RC24207 (W0703-113) March 23, 2007
Computer Science

IBM Research Report

A Frequency-aware Parallel Algorithm for Counting Stream Items on Multicore Processors

Dina Thomas*
Department of Electrical Engineering
Stanford University
Stanford, CA 94305

Rajesh Bordawekar, Charu Aggarwal, Philip S. Yu
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

*Work done at IBM T. J. Watson Research Center

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

A Frequency-aware Parallel Algorithm for Counting Stream Items on Multi-core Processors

Dina Thomas*
Department of Electrical Engineering
Stanford University
Stanford, CA 94305, USA
dinathomas@gmail.com

Rajesh Bordawekar, Charu Aggarwal, Philip S. Yu
IBM T. J. Watson Research Center
Hawthorne, NY 10532, USA
{bordaw, charu, psyu}@us.ibm.com

ABSTRACT

We present a parallel counting algorithm for estimating frequencies of items from data streams where a stream is ingested and queried in parallel by partitioning data over multiple processing cores of a multi-core processor. We demonstrate that current probabilistic counting algorithms are ill-suited to be used as the sequential kernel in the parallel algorithm due to space limitations, inaccuracies in approximating counts of low frequency items, and inability to identify the absent items in a stream. To address these concerns, we have devised a new sequential counting algorithm, called the Frequency-aware Counting Algorithm (FCM). FCM is related to the Cormode-Muthukrishnan Count-Min algorithm and incorporates novel capabilities for improving accuracy in estimating low-frequency items by dynamically capturing frequency changes of items in the stream, and using a variable number of hash functions per item as determined by an item's current frequency. FCM also uses an auxiliary space-efficient data structure to reduce the errors due to absent items.

We have implemented the parallel counting algorithm on the multi-core Cell processor using the FCM algorithm as the sequential kernel. We experimentally and analytically demonstrate that with similar space consumption, FCM computes better frequency estimates of both the low- and high-frequency items than the Count-Min algorithm by reducing collisions with high-frequency items. FCM also significantly reduces the errors in identifying absent items. In the parallel scenario, as the number of processing cores is increased, using a hash-based data partitioning approach, our parallel algorithm is able to scale the overall **performance linearly** as well as improve the estimate **accuracy**.

1. INTRODUCTION

Recent technological advances have led to a proliferation of applications which can generate and process *data streams*. Data streams are sequences of data items that can be generated *continuously* at *dynamically varying* rates and need to be processed at *equivalent* rates as soon as they are received by the processing elements. Such data streaming applications often process large quantities of data that can potentially grow without limit at a rapid rate, putting enormous burden on the computational and memory resources of the underlying system [25].

One of the key data streaming applications involves determining frequency statistics of the stream items in *real-time*. Examples of such statistics include frequency moments, determining heavy hitters, and order statistics. In this paper, we explore the problem of frequency querying of data streams. In particular, one can query a stream to calculate the number of occurrences or the *frequency* of items in the section of the stream *observed so far*. Formally, this **stream frequency counting** problem can be defined as follows: Let stream $S = (s_1, \ldots, s_N)$ be a sequence of items, where each s_i is a member of a domain $D = (1, \ldots, d)$. Estimate the frequency of an unique item s_j in the subsequence $S(t) = (s_1, \ldots, s_t)$, where $t \leq N$ (This type of query is also referred to as the *point* query). Clearly, the values of N and d can be very large and the item frequency can vary over time. For example, in web click streams or phone call streams, the number of possible unique items (i.e., web pages or phone numbers) could easily range in the order of hundreds of millions or even billions. In many cases, the processing of data collected in large sensor networks is performed on the sensor nodes which have limited memory and power consumption [26]. Obviously, to satisfy the memory and real-time execution constraints, the input stream data can not be stored in its entirety. Therefore, the counting applications employ algorithms that strive to maximize the computational performance while minimizing the memory usage.

