Approximate Aggregation Techniques for Sensor Databases

Jeffrey Considine, Feifei Li, George Kollios, and John Byers
Computer Science Dept., Boston University
{jconsidi, lifeifei, gkollios, byers}@cs.bu.edu

Abstract

In the emerging area of sensor-based systems, a significant challenge is to develop scalable, fault-tolerant methods to extract useful information from the data the sensors collect. An approach to this data management problem is the use of sensor database systems, exemplified by TinyDB and Cougar, which allow users to perform aggregation queries such as MIN, COUNT and AVG on a sensor network. Due to power and range constraints, centralized approaches are generally impractical, so most systems use in-network aggregation to reduce network traffic. Also, aggregation strategies must provide fault-tolerance to address the issues of packet loss and node failures inherent in such a system. An unfortunate consequence of standard methods is that they typically introduce duplicate values, which must be accounted for to compute aggregates correctly. Another consequence of loss in the network is that exact aggregation is not possible in general. With this in mind, we investigate the use of approximate in-network aggregation using small sketches. Our contributions are as follows: 1) we generalize well known duplicate-insensitive sketches for approximating COUNT to handle SUM (and by extension, AVG and other aggregates), 2) we present and analyze methods for using sketches to produce accurate results with low communication and computation overhead (even on low-powered CPUs with little storage and no floating point operations), and 3) we present an extensive experimental validation of our methods.

1 Introduction

Recent advances in micro-electronics and wireless technology are enabling the creation of small, cheap, and smart sensors. A smart sensor is a device with measurement, communication and computation capabilities, typically powered by a small battery. Individually, these sensors have limited capabilities, but there has been a great deal of research in developing *sensor networks* composed of a large numbers of these sensors. As a whole, sensor networks can be much more powerful and are being applied experimentally in a wide variety of areas — some sample applications include environmental monitoring, surveillance, and traffic monitoring. We focus on challenges posed by the limited capabilities of sensors, in particular, low ranges, limited bandwidth, and high losses, though at times we are constrained by other factors such as limited storage.

Motivation. Each sensor can produce a stream of data about its surroundings. Collectively, the sensors can produce a large amount of data, of which applications are usually only interested in aggregates (specific readings may be noisy or even unavailable). To reduce the cost and complexity of programming sensor networks to extract useful aggregates, a database approach has been advocated [13, 16]. Within this approach, special attention has been given to efficient query processing for aggregate queries [13, 16, 18].

For example, in the TAG system [13], users connect to the sensor network using a workstation or base station directly connected to a sensor designated as the root node. Aggregate queries over the sensor data are formulated using a simple SQL-like language, and then distributed across the network, e.g. by smart flooding. As the query is distributed across the network, a spanning tree is formed for the sensors to return data back to the root node. At each node in the tree, the sensor combines its own values with the data received from its children, and sends the aggregate to its parent. If there are no failures, this technique works extremely well for decomposable aggregates, namely distributive and algebraic aggregates [10] such as MIN, MAX, COUNT and AVG.

However, this technique breaks down when failures are introduced into the system. In sensor networks, both node and link failures are common. Node failures are expected to be relatively frequent, since the sensors are meant to be small, cheap, and mass-produced, and they will be placed in a variety of uncontrolled environments. Link failures (and packet losses) are also expected to be high due to environmental interference, packet collisions, and low signal-to-noise ratios [18]. Returning to the previous aggregation technique, if a node fails or its message does not reach its parent, the values associated with the entire subtree are lost. If the failure occurs close to the root node, then the effect on the resulting aggregate can be significant. Some approaches to address this problem based upon multi-path routing were proposed in [13], but they were primarily effective for duplicate insensitive aggregates such as MIN and MAX. For duplicate sensitive aggregates such as COUNT or AVG, the results are not satisfactory, and degrade rapidly as the number of sensors increases.

Contributions. In this paper, we propose a robust and scalable method for computing duplicate sensitive aggregates. Since exact solutions are generally impractical to guarantee in the face of losses, we provide an approximate solution which is robust against both link and node failures. Our contributions can be summarized as follows:

- We extend well-known duplicate insensitive sketches [7] to handle SUM aggregates. Through analysis and experiments, we show that the new sketch provides accurate approximations.
- We present a method to combine duplicate insensitive sketches with multi-path routing techniques to produce highly accurate sketches with low communication and computation overhead.
- We provide an analysis of the expected performance of previous methods as well as our method.
- Finally, we present an extensive experimental evaluation of our proposed system which we compare with previous approaches.

The remainder of this paper proceeds as follows. Background material is covered in Section 2. Counting sketches, along with theory and new generalizations, are discussed in Section 3. A robust aggregation framework using these sketches are then presented in Section 4. We validate our methods experimentally in Section 5 and conclude in Section 6.

2 Background

We now briefly survey the related work of our methods. Sensors and their limitations are described in Section 2.1. Previous frameworks for processing aggregates are covered in 2.2. Finally, the sketches which we use to improve upon these frameworks are introduced in Section 2.3.

2.1 Smart Sensor Devices

Smart sensors are full fledged computer systems, with a CPU, main memory, operating system and a suit of sensors. They are powered by small batteries and their lifetime is primarily dependent on the extent to which battery power is conserved. The power consumption tends to be dominated by transmitting and receiving messages and most systems try to minimize the number of messages in order to save power. However, the communication between sensors is wireless and the packet loss rate between nodes can be high. For example, [18] reports on experiments in which more than 10% of the links suffered average loss rate greater than 50%. Another challenge is that links may be asymmetric, both in loss rates and even reachability, making common ACK-based protocols such as TCP difficult or impossible. These limitations motivate query evaluation methods in sensor networks that are fundamentally different from the traditional distributed query evaluation approaches. First, the query execution plan must be energy efficient and second, the process must be as robust as possible given the communication limitations in these networks.

2.2 In-network Aggregate Query Processing

A simple approach to evaluate an aggregation query is to retrieve the values from all sensors in the base station and compute the aggregate there. Although this approach is simple, the number of messages and the power consumption can be large. A better approach is to leverage the computational power of the sensor devices and compute aggregates in-network. Aggregates that can be computed in-network include all decomposable functions [18].

Definition 1 *A function f is decomposable, if it can be computed by another function g as follows:* $f(v_1, v_2, ..., v_n) = g(f(v_1, ..., v_k), f(v_{k+1}, ..., v_n))$.

Using decomposable functions, the value of the aggregate function can be computed for disjoint subsets, and these values can be used to compute the aggregate of the whole using the merging function g. Our discussion is based on the Tiny Aggregation (TAG) framework used in TinyDB [13]. However similar approaches are used to compute aggregates in other systems [16, 17, 18, 11].

