distinct values and their counts are delivered to sink where quantile or other queries are answered. With this scheme we can derive exact answers for queries at the cost of high transmission traffic. We did not compare with [4] because the approaches developed in [4] mainly focused on distributed stream environment, not specialized for a wireless sensor network. The simulated sensors are organized on a balanced routing tree. Each parent nodes have 4 children. So the total number of sensors in this network can be 5, 21, 85, 341, 1365, 5461, etc. The monitoring data range is [1, 2^8]. Here we assume the wireless communication is ideal and there is no link loss. Thus we focus on the query algorithms. In each experiment, we run the continuous median queries for 500 rounds and calculate the average performance. Here we use the metric *number of transmissions* to evaluate the traffic cost of all these approaches which is defined as the total size of transmitted data packets in one round.

6.1 Experiment One

Firstly, we consider the exact query algorithm. Since LIST and our Flexible Buckets algorithms both can provide the exact answer for median query, we just evaluate their traffic cost. Each round, the sensor values has 25 percent probability to plus one and 25 percent probability to minus one. They have 50 percent probability to stay the same as prior round. The number of sensor nodes in the network is varying from 21 to 5,461. The results are illustrated in Fig. 9 where 'Slip' and 'Hierarchical' denote two different refining algorithms. The two approaches are tested on two different initial data distributions. Random data denotes that the sensor values are randomly generated in the beginning and Correlated data means the sensor values are correlated to each other. From Figs. 9a and 9b, we can find that our exact query algorithm transmits much less bytes than the LIST scheme. Moreover, the effects of two refining algorithms are similar on both data sets. This is because the data changing speed is slow, however, when data changes quickly, the performance of two refining algorithms will be different, which will be shown in the third experiment.

6.2 Experiment Two

Second, we evaluate the performance of the approximate algorithm (denoted as Wavelet-like). Precision and traffic cost are considered in our experiments. For a *Median* query, the relative percentage error is defined as $err = \frac{|\text{realrank(c)} - n/2|}{n}$. In the first test, the number of values in an AF-Bucket structure varies from 6 to 32. The network size is fixed on 1,365 sensor nodes. This test is conducted over two types of data sources, random date and correlated data. The results are reported in Fig. 10. In Fig. 10, the percentage error of median query

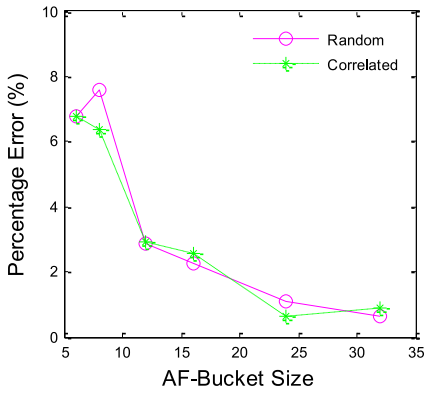


Fig. 10. Percentage errors.

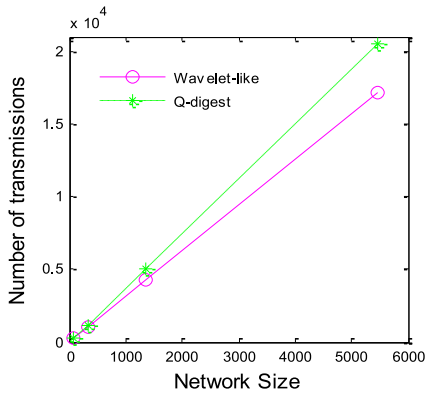


Fig. 12. Traffic on varying networks.

decreases from 7 percent to less than 2 percent as the AF-Bucket size increasing from 6 to 32. In other words, the performance will be better with more traffic. When the AF-Bucket size is 12, the accuracy of median query is already higher than 97 percent on both random and correlated data. Then the next experiment is used to compare the traffic cost of our approximate approach (Wavelet-like) and LIST. The initial sensor values are generated randomly. The AF-Bucket size is set to 12 with which the percentage error is lower than 3 percent as illustrated in Fig. 10. Fig. 11 shows the maximum message size on different network size. Compared with the LIST approach, the maximum message size of our scheme is much less. Thus, the traffic load of sensor nodes in our approach is more balanced. Q-digest [29] is an important approximate approach on holistic queries. We compare our approximate algorithm with Q-digest. In this test, the percentage errors of both algorithms are set around 2 percent with appropriate parameter configuration. Fig. 12 reports the transmission cost of the two algorithms, where our Wavelet-like approximate algorithm achieves the same precision with much less communication cost.