A well-known memory-efficient technique for counting items from data streams uses probabilistic data structures [15, 12], e.g., sketches [2]. The sketch method is essentially a random projection based approach which uses either linear projections [2] or hash functions [4, 5] to condense the input stream data into a summary. The results for a point query can then be extracted from these condensed summaries. While the sketch-based approach reduces the space complexity of the counting process, additional modifications are needed to improve their computational performance. A traditional approach for improving computational performance involves partitioning the work across multiple processing entities and executing them in parallel. In recent times, such

*Work done at IBM T. J. Watson Research Center.

parallel approaches have become even more practical due to availability of systems that use *multi-core* processors [8, 7, 18]. The multi-core processors support multiple, potentially heterogeneous, on-chip processing cores connected via high-bandwidth, low-latency interconnection fabric. Such features enable the multi-core processors to provide very high computational performance at relatively low power consumption. These capabilities make the multi-core processors potentially suitable platforms for streaming data processing.

In this paper, we describe a new parallel algorithm for the stream counting problem using a probabilistic sketch data structure. The parallel algorithm uses data partitioning to distribute stream ingestion and querying across multiple processor cores. While designing the parallel algorithm, we identify two key problems with the existing probabilistic counting algorithms: (1) Under limited memory, the estimates for infrequent (low-frequency) items are inaccurate, and (2) It is not possible to detect if an item was absent in the data stream analyzed so far. Hence, the estimates for zero-frequency items (i.e., items absent from the input stream) are also inaccurate. These problems gain importance in the parallel execution model as the data partitioning strategy can lead to an uneven distribution of items across the processing cores. Therefore, items with even modestly high frequencies can sometimes have low or zero-frequencies on particular processors. To address these issues, we propose a new sequential counting algorithm, called the Frequency-aware Counting (FCM) algorithm. The FCM algorithm is related to the Cormode-Muthukrishnan Count-Min (CM) algorithm and incorporates a number of novel features to address the estimation problems for low-frequency and zero-frequency items in the parallel setting. Our parallel algorithm uses the FCM algorithm as the sequential counting kernel.

We experimentally evaluate the parallel algorithm on a modern heterogeneous multi-core processor: the 9-core Cell processor. Although the Cell was initially designed as a game processor, it is being increasingly used in blade servers for developing commercial and scientific applications [19]. Cell's high single-precision floating point performance (204 GFLOPS at 3.2 GHz) along with high memory (25.6 GB/s peak) and I/O (76.8 GB/s peak) bandwidths make it ideal for stream processing [22].

The main contributions of our work are the design and analysis of new parallel and sequential stream counting algorithms, and backing it by experimental performance evaluation on a commercial multi-core processor. Specifically, this work makes the following key contributions:

1. **Frequency-aware Counting Algorithm (FCM):** The FCM algorithm uses three novel ideas to address the estimation issues with low- and high-frequency items. First, it uses a space-conserving data structure to *dynamically* capture the *relative* frequency phase of items from an incoming data stream. Second, the FCM uses a variable number of hash functions for each item as per its current frequency phase (i.e., "high" or "low"). A high-frequency item uses fewer hash functions, whereas a low-frequency item uses more hash functions to update the sketch. Finally, the FCM algorithm uses an additional sketch data structure called the Zero-frequency table to reduce estimation errors due to absent items. We have analytically evaluated these features and computed error bounds for the frequency-aware counting algorithm.

2. **Scalable Parallel Algorithm:** Our parallel algorithm partitions the work across multiple processing cores of a multi-core processor. The algorithm uses two different data partitioning strategies: block-based and hash-based distribution. Each processing core executes the FCM algorithm as the sequential kernel and computes its local count. The local counts are then aggregated to compute the final result.

3. **Implementation on the Cell Processor:** We have implemented the parallel algorithm and the FCM algorithm on the multi-core Cell processor. We have optimized the code to exploit Cell's architectural and programming capabilities (e.g., data and work partitioning, computational strip-mining, and short-vector data parallelization). We have evaluated our implementation using Zipf and Normal datasets and demonstrated that our FCM algorithm improves the estimation quality over the CM algorithm for the entire frequency range of the input data, in both sequential and parallel scenarios. Our experiments illustrate that simply partitioning data across multiple processors does not lead to an improvement in estimation quality as the number of processors is increased. Our parallel algorithm achieves linear performance scalability and estimation improvement, using hash-based data partitioning, along with the zero-frequency table.

This paper is organized as follows: Section 2 examines a number of issues in parallelizing the sketch-based counting algorithms and discusses our proposed parallel sketch-based counting algorithm. In Section 3, we experimentally and analytically compare three existing sketch-based counting algorithms. Section 4 discusses the FCM, new frequency-aware sketch-based counting algorithm. In Section 5, we overview the Cell processor and present details of our implementation on the Cell processor. Section 6 presents the experimental evaluation results. We discuss related work in Section 7. The conclusions are summarized in Section 8.