In TAG, the in-network query evaluation has two phases, the *distribution* and the *collection* phase. During the distribution phase, the query is flooded in the network and organizes the nodes into an *Aggregation Tree*. The base station broadcasts the query using its sensor which will be the *root* of the tree. The query message has a counter that stores the level of each sensor node in the tree. When a sensor receives this message, it re-transmits it with the counter increased by one. In this way, each node is assigned to a specific level equal to the node's hop distance from the root. Also, each sensor chooses one of its neighbors with a smaller hop distance from the root to be its parent in the aggregation tree.

During the collection phase, each leaf node produces a single tuple and forwards this tuple to its parent. The non-leaf nodes receive the tuples of their children and combine these values. Then, they submit the new partial results to their own parents. This process runs continuously and after h steps, where h is the height of the aggregation tree, the total result will arrive at the root. In order to conserve energy, sensor nodes sleep as much as possible during each step where the processor and radio are idle. When a timer expires or an external event occurs, the device wakes and starts the processing and communication phases. At this point, it receives the messages from its children and then submits the new value(s) to its parent. After that, if no more processing is needed for that step, it enters again into the sleeping mode [14].

This approach works very well for ideal network conditions. To address packet losses and node failures, Madden at al. [13] proposed some methods to improve the performance of their system. One solution is to cache previous values and reuse them if newer ones are unavailable. Of course, the previous values may reflect losses at lower levels of the tree.

Another approach considered in [13] takes advantage of the fact that a node may select multiple parents from neighbors at a higher level. Using this approach, which we refer to as "fractional parents," the aggregate value is decomposed into fractions equal to the number of parents. Each fraction is then sent to a distinct parent instead of sending the whole value to a single parent. For example, given an aggregate sum of 15 and 2 parents, each parent would be sent the value 7.5. It is easy to demonstrate analytically that this approach does not improve the expected value of the estimate over the single parent approach; it only helps to reduce the variance of the estimated value at the root. Therefore, the problem of losing a significant fraction of the aggregate value due to network failures remains.

2.3 Counting Sketches

Counting sketches were introduced by Flajolet and Martin in [7] for the purpose of quickly estimating in one pass the number of distinct items in a database (or stream) while using only a small amount of space. Furthermore, these sketches are easily combined, so the number of distinct items in the equi-join of a set of tables can be estimated by sketching each table independently, and then combining the sketches to form the sketch of the equi-join. Since then, there has been much work developing counting sketches (e.g. [6, 9, 3, 8, 2]) and generalizations (e.g. [1]). Counting sketches are the first of a class of duplicate insensitive sketches that remove sensitivity from approximating duplicate sensitivfe aggregates.

It is well known that exact solutions to the distinct counting problem require $\Omega(n)$ space. As shown in [1], $\Theta(\log n)$ space is required to approximate the number of distinct items in a multi-set with n distinct items. The original counting sketches of [7] achieve this bound, though they assume a fixed hash function that appears random, so they are vulnerable to adversarial choices of inputs. We use these sketches since they are very small and accurate in practice, and describe them in detail in Section 3.

Given the strong assumptions about the hash function in [7], another sketching scheme was proposed in [1] which only used a linear hash function (within an appropriate $GF(2^p)$), but by choosing the hash function randomly, this scheme is provably robust to adversarial inputs. This sketching scheme requires about 50% more space in practice, but the linear hash functions need to be agreed upon beforehand (allowing adversarial inputs again), or they must be distributed with the query. Another recent technique [5] works similarly to [7] but only uses $O(\log \log n)$ space. These "loglog" sketches are also vulnerable to adversarial inputs, but have provably good behavior otherwise (again, assuming the hash function appears random). The authors claim that these sketches have comparable accuracy to those of [7] while using roughly a third of the space in practice.

We note that both of these sketching techniques can be adapted to sketching summations (the subject of Section 3), since they both can reduced to calculating the maximum of a set of geometrically distributed random variables. However, the straight-forward implementation of this approach involves both logarithms and exponentiation so it is unsuitable for sensor networks as they typically lack floating point hardware. We also note that this approach loses the theoretical elegance of [1], both in using only linear hash functions and maintaining provable robustness against adversarial inputs. However the general approach of Section 3 is applicable using loglog sketches so further space improvements are possible.

3 Sketch Theory

One of the core ideas behind our work is that duplicate insensitive sketches will allow us to leverage the robustness typically associated with multi-path routing. We now present some of the theory behind such sketches and extend it to handle more interesting aggregates. First, we present in Section 3.1 details of the counting sketches of [7] (FM sketches) along with necessary parts of the theory behind them. Then, we generalize these sketches to handle summations in Section 3.2, and show that they have almost exactly the same accuracy as FM sketches.

3.1 Counting Sketches

We now describe FM sketches for the distinct counting problem.

Definition 2 *Given a multi-set of items $M = \{x_1, x_2, x_3, \ldots\}$, the* distinct counting *problem is to compute $n \equiv |\text{distinct}(M)|$.*

Given a multi-set M, the FM sketch of M, denoted $S(M)$, is a bitmap of length k (the choice of k will be discussed shortly). The entries of $S(M)$, denoted $S(M)[0, \ldots, k-1]$, are initialized to zero and are set to one using a random binary hash function h applied to the elements of M. Formally,

$$S(M)[i] \equiv 1 \text{ iff } \exists x \in M \text{ s.t. } \min\{j \mid h(x,j) = 1\} = i.$$

By this definition, each item x is capable of setting a single bit in $S(M)$ to one – the minimum i for which $h(x,i) = 1$. This gives a simple serial implementation which is very fast in practice and requires two invocations of h per item on average.

Theorem 1 *An element x_i can be inserted into an FM sketch in $O(1)$ expected time.*

Algorithm 1 CountInsert(S,x)

```
1: i = 0;
2: while hash(x,i) = 0 do
3:     i = i + 1;
4: end while
5: S[i] = 1;
```

We now describe some interesting properties of the sketches observed in [7].

Property 1 *The FM sketch of the union of two multi-sets is the bit-wise OR of their FM sketches. That is,*

$$S(M_1 \cup M_2)[i] = (S(M_1)[i] \vee S(M_2)[i]).$$

Property 2 *$S(M)$ is entirely determined by the distinct items of M. Duplication and ordering do not affect $S(M)$.*

Property 1 allows each node to compute a sketch of locally held items and send the small sketch for aggregation elsewhere. Since aggregation (i.e. union operations) is cheap, it may be performed in the network without burdening any nodes. Property 2 allows the use of multi-path routing of the sketches for robustness without affecting the accuracy of the estimates. We expand upon these ideas further in Section 4. The next lemma provides key insight into the behavior of FM sketches and will be the basis of efficient implementations of summation sketches in the next section.