6.3 Experiment Three

In this experiment, we evaluate the performance of the hybrid algorithm. In this simulation we assume that the data changes to the same direction (worst case for exact query algorithm) and each node has 50 percent probability to change its value and 50 percent probability to stay the same. The *data changing rate* denotes the maximum value change of sensors in one round. For example when the changing rate is 10 per round, then each sensor value has 50 percent probability to change by [1, 10]. In this experiment, the *efficiency* parameter (in section 5) is set to 0.5. During the experiment, the *data changing rate* is varying from 5 to 30 per round.

The performances of exact query algorithms and approximate algorithm are shown in Fig. 13. When the *data changing rate* is less than 10 per round, the exact query algorithms Slip and Hierarchical provide exact query answers with little communication cost. As the changing rate increasing, the range refining becomes more and more frequently and significantly increasing the communication cost. When the changing rate is above 15 per round, the approximate algorithm performs better than the above two methods. According to Fig. 13, the Slip refining algorithm performs better than Hierarchical method. It is because that the Hierarchical refining needs an additional relocating round. So in the following test of the hybrid approach, we adaptively combine the Slip algorithm and Wavelet-like approximate algorithm. In the last test, we compare our approach with the Q-digest scheme. Both Q-digest and our hybrid approach achieve the percentage error around 2 percent. Then their communication costs are shown in Fig. 14.

The experimental results show that the hybrid approach works better in varying conditions. The communication cost of our approach is much lower than Q-digest for varying data changing rate, especially when the sensor values change slowly, our hybrid method can reduce the traffic cost more than a half. As discussed in Section 5, our hybrid approach adaptively applies different algorithm under

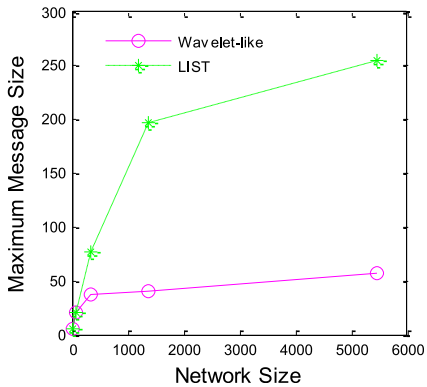


Fig. 11. Maximum message size.

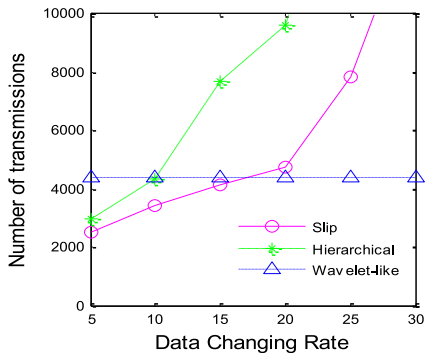


Fig. 13. Traffic on changing data.

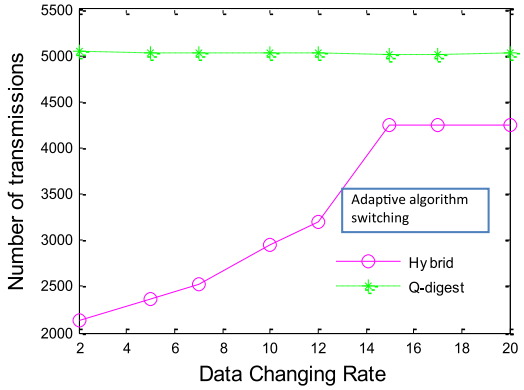


Fig. 14. Traffic on changing data.

varying circumstances. As illustrated in Fig. 14, the hybrid method selects the exact query scheme (Slip) when the data changing rate is low which saves more than a half traffic cost than Q-digest. When the dynamics of sensory readings become high, in this experiment, more than 15 per round, our hybrid approach automatically switches to the approximate algorithm.