2. PARALLEL STREAM PROCESSING

Sketch-based probabilistic counting algorithms can be evaluated using three criteria: (1) Quality of results, (2) Space utilization, and (3) Time to store and query the stream datasets. The key goals of parallelizing such algorithms are to improve both the result quality, and times for stream ingestion and processing, while maintaining the space utilization.

For devising a parallel algorithm, we assume an abstract parallel machine consisting of p processing units sharing the main memory. In addition, each processing unit has private local memory. The per-processor local memory is usually substantially smaller than the shared main memory. These processors can communicate either using shared data structures or via explicit messages. One of the processors can act as the coordinator and manage tasks for the remaining processors. This abstract model captures the architecture of most current multi-core processors designed for stream data processing (such as the Cell [8] and the Network processors [7, 18]) and standard multi-threaded programming models like the Pthreads. Thus, our algorithm can be implemented on a wide variety of available software and hardware platforms.

Stream Ingestion Phase

1. Partition the stream across multiple processing units
2. Each processing unit initiates and fills its local sketch

Query Processing Phase

1. Send the query to the participating processing units
2. Each processing unit computes local counts
3. A processing unit sums up the local counts to generate the final result

Figure 1: Outline of the Parallel Stream Counting Algorithm

As the item counting problem is commutative in nature, it can be easily parallelized via partitioning items over participating processors. Figure 1 presents an outline of the parallel counting algorithm. Conceptually, this algorithm has two phases: In the *stream ingestion* phase, the coordinating processor distributes the incoming stream over multiple processing units. Each processing unit initializes a private sketch in its local memory and populates it using a traditional (sequential) counting algorithm over the part of the stream assigned to it. In the *querying* phase, depending on the type of the query (point or range query), the coordinator sends one or more items to the participating processors. Each processor, in turn, uses its private sketch to calculate the local count. The local counts are added by the coordinator and the resultant sum is returned as the approximate result of the query. The two phases can overlap, i.e., a stream can be queried as it is being processed.

The performance of the parallel algorithm depends on: (1) Stream data partitioning strategies and (2) The sequential algorithm. The data partitioning strategies divide the work across multiple cores by distributing the stream data. The data partitioning modifies statistical characteristics of the input data stream. As a consequence, a frequent stream item may appear as a *low frequency* item to some processors and some processors may not even view the item (i.e., it will appear as an absent – *zero-frequency* – item). As we illustrate in the following sections, such behavior leads to increased errors in computing per-processor local counts, in particular, when using limited memory. As the final result is computed by adding local counts, the data partitioning strategies also affect the extent of the error in the final result. The sequential counting kernel used by each processor determines the quality of local results, per-processor space consumption, and the stream ingestion time. The ideal sequential counting kernel should compute accurate results while consuming as little memory as possible.

There are two ways of partitioning data streams: block-based and hash-based partitioning.

- Hash-based partitioning: This approach uses a value-based partitioning strategy in which the coordinating processor hashes the input stream values into p buckets. The buckets are then distributed over different processor groups (e.g., 8 processors can be partitioned as 4 processor groups, each containing 2 processors, or 8 processor groups, each with an individual processor). Each processor within a processor group reads disjoint sections of a bucket and populates its local sketch. During the querying phase, the coordinating processor hashes the query item into a bucket. Only those processors that are associated with this bucket are queried and their estimates are aggregated as the estimated count of the query item.

  The value-based partitioning may lead to unequal distribution of data across the processor groups which can cause load imbalance. This is generally not an issue when the number of processors is small (Section 6). On the other hand, this method of partitioning groups together all occurrences of items with a particular value. Thus, the *relative* frequency of items observed by a processor group is higher than in the input stream.

- Block-based partitioning: In this method, no pre-processing is performed on the input stream. As the stream is read by the coordinating processor, it is divided into equal disjoint chunks. Each chunk is sent to a processor in a round-robin fashion. In the *querying* phase, the coordinating processor forwards the query item to all the processing units. Each processor returns an estimate for the count of the item. The coordinating processor sums up the individual local counts and returns the result as the answer to the query.

  Unlike the hash-based partitioning, block-based partitioning distributes work equally among the processors. Also, the coordinator needs to do less work as there is no need to compute hash values per item in the input stream.