Lemma 1 *For $i < \log_2 n - 2\log_2 \log n$, $S(M)[i] = 1$ with probability $1 - O(ne^{-\log^2 n})$. For $i \geq \frac{3}{2}\log_2 n + \delta$, with $\delta \geq 0$, $S(M)[i] = 0$ with probability $1 - O\left(\frac{2^{-\delta}}{\sqrt{n}}\right)$.*

Proof: This lemma is proven in [7] and follows from basic balls and bins arguments. ■

The lemma implies that given an FM sketch of n distinct items of unbounded length, one expects an initial prefix of all ones and a suffix of all zeros, while only the setting of the bits around $S(M)[\log_2 n]$ exhibit much variation. This gives a bound on the number of bits k required for $S(M)$ in general; $k = \frac{3}{2}\log_2 n$ bits suffice to represent $S(M)$ with high probability. It also suggests that just considering the length of the prefix of all ones in this sketch can produce an estimate of n. Formally, let

$$R_n \equiv \min\{i \mid S(M)[i] = 0\}$$

when $S(M)$ is an FM sketch of n distinct items. That is, R_n is a random variable marking the location of the first zero in $S(M)$. In [7], a method to use R_n as an estimator for n is developed using the following theorems.

Theorem 2 *The expected value of R_n for FM sketches satisfies*

$$E(R_n) = \log_2(\varphi n) + P(\log_2 n) + o(1),$$

where the constant φ is approximately 0.775351 and $P(u)$ is a periodic and continuous function of u with period 1 and amplitude bounded by 10^{-5}.

Theorem 3 *The variance of R_n for FM sketches, denoted σ_n^2, satisfies*

$$\sigma_n^2 = \sigma_\infty^2 + Q(\log_2 n) + o(1),$$

where constant σ_∞^2 is approximately 1.12127 and $Q(u)$ is a periodic function with mean value 0 and period 1.

Thus, R_n can be used for an unbiased estimator of $\log_2 n$ if the small periodic term $P(\log_2 n)$ is ignored. A much greater concern is that the variance is slightly more than one, dwarfing $P(\log_2 n)$, and implying that estimates of n will often be off by a factor of two in either direction. To address this, methods for reducing the variance will be discussed in Section 3.3.

3.2 Summation Sketches

As our first theoretical contribution, we generalize approximate counting sketches to handle summations. Given a multi-set of items $M = \{x_1, x_2, x_3, \dots\}$ where $x_i = (k_i, c_i)$ and c_i is a non-negative integer, the *distinct summation* problem is to calculate

$$n \equiv \sum_{\text{distinct}((k_i,c_i)\in M)} c_i.$$

When c_i is restricted to one, this is exactly the distinct counting problem.

We note that for small values of c_i, one might simply count c_i different items based upon k_i and c_i, e.g. $(k_i, c_i, 1), \dots, (k_i, c_i, c_i)$, which we denote *sub-items* of (k_i, c_i)). Since this is merely c_i invocations of the counting insertion routine, the analysis for probabilistic counting applies. Thus, this approach is equally accurate and takes $O(c_i)$ expected time. While very practical for small c_i values (and trivially parallelizable in hardware), this approach does not scale well for large values of c. Therefore, we consider more scalable alternatives for handling large c_i values.

Algorithm 2 SummationInsert(S,x,c)

```
1:  d = pick_threshold(c);
2:  for i = 0, ... , d - 1 do
3:      S[i] = 1;
4:  end for
5:  a = pick_binomial(seed=(x, c), c, $1/2^d$);
6:  for i = 1, ... , a do
7:      j = d;
8:      while hash(x,c,i,j) = 0 do
9:          j = j + 1;
10:     end while
11:     S[j] = 1;
12: end for
```

The basic intuition beyond our more scalable approach is as follows. We intend to set the bits in the summation sketch *as if* we had performed c_i successive insertions to an FM sketch, but we will do so much more efficiently. The method proceeds in two steps: we first set a prefix of the summation sketch bits to all ones, and then set the remaining bits by randomly sampling from the distribution of settings that the FM sketch would have used to set those bits. Ultimately, the distribution of the settings of the bits in the summation sketch will bear a provably close resemblance to the distribution of the settings of the bits in equivalent FM sketch, and we then use the FM estimator to retrieve the value of the count.

We now describe the method in more detail. First, to set the prefix, we observe that it follows from Lemma 1, that the first

$$\delta_i = \lfloor \log_2 c_i - 2\log_2 \log c_i \rfloor$$

bits of a counting sketch are set to one with high probability after c_i insertions. So our first step in inserting (k_i, c_i) into the summation sketch is to set the first δ_i bits to one. This introduces a minimal amount of bias in the subsequent estimator — in the proof of Theorem 2 in [7], the authors prove that the case where the first δ_i bits are not all set to one only affects the expectation of R_n by $O(n^{-0.49})$. In practice, we could correct for this small bias, but we disregard it in our subsequent experiments with sensor databases.

The second step sets the remaining $k - \delta_i$ bits by drawing a setting at random from the distribution induced by the FM counting sketch setting those same bits. We achieve this by simulating the insertions of items that set bits δ_i and higher in the counting sketch. First, we say an insertion x_i *reaches* bit z of a counting sketch if and only if $\min\{j \mid h(x_i, j) = 1\} \geq z$. The distribution of the number of items reaching bit z is well-known for FM sketches. An item x_i reaches bit z if and only if $\forall_{0\leq j<z}(h(x_i, j) = 0)$, which occurs with probability $1/2^z$. So for a set of c_i insertions, the number of insertions reaching bit δ_i follows a binomial distribution with parameters c_i and $1/2^{\delta_i}$. This leads to the following process for setting bits $\delta_i, \delta_i + 1 \dots k$ (initialized to zero). First, draw a random sample y from $B(c_i, 1/2^{\delta_i})$, and consider each of these y insertions as having reached bit δ_i. Then use the FM coin-flipping process to explicitly set the remaining bits beyond δ_i.

The pseudo-code for this approach is shown in Algorithm 2, and the analysis of its running time is presented next.