7 Conclusions

In this paper, we tackle one type of popular queries, continuous holistic query, over sensor network. Compared to the counterpart of this type of query, non-holistic query, not much work has been done. However, holistic query is indeed important for many sensor network applications to collect statistical data. To avoid sending all the sensing data back to the sink, we propose two approaches to monitor continuous holistic queries, an exact one, Flexible Bucket (F-Bucket), to answer queries accurately and a wavelet-like approximate one to obtain the results with small error. Moreover, we present a hybrid approach based on the exact and approximation solutions, which applies the exact algorithm when the data changing rate is low and uses the approximation one when the rate becomes high. Experimental results show that the hybrid approach can achieve the similar accuracy but with much less traffic cost compared to the other approximate methods.

In this paper, we take one typical query *Median* as an example to illustrate our idea. In fact, the proposed methods can be naturally extended to solve other holistic queries. For example, all *Quantile* queries can be solved by our approach with different parameters. With the exact query scheme, we obtain varying quantiles by adjusting the position of *focused window* during range refining process. For the approximate scheme, different quantiles can be directly calculated with the data distribution in AF-Bucket. Generally, as both our exact and approximate schemes can return the data distribution of all sensor values using F-Bucket and AF-Bucket respectively, many other types of queries can be answered with the data distribution.

Acknowledgments

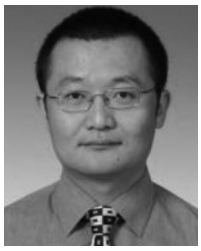
This research was supported in part by the NSFC under Grant No. 61472218 and the NSFC Distinguished Young Scholars Program under Grant No. 61125020.