  However, unlike hash-based partitioning, all processors are queried for each query. When the coordinating processor sums up the estimates from each unit, it also sums up the error returned by each unit. Hence the final error bound of this method is p times the error of a single processor, where p is the total number of processors. In the case of hash-based partitioning, the final error bound is p' times the error of a single processor, where p' is the number of processors in the processor group.

3. CHOOSING THE SEQUENTIAL COUNTING KERNEL

As our target multi-core processors have limited per-core memory (e.g., 256 KB for the Cell, 32 KB for the Intel IXP2800), it is necessary to select a sequential counting algorithm with the lowest space consumption. For selecting the sequential counting kernel, we evaluated three well-known counting algorithms: AMS [2], Count Sketch (CCFC) [4], and Count-Min (CM) [6]. All of these algorithms use some form of the probabilistic sketch data structure which gets updated using random values generated via multiple hash functions. However, they differ in the number of hash functions used, the size of the sketch and the way the sketch is updated (see Section 7 for more detailed comparison).

Table 1 presents an analytical comparison of the total space usage, the number of hash functions used, and the hash function ranges, between the AMS, CCFC, and CM algorithms. For comparison purposes, we use a pair of user-specified parameters, error bound, ϵ, and probability bound, δ. For the CCFC and AMS algorithms, the error is within a factor of ϵ with the probability of $1 - \delta$, and for the CM algorithm, the error is within a factor of ϵ times the L1-norm of the sketch with the probability of $1 - \delta$. As Table 1 illustrates, for a given ϵ and δ, for CM and CCFC algorithms,

Algorithm	Hash range w	# Hash Functions d	Space
AMS	1	$2log(\frac{1}{\delta})$	$\frac{16}{\epsilon^2 d}$
CCFC	$\frac{8}{\epsilon^2}$	$log(\frac{n}{\delta})$	wd
CM	$\frac{e}{\epsilon}$	$log(\frac{1}{\delta})$	wd

Table 1: Comparing Algorithmic Characteristics of the three selected Stream Counting Algorithms: AMS, CCFC, and CM.

the space utilization is determined by the number of hash functions and the hash function range. Among the three algorithms, the CM algorithm is the most space-efficient and the space complexity of the other two algorithms is worse by an order of magnitude.

To experimentally compare these algorithms, we analyzed the implementations of these three algorithms from the MassDAL Public Codebank[1] against 16 MB (4M Integers) of Zipf ($\lambda = 1.1$) and Normal ($\sigma = 20$) data. We ran a point query using $\epsilon = 0.1$ and $\delta = 0.001$, to compute the frequency of a stream item against every member of the dataset and compared the results computed by the algorithms with the actual frequency. Figures 2 and 3 illustrate the results for the Normal ($\sigma = 20$) and Zipf ($\lambda = 1.1$) datasets. The X-axis represents the input data set sorted using the actual frequency and Y-axis represents the frequency calculated by various algorithms.

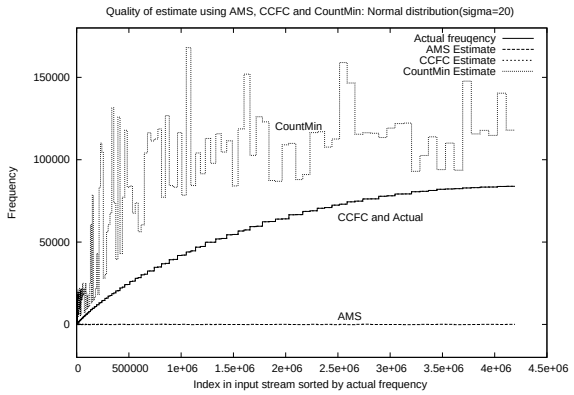

Figure 2: Evaluation of the three selected Counting Algorithms on a 16 MB Normal ($\sigma = 20$) dataset.