Theorem 4 *An element $x_i = (k_i, c_i)$ can be inserted into a sum sketch in $O(\log^2 c_i)$ expected time.*

Proof Sketch: Let α_i denote the number of items chosen to reach δ_i. Setting the first δ_i bits takes $O(\delta_i)$ time and simulating the α_i insertions takes expected $O(\alpha_i)$ time. The total expected time to insert x_i is then $O(\delta_i + f(\alpha_i) + \alpha_i)$, where $f(\alpha_i)$ denotes the time to pick α_i. Thus, the time depends on both α_i and the method used to pick α_i. By construction,

$$E(\alpha_i) = c_i * 1/2^{\lfloor \log_2 c_i - 2\log_2 \log c_i \rfloor},$$

so

$$\log^2 c_i \leq E(\alpha_i) < 2\log^2 c_i.$$

Selecting an appropriate method for picking α_i requires more care. While there exist many efficient methods for generating numbers from a binomial distribution ([12] has a brief survey), these generally require floating point operations or considerable memory for pre-computed tables (linear in c_i). Since existing sensor motes often have neither, in Section 4.3.1 we describe a space-efficient method that uses no floating point operations, uses pre-computed tables of size $\mathrm{O}(c_i/\log^2 c_i)$, and runs in time $\mathrm{O}(\log^2 c_i)$. Combining these results give the stated time bound. ■

We note that for small c_i values, it may be faster to use a hybrid implementation combining the naive and scalable insertion functions. Especially for very low c_i values, the naive insertion function will be faster. This is safe as long the threshold for choosing the insertion function is globally agreed upon.

Theorem 5 *The expected value of R_n for sum sketches satisfies*

$$E(R_n) = \log_2(\varphi n) + P(\log_2 n) + o(1),$$

where φ and $P(u)$ are the same as in Theorem 2.

Proof: The proof of this theorem follows the proof of Theorem 2 since the sum insertion function approximates repeated use of the count insertion function. Let

$$c_{\max} = \max\{c_i \mid (k_i, c_i) \in M\}$$

and

$$\delta_{\max} = \lfloor \log_2 c_{\max} - \log_2 \log c_{\max} \rfloor.$$

By the insertion method, the bottom $\delta_{\max}$ bits of $S_\Sigma(M)$ are guaranteed to be set. By construction (and Property 1), the remaining bits are distributed identically to those of an FM sketch with n distinct items have inserted. Thus, R_n (and its distribution) are the same except for the cases when the FM sketch had one of the first $\delta_{\max}$ bits not set. By Lemma 1, these cases occur with probability $O(ne^{-\log^2 n})$, so the difference in the expectation is at most $(\log_2 n - \log_2 \log n) * O(ne^{-\log^2 n})$, which is bounded (loosely) by $O(1/n)$. Therefore, $E(R_n)$ for summation sketches is within $o(1)$ of that of FM sketches. ■

Theorem 6 *The variance of R_n for sum sketches, also denoted σ_n^2, satisfies*

$$\sigma_n^2 = \sigma_\infty^2 + Q(\log_2 n) + o(1),$$

where σ_∞^2 and $Q(u)$ are the same as in Theorem 3.

Proof: The proof of Theorem 3 is adapted in a similar fashion. ■

3.3 Improving Accuracy

To improve the variance and confidence of the estimator, FM sketches can use multiple bitmaps. That is, each item is inserted into each of m bitmaps (using different independent hash functions) to produce m R values, $R^{\langle 1\rangle}, \ldots, R^{\langle m\rangle}$. The estimate is then calculated as follows:

$$n \approx (m/\varphi)2^{\sum_i R^{\langle i\rangle}/m}.$$

This estimate is more accurate, with standard error $O(1/\sqrt{m})$, but comes at the cost of increased insertion times ($O(m)$). To avoid this overhead, an algorithm called *Probabilistic Counting with Stochastic Averaging*, or PCSA, was proposed in [7]. Instead of inserting each item into each of the m bitmaps, each item is hashed to one of them and only inserted into that one. Thus, each of the bitmaps summarizes approximately n/m items. While there is some variation in how many items are assigned to each bitmap, further analysis showed that the standard error of PCSA is roughly $0.78/\sqrt{m}$. Using PCSA, insertion takes $O(1)$ expected time.

PCSA can also be applied to summation sketches, but greater care must be applied when combining PCSA to summation sketches. The potential for imbalance is much larger with summation sketches - a single item can contribute an arbitrarily large fraction of n. Thus, unless there is some guarantee about the distribution of c_i values, we employ the following strategy. Each c_i value has the form

$$c_i = q_i m + r_i$$

for some integers q_i and r_i, with $0 \leq r_i < m$. We then add r_i distinct items once as in standard PCSA, and then add q_i to each bitmap independently. Thus, we preserve the balance necessary for the improved accuracy and its analysis, but at the cost of $O(m\log^2(c_i/m))$ for each insertion. We employ these PCSA optimizations in our experiments.

3.4 Time/Space/Accuracy Tradeoffs

In situations where computational resources are severely constrained, it may be desirable to reduce the cost of performing insertion operations with summation sketches. We now briefly mention some tradeoffs in the computational time at the cost of increased communication and decreased accuracy. While this is unlikely to be desirable in sensor networks, given the high power costs of communication relative to computation, it may be desirable in other settings where there are large numbers of items per node.

Suppose that the largest value being inserted is bounded by y^x. Insertions with the algorithm described already take $O(x^2 \log^2 y)$ time. We can instead use x different summation sketches, each corresponding to a different digit of the c_i's using radix y. To add a c_i value, each digit of c_i is inserted into the corresponding sketch, taking expected $O(x \log^2 y)$ time, and estimates are made by summing the counting sketch estimates with the appropriate weights. The accuracy of this approach is essentially the same, and the increase in space is bounded by a factor of x (the range of values each sketch is approximating is smaller now). The insertion time can be further dropped to $O(z \log^2 y)$ expected time without affecting space requirements, if only the z most significant non-zero digits are inserted, but the expected estimate may be too low by a factor of $(1 + (y - 1)/(y^z - 1))$.

We note that while it is tempting to try collapsing the sketches back into one to avoid the space overhead at this point, this is generally fallacious and breaks our simulation of multiple counting sketch insertions. We point out that the distribution from inserting a single item and then shifting the sketch by x bits (possibly filling in x bits of ones) is very different from the distribution of inserting 2^x bits. The former has one possible result, while the latter has many different possible results. Furthermore, the first approach will be biased towards high estimates since many underestimates are now ruled out.

4 Approximate Estimation of Duplicate Sensitive Aggregates

In this section, we show how to use duplicate insensitive sketches to build a robust, loss-resilient framework for aggregation. Section 4.1 describes our algorithm leveraging the broadcast nature of wireless communication between sensors and the sketching techniques discussed in Section 3. Section 4.2 then provides a simple analytic evaluation of the proposed methods for a restricted class of regular topologies. Section 4.3 discusses practical details of implementations on sensor motes.

4.1 Algorithm

In sensor networks, when a sensor sends a message, all nodes that reside inside its communication range can receive this message. Therefore, a node can broadcast a packet to all of its neighboring nodes at once. This fact allows us to cheaply apply the idea of multipath routing for fault tolerance while using duplicate insensitive sketches to avoid complications in aggregation.