References

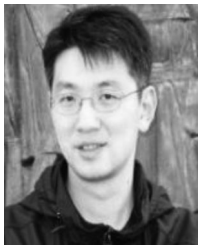
[1] R. Akbarinia, P. Valduriez, and G. Verger, "Efficient evaluation of SUM queries over probabilistic data," *IEEE Trans. Knowl. Data Eng.*, vol. 25, no. 4, pp. 764–775, Apr. 2013.
[2] M. Cardei and J. Wu, "Energy-efficient coverage problems in wireless ad-hoc sensor networks," *Comput. Commun.*, vol. 29, no. 4, pp. 413–420, 2006.
[3] C. M. Chen, Y. H. Lin, Y. C. Lin, and H. M. Sun, "RCDA: Recoverable concealed data aggregation for data integrity in wireless sensor networks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 23, no. 4, pp. 727–734, Apr. 2012.
[4] G. Cormode, M. Garofalakis, S. Muthukrishnan, and R. Rastogi, "Holistic aggregates in a networked world: Distributed tracking of approximate quantiles," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2005, pp. 25–36.
[5] J. Considine, F. Li, G. Kollios, and J. Byers, "Approximate aggregation techniques for sensor databases," in *Proc. Int. Conf. Data Eng.*, 2004, pp. 449–460.
[6] W. Choi and S. K. Das, "A novel framework for energy-conserving data gathering in wireless sensor networks," in *Proc. INFOCOM*, 2005, pp. 1985–1996.
[7] D. Chu, A. Deshpande, J. M. Hellerstein, and W. Hong, "Approximate data collection in sensor networks using probabilistic models," in *Proc. Int. Conf. Data Eng.*, 2006, p. 48.
[8] L. Chih-Yu, P. Wen-Chih, and T. Yu-Chee, "Efficient in-network moving object tracking in wireless sensor networks," *IEEE Trans. Mobile Comput.*, vol. 5, no. 8, pp. 1044–1056, Aug. 2006.
[9] A. Deshpande, C. Guestrin, S. Madden, J. M. Hellerstein, and W. Hong, "Model-driven data acquisition in sensor networks," in *Proc. Int. Conf. Very Large Databases*, 2004, pp. 588–599.
[10] A. Deligannakis, Y. Kotidis, and Y. Roussopoulos, "Hierarchical in-network data aggregation with quality guarantees," in *Proc. Int. Conf. Extending Database Technol.: Adv. Database Technol.*, 2004, pp. 577–578.
[11] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh, "Data Cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals," in *Proc. Int. Conf. Data Eng.*, 1996, pp. 29–53.
[12] M. B. Greenwald and S. Khanna, "Space-efficient online computation of quantile summaries," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2001, pp. 58–66.
[13] M. B. Greenwald and S. Khanna, "Power-conserving computation of order-statistics over sensor networks," in *Proc. ACM Symp. Principles Database Syst.*, 2004, pp. 275–285.
[14] J. M. Hellerstein, W. Hong, S. Madden, and K. Stanek, "Beyond average: Toward sophisticated sensing with queries," in *Proc. Int. Conf. Inf. Process. Sensor Netw.*, 2003, pp. 63–79.
[15] J. Hershberger, N. Shrivastava, S. Suri, and C. D. Toth, "Adaptive spatial partitioning for multidimensional data streams," in *Proc. 15th Annu. Int. Symp. Algorithms Comput.*, 2004, pp. 522–533.
[16] H. Jiang, J. Cheng, D. Wang, C. Wang, and G. Tan, "A general framework for efficient continuous multi-dimensional top-k query processing in sensor networks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 23, no. 9, pp. 1668–1680, Sep. 2012.
[17] Y. Kotidis, "Snapshot queries: Towards data-centric sensor networks," in *Proc. Int. Conf. Data Eng.*, 2005, pp. 131–142.
[18] M. Li and Y. Liu, "Iso-Map: Energy-efficient contour mapping in wireless sensor networks," *IEEE Trans. Knowl. Data Eng.*, vol. 22, no. 5, pp. 699–710, May 2010.
[19] M. Li, Y. Liu, and L. Chen, "Non-threshold based event detection for 3D environment monitoring in sensor networks," *IEEE Trans. Knowl. Data Eng.*, vol. 20, no. 12, pp. 1699–1711, Dec. 2008.
[20] Y. Liu, Y. He, M. Li, J. Wang, K. Liu, and X. Y. Li, "Does wireless sensor network scale? A measurement study on GreenOrbs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 24, no. 10, pp. 1983–1993, Oct. 2013.
[21] S. Madden, M. J. Franklin, J. Hellerstein, and W. Hong, "TAG: A Tiny AGgregation service for ad-hoc sensor networks," in *Proc. 5th Symp. Operating Syst. Design Implementation*, 2002, pp. 131–146.
[22] G. Manku and R. Motwani, "Approximate frequency counts over data streams," in *Proc. 28th Int. Conf. Very Large Databases*, 2002, pp. 346–357.
[23] S. Madden, R. Szewczyk, M. J. Franklin, and D. Culler, "Supporting aggregate queries over ad-hoc wireless sensor networks," in *Proc. 4th IEEE Workshop Mobile Comput. Syst. Appl.*, 2002, pp. 49–58.