As these Figures illustrate, for both datasets, for the entire frequency range, the CCFC algorithm computes the best approximations while the AMS algorithm performs the worst. The quality of approximation by the CM algorithm improves as the item frequency increases. However, the CCFC algorithm requires far more space ($log\frac{n}{\delta}\frac{8}{\epsilon^2}$) than the CM algorithm ($log\frac{1}{\delta}\frac{e}{\epsilon}$) and its space consumption increases quadratically ($O(\frac{1}{\epsilon^2})$) as the error parameter ϵ is reduced. In our experiments, the CM sketch required only 1536 bytes whereas the CCFC sketch used 51816 bytes and the AMS sketch used 83816 bytes. Furthermore, the CCFC performance degraded substantially when run with space comparable to the CM algorithm (for the Normal ($\sigma = 20$) data, the average error per unique items increased from 0 to 30693, and for the Zipf ($\lambda = 1.1$) data, from 150 to 7100). Among the three algorithms, the CM algorithm provides the best accuracy while consuming the lowest space and its space consumption increases linearly as the error parameter ϵ is reduced (unlike CCFC, where the space consumption increased quadratically). Therefore, we decided to use the Count-Min (CM) algorithm as the basis of our sequential kernel.

[1]www.cs.rutgers.edu/∼muthu/massdal.html.

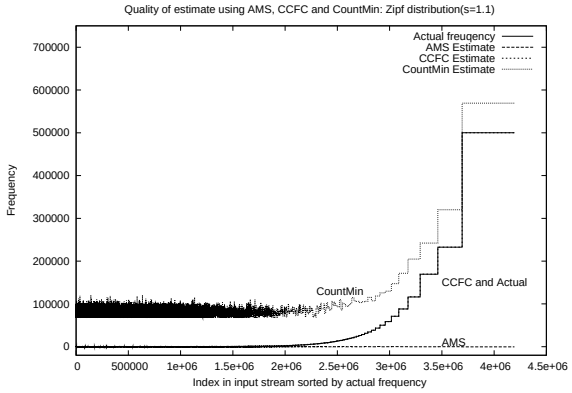

Figure 3: Evaluation of the three selected Counting Algorithms on a 16 MB Zipf ($\lambda = 1.1$) dataset.

4. FCM: A FREQUENCY-AWARE COUNTING ALGORITHM

As noted in the previous section, the Count-Min (CM) algorithm is more accurate for approximating frequencies of high-frequency (heavy hitter) items. The CM algorithm uses a sketch with d hash functions of range w (it can be viewed as a table with d rows and w columns), and for every item to be inserted, uses every hash function to select a position in the corresponding row and increments its value by 1. At query time, the count estimate of an item is computed as the minimum of values stored in the corresponding positions in all d rows. This approach causes the low-frequency counts being tainted due to the collisions between the low- and high-frequency items. The Count Sketch (CCFC) algorithm addresses this problem by considering additional hash functions to partition the d hash functions into two disjoint subsets and updating them differently. As we have observed in the previous section, the CCFC sketches are at least a factor $\frac{1}{\epsilon} \cdot log(n)$ larger than the CM sketches. Hence, we need a different strategy for reducing collisions.

4.1 Frequency-aware Sketch Processing

Given an error bound ϵ and a probability bound δ, FCM uses the same sized sketch data structure as the CM algorithm. The FCM sketch is a set of d uniform pairwise independent hash functions (Figure 4), each with the range w . We use the universal hash functions for computing hash values (i.e., hash(x)= $(a \cdot x + b) mod(P)$, where a and b are constants and P can be either $2^{31} - 1$ or a large number).

In contrast to CCFC, the FCM sketch uses *variable* number of hash functions for an item based on its current *frequency phase*. An item, when deemed as a high-frequency item, uses *fewer* hash functions than a low-frequency item. To reduce the number of collisions further, a subset of the d hash functions is updated per item. The subset is chosen as a hash value of the item using two additional hash functions: first one is used to compute an initial offset into the d rows and the other computes a gap between consecutive hash tables, and the subset is chosen in a round-robin fashion. This approach differs from the CM sketch where every item updates all rows of the sketch beginning with the first row. For example, Figure 4 illustrates the ingestion of a high-frequency item i_h and a low-frequency item i_l into the FCM sketch. Both i_h and i_l have different initial offsets and gaps. The low-frequency item i_l uses more hash functions (6) than the item high-frequency i_h (3). Even with these schemes, there may be a collision between the two items, as illustrated in Figure 4.