In the rest of this section, we consider continuous queries, noting that our methods can also handle one-shot queries as a simple special case. Aggregate query computation proceeds in two phases. In the first phase, the continuous query is distributed through the sensor network using flooding. The goal of this step is to create a robust topology where each node is assigned a level and a list of parent nodes. Therefore the created graph is not a tree but a DAG. This construction will be reused for continuous computation of the query. In the second phase, the actual computation of the aggregate occurs.

Next we show the steps that are used to create the topology. Each message M is a tuple of two values: a unique node ID and the Level of the transmitting node. Each node maintains the set of nodes in its neighborhood, and distinguishes between parent nodes, children nodes and others. Initially, all nodes set their Level to infinity (∞). Upon receipt of a message, a node updates the list of parents and its value of Level (if appropriate) and broadcasts a new message if its Level has been reduced. If acknowledgments are used (as in TinyOS), then the node can subsequently update its children list as well.

The root node initiates this phase by sending the first message with Level zero. The rest of the nodes run the following algorithm:

Algorithm 3 BUILDTOPOLOGY

```
1: if (Message M received) then
2:    if (M.Level < Node.Level) then
3:       Node.Level = M.Level;
4:       Update ParentList;
5:       M.ID = Node.ID, M.Level = M.Level+1;
6:       Broadcast M;
7:    end if
8: end if
```

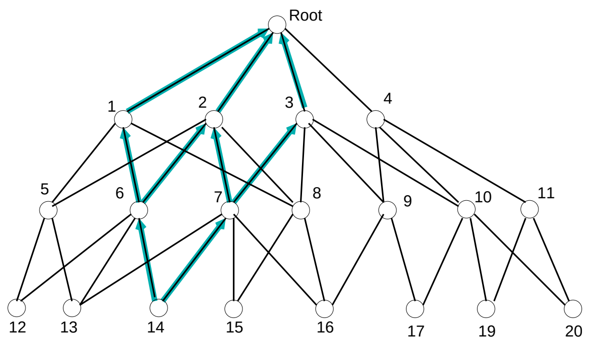


Figure 1. A robust routing topology with multiple paths to the root.

An example of a possible topology created with the above algorithm is shown in Figure 1. The advantage of the multiple parents approach is obvious. Most nodes in the network have multiple paths (of the same length) that can reach the root node, so an individual link or node loss has limited effects. For example, node 14 in Figure 1 has a number of different paths that can reach the root. So, if node 6 fails, the value of node 14 will reach the root via node 7. Another advantage of the DAG topology is that the topology need not be repaired every time a small number of nodes fail, unlike the spanning tree approach.

After the topology is discovered, the second phase starts. The nodes that participate in the query use their readings to create sketches and transmit these sketches to their parents. Parent nodes receive the sketches from their children and create a partial sketch that is also sent to their parents. Eventually, the root node will receive a set of final partial sketches that it will combine to estimate the aggregate value. This process is executed in each *epoch*. An epoch is a fixed time interval specified in the query and reflects how often a new aggregation should be produced [13]. We assume that the same protocol as in TinyDB is used to synchronize nodes. The epoch is divided in smaller time intervals and each node transmits only in one of those intervals. Parent nodes know when their children transmit; this is agreed during the topology construction.

A significant advantage of our approach is the fault tolerance inherent in broadcasting the sketch to multiple parents, which eliminates the need for packet-level retransmissions when individual packets are lost. Instead, in each epoch, each node simply broadcasts its sketch once. This approach can save considerable bandwidth and time spent identifying lost packets and scheduling retransmissions. In practice, if *no* parent acknowledges a given transmission, rebroadcast seems a reasonable option. Algorithm 4 shows the code that runs at each node.

In this algorithm we consider only two aggregates, COUNT and SUM (from which AVG can be computed directly). For the SUM query, the value can be different in each epoch; in that case the summation is over only the most recent reading. These techniques can also be extended to other aggregate functions beyond summation and counting. For example, the second moment can also be computed as an average of the squares of the items, as can any other non-centralized moment. The second moment can then be combined with the average of the items, to compute the variance and standard deviation too. We note that in computing the variance and standard deviation, the same counting sketch can be shared for the second moment and the average. Additionally, we note that the sketches described are easily generalized to handle other data types such as fixed point and signed numbers, and to a certain extent, products (summing logarithms) and floating point.

Algorithm 4 COMPUTEAGGREGATE

```
1: while Query is Running do
2:   InitializeSketch($S$);
3:   if Node is a parent then
4:     for each Child i=0, ..., k do
5:       Receive($S_i$);
6:     end for
7:     $S$ = Merge($S$, $S_1$, $S_2$, ..., $S_k$);
8:   end if
9:   if Node satisfies the query then
10:    if aggregate=COUNT then
11:      Count_insert($S$, NodeID);
12:    else if aggregate=SUM then
13:      Sum_insert($S$, NodeID, Value);
14:    end if
15:  end if
16:  Broadcast ($S$);
17: end while
```

4.2 Analysis

We now analyze the methods discussed so far for a restricted class of regular topologies. We consider the resilience of a single spanning tree in the presence of independent link failures, and compare the performance when multiple parents are leveraged. We specifically examine performance of the COUNT aggregate, counting all of the nodes in the network, but this analysis can be naturally extended to node and packet failures and other duplicate insensitive aggregates.

4.2.1 Fault Resilience of the Spanning Tree

First, we consider a baseline routing topology in which aggregates are computed across a single spanning tree, i.e. whereby each node transmits its aggregate to a unique parent. For simplicity, we assume that the degree of the tree is d, although our analysis can be extended for any other tree. We assume that link losses occur independently at random with probability p and that the height of the tree is h. In general, the probability of a value from a node at level i to reach the root is proportional to $(1-p)^i$.

We ignore the correleation between losses for simplicity here; however, more elaborate analysis that takes that into account is possible. The expected value of the COUNT aggregate is $E(count) = \sum_{i=0}^{h}(1-p)^i n_i$, where n_i is the number of nodes at level i. Assuming complete d-ary tree we have: $E(count) = \sum_{i=0}^{h}((1-p)d)^i = \frac{(d-pd)^{h+1}-1}{d-pd-1}$. For $h=10$, $d=3$ and $p=0.1$ (a 10% link loss rate) the expected fraction of the nodes that will be counted is poor, only 0.369. We also note that when the underlying routing topology is a spanning tree, node and link losses are equivalent, as there is a bijection between edges and nodes (loss of an edge is tantamount to loss of the node beneath that edge).