[24] Y. Matias, J. S. Vitter, and M. Wang, "Wavelet-based histograms for selectivity estimation," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 1998, pp. 448–459.
[25] C. Olston, B. T. Loo, and J. Widom, "Adaptive precision setting for cached approximate values," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2001, pp. 355–366.
[26] S. Papadopoulos, A. Kiayias, and D. Papadias, "Secure and efficient in-network processing of exact SUM queries," in *Proc. 27th Int. Conf. Data Eng.*, 2011, pp. 517–528.
[27] B. Przydatek, D. Song, and A. Perrig, "SIA: Secure information aggregation in sensor networks," in *Proc. 1st ACM Conf. Embedded Netw. Sensor Syst.*, 2003, pp. 255–265.
[28] Q. Ren, L. Guo, J. Zhu, M. Ren, and J. Zhu, "Distributed aggregation algorithms for mobile sensor networks with group mobility model," *Tsinghua Sci. Technol.*, vol. 17, no. 5, pp. 512–520, Oct. 2012.
[29] N. Shrivastava, C. Buragohain, D. Agrawal, and S. Suri, "Medians and beyond: New aggregation techniques for sensor networks," in *Proc. 2nd ACM Conf. Embedded Netw. Sensor Syst*, 2004, pp. 239–249.
[30] M. A. Sharaf, J. Beaver, A. Labrinidis, and P. K. Chrysanthis, "TiNA: A scheme for temporal coherency-aware in-network aggregation," in *Proc. 3rd ACM Int. Workshop Data Eng. Wireless Mobile Access*, 2003, pp. 69–76.
[31] H. O. Tan, I. Korpeoglu, and I. Stojmenovic, "Computing localized power-efficient data aggregation trees for sensor networks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 3, pp. 489–500, Mar. 2011.
[32] C. Wang, C. Jiang, S. Tang, and X. Y. Li, "SelectCast: Scalable data aggregation scheme in wireless sensor networks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 23, no. 10, pp. 1958–1969, Oct. 2012.
[33] K. Xing, F. Liu, X. Cheng, and D. H. C. Du, "Real-time detection of clone attacks in wireless sensor networks," in *Proc. 28th IEEE Int. Conf. Distrib. Comput. Syst.*, 2008, pp. 3–10.
[34] Y. Yao and J. Gehrke, "The cougar approach to in-network query processing in sensor networks," *ACM SIGMOD Rec.*, vol. 31, no. 3, pp. 9–18, 2002.
[35] M. Ye, K. C. K. Lee, W. C. Lee, X. Liu, and M. C. Chen, "Querying uncertain minimum in wireless sensor networks," *IEEE Trans. Knowl. Data Eng.*, vol. 24, no. 12, pp. 2274–2287, Dec. 2012.
[36] M. Ye, W. C. Lee, D. L. Lee, and X. Liu, "Distributed processing of probabilistic top-k queries in wireless sensor networks," *IEEE Trans. Knowl. Data Eng.*, vol. 25, no. 1, pp. 76–91, Jan. 2013.
[37] Y. Yi, R. Li, F. Chen, A. X. Liu, and Y. Lin, "A digital watermarking approach to secure and precise range query processing in sensor networks," in *Proc. INFOCOM*, 2013, pp. 1950–1958.
[38] L. Yu, J. Li, S. Cheng, S. Xiong, and H. Shen, "Secure continuous aggregation in wireless sensor networks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 3, pp. 762–774, Mar. 2014.
[39] Y. Zhang, X. Lin, Y. Tao, W. Zhang, and H. Wang, "Efficient computation of range aggregates against uncertain location based queries," *IEEE Trans. Knowl. Data Eng.*, vol. 24, no. 7, pp. 1244–1258, Jul. 2012.

Kebin Liu received the BS degree in the Department of Computer Science from Tongji University, in 2004, and the MS and PhD degrees from Shanghai Jiaotong University, in 2007 and 2010. He is currently an assistant researcher in the School of Software and TNLIST, Tsinghua University. His research interests include sensor networks and distributed systems.



Lei Chen received the BS degree in computer science and engineering from Tianjin University, China, in 1994, the MA degree from the Asian Institute of Technology, Thailand, in 1997, and the PhD degree in computer science from the University of Waterloo, Canada, in 2005. He is currently an associate professor in the Department of Computer Science and Engineering, Hong Kong University of Science and Technology. His research interests include multimedia database, sensor databases, peer-to-peer databases, and probabilistic databases.



Yunhao Liu received the BS degree from Automation Department, Tsinghua University, China, in 1995, and the MA degree from Beijing Foreign Studies University, China, in 1997, and the MS and PhD degree in computer science and engineering from Michigan State University in 2003 and 2004, respectively. He is currently a professor in School of Software and TNLIST, Tsinghua University. His research interests include sensor networks, security, and high-speed network.

Wei Gong received the BS degree from the Department of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China, in 2003, and the MS and PhD degrees from the School of Software and the Department of Computer Science and Technology, Tsinghua University, in 2007 and 2012, respectively. His research interests include wireless sensor networks, RFID applications, and mobile computing.

Amiya Nayak received the BMath degree in computer science and combinatorics and optimization from the University of Waterloo, Ontario, Canada, in 1981, and the PhD degree in systems and computer engineering from Carleton University, Ontario, Canada, in 1991. Currently, he is a full professor at the School of Electrical Engineering and Computer Science, University of Ottawa. His research interests include the area of fault tolerance, distributed systems/algorithms, and mobile ad hoc networks with more than 150 publications in referred journals and conference proceedings. He is a senior member of the IEEE.