4.2.2 Fault Resilience of Multiple Paths

In order to analyze the use of multiple paths we now make a stronger assumption about the routing topology. Starting with the leaves at level 0, we assume that each node at level i has exactly d neighbors within its broadcast radius at level $i+1$, for all $0 \leq i \leq h-1$. From these neighbors, each node selects $k \leq d$ of these nodes as its parents, where k is a fault-resilience parameter, and it transmits its aggregate value to all k of these nodes. The benefit follows from the key observation that only one copy of the message initiated at a leaf must reach the root; alternatively, in order for a node *not* to be counted, *all* of the routes taken by its messages must traverse lossy links. While somewhat tighter bounds can be obtained, we will be satisfied with the following simple analysis, which suffices to provide close agreement with our experimental results. Let $\mathcal{E}_i$ denote the event that a copy of the leaf's value reached level i conditioned on the value having reached level $i-1$. With leaves at level 0, these events are well-defined for levels $1, 2 \ldots h$. Clearly $\Pr[\mathcal{E}_i] \geq (1-p^k)$ (this is a lower bound since only one copy is assumed to have reached level $i-1$), and thus the overall probability of a message successfully reaching the root is $\Pi_i \Pr[\mathcal{E}_i] \geq (1-p^k)^h$. Using the same argument for the other levels of the tree we can get the following: $E(count) \geq \sum_{i=0}^{h}(1-p^k)^i n_i = \frac{(d-p^k d)^{h+1}-1}{d-p^k d-1}$. For $k=2$, $p=0.1$ and $h=10$ we get $E(count) \approx 0.9N$, where N is the total number of nodes. For $k=3$ the bound is close to $0.99N$, thus we have only a 1% degradation in the set of reporting sensors.

4.3 Practical Details

Since our protocols are being developed for use in sensor networks, it is important to ensure that they do not exceed the capabilities of individual sensors. Section 4.3.1 considers the computational costs of generating random numbers for the summation sketches of 3.2. Section 4.3.2 considers the bandwidth overhead of sending sketches.

4.3.1 Binomial Random Number Generation

Existing sensor motes have a small word size (8 or 16 bits), lack floating point hardware and have little available memory for pre-computed tables. For these reasons, standard methods for drawing from the binomial distribution are unsuitable. Here, we outline a randomized algorithm which draws from $B(n,p)$ in $O(np)$ expected running time using $O(1/p)$ space in a pre-computed table and without use of floating point operations. We first note the following relationship between drawing from the binomial distribution and drawing from the geometric distribution, also used in [4].

Fact 1 *Suppose we have a method to repeatedly draw at random from the geometric distribution $G(1-p)$. Let d be the random variable that records the number of draws from $G(1-p)$ until the sum of the draws exceeds n. The value $d-1$ is then equivalent to a random draw from $B(n,p)$.*

The expected number of draws d from the geometric distribution using this method is np, so to bound the expected running time to draw from $B(n,p)$, we simply need to bound the running time to draw from $G(1-p)$. We will make use of the elegant alias method of [15] to do so in $O(1)$ expected time. In [15] Walker demonstrates the following (which has a simple and beautiful implementation):

Theorem 7 (Walker) *For any discrete probability density function $\mathcal{D}$ over a sample space of size k, a table of size O(k) can be constructed in O(k) time that enables random variables to be drawn from $\mathcal{D}$ using two table lookups.*

We can apply this method directly to construct a table of size $n+1$ in which the first n elements of the pdf $\mathcal{D}$ respectively correspond to the probabilities p_i of drawing $1 \leq i \leq n$ from the geometric distribution $G(1-p)$, and the final element corresponds to the tail probability of drawing any value strictly larger than n from $G(1-p)$. Note that for simulating a draw from $B(n,p)$ using the method implicitly defined by Fact 1, we never care about the exact value of a draw from $G(1-p)$ that is larger than n. This direct application enables $O(1)$ draws from $G(1-p)$ in $O(n)$ space, thus yields $O(np)$ expected running time to draw from $B(n,p)$.

To achieve $O(1/p)$ space, we make use of the memoryless property of the geometric distribution (which the binomial distribution does not have). Instead of storing the first n probabilities p_i for the geometric distribution, we store only the first $\lceil 1/p \rceil$ such probabilities, and a final element corresponding to the tail probability of drawing any value strictly larger than $\lceil 1/p \rceil$ from $G(1-p)$. By the memorylessness property, if we select the event corresponding to the tail probability, we can recursively draw again from the table, setting our outcome to $\lceil 1/p \rceil + x$, where x is the result of the recursive draw. The recursion terminates whenever one of the first $\lceil 1/p \rceil$ events in the table is selected, or whenever the accumulated result exceeds n. Since $1/p$ is the expectation of $G(1-p)$, this recursion terminates with constant probability at each round, and thus the expected number of table lookups is O(1). Further reduction in space is possible, but at the cost of incurring a commensurate increase in the expected number of recursive calls.

Using table sizes of $\lceil 1/p \rceil$ and assuming a maximum sensor value of $c_i \leq 2^{16}$ (from a 16 bit word size), the lowest value of p used in summation sketches will be $16^2/2^{16} = 1/2^8$. Therefore, we will have tables for $p = 1/2^1, \ldots, 1/2^8$, with $2, \ldots, 2^8$ entries each, respectively. Walker's method utilizes two values for each entry - the first is an index into the table and the second is a real value used for comparison. The index value only requires one byte since the largest table size is 2^8, and a 64 bit fixed-point real value (8 bytes) should more than suffice. This gives a total table size of $\sum_{i=1}^{8}(2^i) * (1+8) = 4590$ bytes. This can be improved further by reducing the number of entries in each table as mentioned before. The smaller tables (e.g. for $p = 1/2$ and $p = 1/4$) can also be removed in favor of directly simulating the "coin flips" of the geometric distribution, but the space savings is negligible.

4.3.2 Sketch Sizes and Compression

The other main limitation of sensor networks is limited bandwidth. Meanwhile, the main disadvantage of our approach is the increased size of sketches over the simple aggregates of TAG. Later in our experiments, we will be using sketches with 20 bitmaps of 16 bits each. This consumes 40 bytes, whereas 2 bytes sufficed for TAG earlier.

However, as follows from Lemma 1, the sketches are easily compressible since they begin and end with long repeating sequences of the same value. We found that the following simple compression scheme achieved a factor of three compression on average. First, we find the lengths of the prefix of all ones common to each bitmap and the suffix of all zeroes common to each bitmap. Both of these lengths are sent explicitly (consuming $2\log_2\log_2 n + O(1)$ bits) and then the remainders of the bitmaps are sent verbatim. We note that this compression method also makes the transmitted size nearly invariant to the actual bitmap sizes (assuming they are long enough). That is, the bitmaps could have used 32 or 64 bits each, basically just extending the suffixes of zero bits which are already being compressed. Note that the average compression factor of three allows two sketches to easily fit in a TinyDB packet along with headers and extra room to handle variation in compression ratios (TinyDB uses 48 byte packets). This suffices for responding to AVG queries in one packet. Compression for the TAG methods is also possible, but unlikely to derive substantial benefit since they already use 2 bytes or fewer.

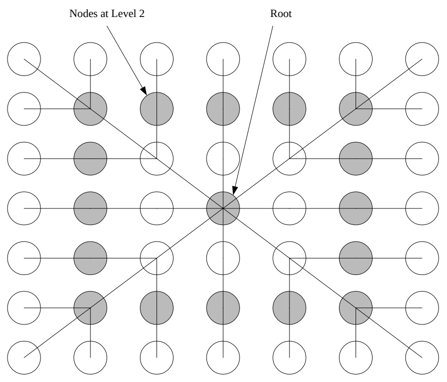


Figure 2. The spanning tree for $N = 49$ nodes on a grid.

5 Experimental Evaluation

We evaluate our methods using the TAG simulator of [13, 14], with modifications to support sketches. Section 5.1 describes the various strategies employed for aggregation and transmission of data. Section 5.2 describes the scenarios we simulated and our results.

5.1 Strategies Employed

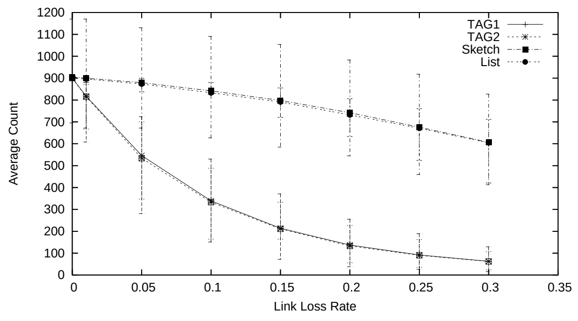
Within each scenario, various strategies for aggregation and transmission were evaluated. Under each of these strategies, each node combines aggregates received from its children with its own reading, and then sends an aggregate to one or more of its parents. Any node within broadcast range which was at a lower level (closer to the root) was considered a candidate parent. The specific strategies considered were

TAG1: The main strategy of [13, 14] (each sensor sends its aggregate to a single parent).

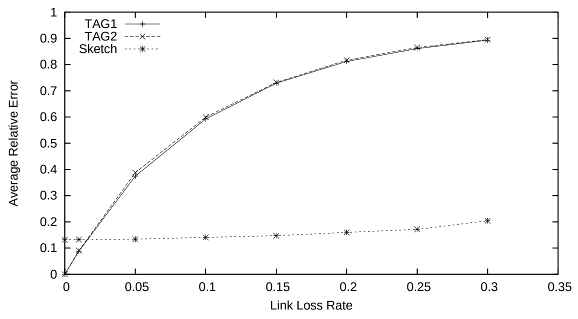
TAG2: The "fractional parents" strategy of [13, 14] described in Section 2.2.

LIST: The aggregate consists of an explicit list of all the items in the aggregate with any duplicates removed. These lists are sent to all parents.

SKETCH: The strategy advocated in Section 4 using duplicate insensitive sketches.



(a) Average counts



(b) Average relative error

Figure 3. Performance varying link loss rates.

5.2 Experimental Results

Our basic experimental scenario is as follows. The network topology is a 30×30 grid with 900 sensors, one placed at each grid point. The communication radius is $\sqrt{2}$ (allowing the nearest eight grid neighbors to be reached) and the default link loss rate is set at 5%. The root node is always at the center of the grid. Figure 2 illustrates a 7×7 grid as an example.

Each experiment performs a count query (counting the sensors) and the aggregate results of 500 runs are taken. The sketches used for count queries use PCSA with 20 bitmaps and 16 bits in each bitmap.

Figure 3 shows the effects of link losses on each strategy's performance. Each curve shows the average count returned with error bars showing the standard deviation. TAG1, TAG2, and LIST all correctly return the answer 900. SKETCH is randomized, but still returns an answer close to 900. However, as the link loss rate increases, both TAG strategies degrade quickly and report counts ultimately averaging fewer than 10% of the total when the loss rate reaches 30%. Meanwhile, both LIST and SKETCH degrade much more gracefully and still report counts of about 66% of the total.

Figure 3(b) shows the relative error of this scenario. Here, given sample value x and correct value $\hat{x}$, the relative error is $\left|\frac{x-\hat{x}}{\hat{x}}\right|$. The correct value is defined to be the result of the LIST strategy, since subject to using the same topology (including spanning tree and parents) with the same losses, the LIST strategy has optimal accuracy. When the loss rate is zero, each strategy has an average relative error of zero, except for SKETCH, which still averages close to 10%. When the loss rate is 15%, the TAG strategies already have more than 70% relative error, while SKETCH has only increased to about 15%. When the loss rate reaches 30%, the TAG strategies have reached an average relative error of 90%, while the multiple parents strategies still have an average relative error of less than 20%. We omit plots of the relative error for all but one of the remaining scenarios since they have similar performance trends and are easily extrapolated from the average counts returned.

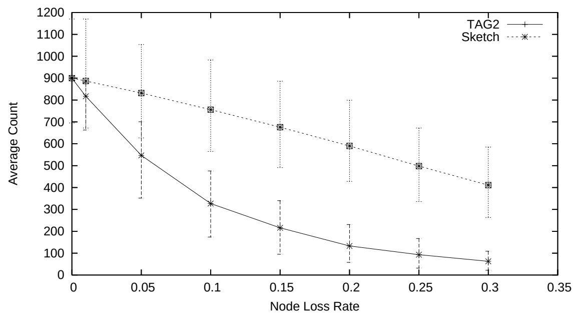


Figure 4. Performance varying node loss rates.

We note that the performance of the TAG strategies are essentially identical when examining the average count returned and the average relative error. The main difference in performance between the two strategies is that TAG1 has a larger variance than TAG2, so we omit TAG1 in subsequent plots. Similarly, the performance of LIST and SKETCH are similar, with LIST having a smaller variance, but we omit LIST since it is infeasible in practice, except for very small sets.

Figure 4 shows the effect of node losses. The general trends here look similar to link loss plots in Figure 3, but the average counts reported drop off faster, while the average relative error grows more slowly. Intuitively, a major difference here for the LIST and SKETCH strategies is that a value can be "lost" if just the node fails, while all of the links to parents must fail to achieve the same loss.

Figures 5 and 6 show the results of using the random grid placements and sum sketches, respectively. The communication range was increased to $2\sqrt{2}$ for the random grid placements, to improve connectivity in areas which randomly received fewer sensors, but it also increases the average number of parents and

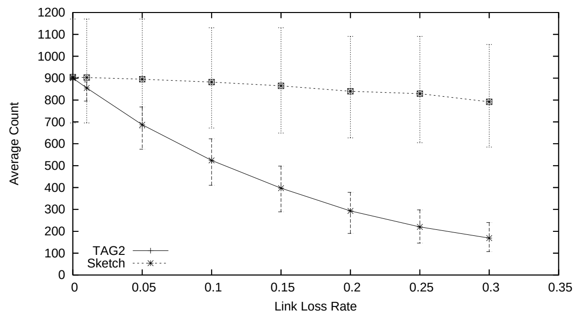


Figure 5. Performance with random placement and communication radius $2\sqrt{2}$

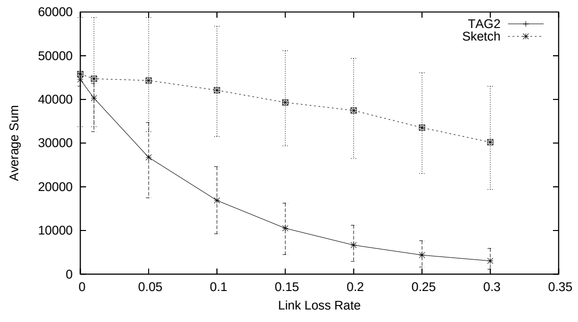
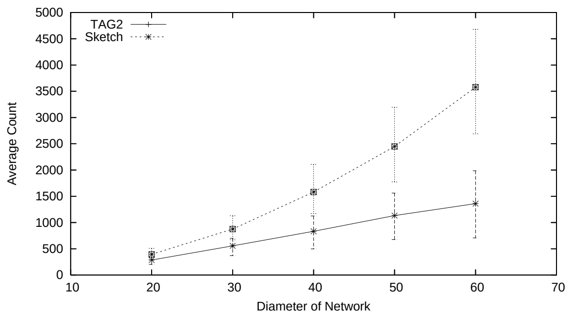
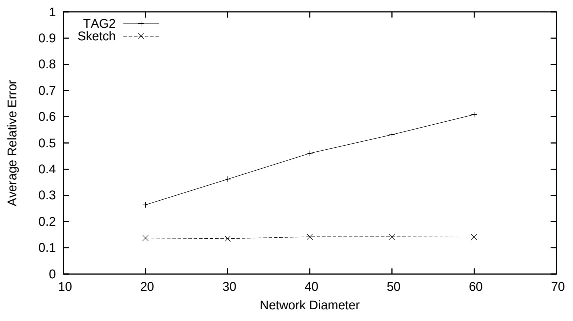


Figure 6. Performance of sum queries using different link loss rates.



(a) Average counts.



(b) Average relative error.

Figure 7. Performance varying network size.

the performance of our scheme. Using sum sketches, each node chose an integer value uniformly at random from the range $[0, 100]$, so the expected sum was $50 * 900 = 45,000$. The basic trends in both figures were essentially the same as when just loss rates were varied. Results for AVG aggregates, combining summation and count sketches, were essentially the same as for SUM and were omitted due to space limitations.

Figure 7 shows the results of our final experiments adjusting the size of the network. Here, the diameter of the network is varied while preserving the grid shape, so a network of diameter d has d^2 nodes laid out on a $d \times d$ grid. Despite the loss rate being held constant, the TAG strategies perform increasingly poorly as the network size increases. Meanwhile, the SKETCH strategy maintains an almost constant average relative error around 13 percent, though it seems slightly higher for the larger network sizes (14 percent).

6 Conclusions and Future Work

In this work, we presented a new framework for efficient and robust aggregation of data in sensor networks using duplicate insensitive sketches. This framework handles not only the many well known counting sketches, but also our new summation sketches and their derivatives, and any other duplicate insensitive sketches which may arise. We then experimentally demonstrated the dramatically improved robustness of this approach over previous ones, and quantified the moderate overheads involved.

We note that the implications of these results reach beyond just sensor networks. For example, these techniques are just as applicable to peer-to-peer networks. In such a setting, nodes tend to have more resources and network behavior may be more predictable, but there is still a significant element of “node failures” from the frequent arrivals and departures seen in such networks. Our results can be applied to any unreliable system with a distributed data set over which best effort aggregate queries are posed.

Acknowledgments

References

[1] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. *Journal of Computer and System Sciences*, 58(1):137–147, 1999.

[2] Z. Bar-Yossef, T.S. Jayram, R. Kumar, D. Sivakumar, and L. Trevisan. Counting distinct elements in a data stream. In *Proc. of RANDOM*, 2002.

[3] G. Cormode, M. Datar, P. Indyk, and S. Muthukrishnan. Comparing data streams using hamming norms (how to zero in). In *VLDB*, 2002.

[4] L. Devroye. Generating the maximum of independent identically distributed random variables. *Computers and Mathematics with Applications*, 6:305–315, 1980.

[5] M. Durand and P. Flajolet. Loglog counting of large cardinalities. In *ESA 2003*, 2003.

[6] P. Flajolet. On adaptive sampling. *COMPUTG: Computing*, 43, 1990.

[7] P. Flajolet and G. N. Martin. Probabilistic counting algorithms for data base applications. *Journal of Computer and System Sciences*, 31, 1985.

[8] S. Ganguly, M. Garofalakis, and R. Rastogi. Processing set expressions over continuous update streams. In *ACM SIGMOD*, 2003.

[9] P.B. Gibbons and S. Tirthapura. Estimating simple functions on the union of data streams. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 281–291, 2001.

[10] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Mining and Knowledge Discovery*, 1(1):29–53, 1997.

[11] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed diffusion: A scalable and robust communication paradigm for sensor networks. In *MobiCOM*, 2002.

[12] V. Kachitvichyanukul and B.W. Schmeiser. Binomial random variate generation. *Communications of the ACM*, 31(2):216–222, 1988.

[13] S. Madden, M.J. Franklin, J.M. Hellerstein, and W. Hong. Tag: a tiny aggregation service for ad-hoc sensor networks. In *OSDI*, 2002.

[14] S. Madden, M.J. Franklin, J.M. Hellerstein, and W. Hong. The design of an acquisitional query processor for sensor networks. In *ACM SIGMOD*, 2003.

[15] A.J. Walker. An efficient method for generating discrete random variables with general distributions. *ACM Transactions on Mathematical Software (TOMS)*, 3(3):253–256, 1977.

[16] Y. Yao and J. Gehrke. The cougar approach to in-network query processing in sensor networks. *SIGMOD Record 31(3)*, 2002.

[17] Y. Yao and J. Gehrke. Query processing in sensor networks. In *CIDR 2003*, 2003.

[18] J. Zhao, R. Govindan, and D. Estrin. Computing aggregates for monitoring wireless sensor networks. In *SNPA*, 2003.