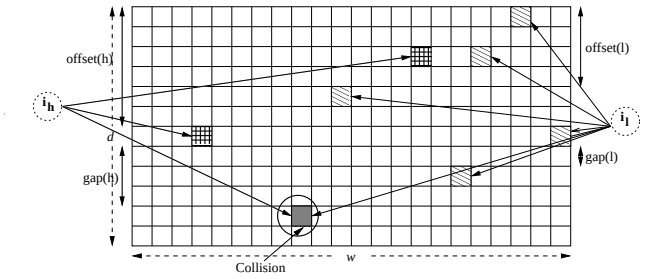

Figure 4: Frequency-aware updating of the FCM sketch. High-frequency items update fewer hash tables than the low-frequency items.

4.1.1 Misra-Gries Frequency Counter

As the frequency of a stream item can dynamically change over time, FCM determines the frequency phase (i.e., high or low) of a stream item over the section of the stream processed so far. For dynamically detecting relative frequency phase of a stream item, we employ a space-efficient counter based on the Misra-Gries (MG) algorithm [23]. The MG algorithm approximates the set of k heavy hitter items from an online stream using $k \cdot log(k)$ space. At any point in the stream, the algorithm tracks those items whose frequency is more than $1/k$ for some constant k, and returns an over-approximation of the set of items that satisfies this condition.

In our implementation, we use a list of $< item, count >$ pairs, called the MG counter, to keep track of counts of unique items (Figure 5). The input stream is divided into

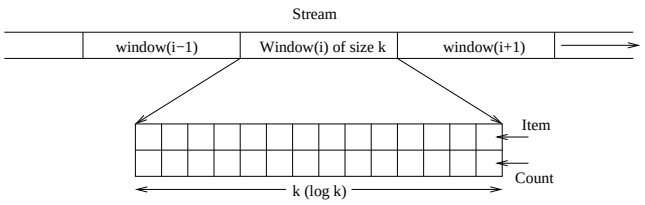

Figure 5: Dynamic Frequency-phase Detection using the Misra-Greis(MG) Counter.

(Stream Ingestion Phase) For every item i in a stream:
I1. Update the MG Counter to determine its frequency phase
I2. Calculate the offset, gap, and # of hash functions, d_h or d_l
I3. Use d_h or d_l hash functions to increment the buckets in the sketch. Update d_{avg}.
I4. Use d hash functions to increment the buckets in the zero-frequency table **(used only in the parallel scenario)**
(Query Processing Phase) For the item i in a point query:
Q1. Calculate the offset, gap
Q2. Compute an estimate from the FCM sketch by minimizing values over d_{avg} buckets
Q3. Compute an estimate from the zero-frequency table by minimizing values over d buckets
Q4. Compute the final result as the minimum of estimates from the sketch and the zero-frequency table.

Figure 6: Frequency-aware Counting Algorithm.

windows of size k. When an item is being processed, we first check if it is in the counter. If the item is in the list, its count is incremented. Otherwise, a new entry is inserted with an initial count depending on the index i of the window being read. After $i \cdot k$ items have been processed, that is, after the i^{th} window, any item with count less than $(i+1)$ has frequency less than $1/k$ and so is deleted from the list. While processing the $(i+1)^{th}$ window, if a new entry is observed then its initial count is set at i, which is the maximum number of occurrences of the item, after which it could have been deleted from the list. This initial count ensures that an entry whose frequency is greater than $1/k$ is not missed. However, this approach may introduce false positives. The efficacy of the MG counter depends on the input distribution and the constant k. FCM classifies an item as an high-frequency item if it is present in the MG counter. We set a threshold value for the number of items being processed to prevent items from early sections of the stream being classified as high-frequency items. This classification is then used to differentially fill in the FCM sketch.

4.1.2 FCM Sketch Updating and Querying

Figure 6 presents the FCM algorithm. Given an item q, we first check if q is present in the MG counter (Step I1). If the item is in the counter, we treat it as an high frequency item and choose a fraction, d_h, of the d hash tables to fill in. Otherwise, if the item is not in the counter we choose a larger fraction, d_l ($d_l \geq d_h$), of the d hash tables to fill in. We use the value of the item to be inserted for computing the offset and gap (Step I2). These values are computed using uniform and pairwise independent hash functions, each with range $[1...d]$. Given an item, we first calculate its offset and gap, and in a round robin fashion, identify d_h or d_l hash tables to fill in. To minimize self-collisions during the round-robin updating of the hash tables, we choose a prime value for d. For each chosen hash table, the item is hashed into a bucket i, $0 < i < w$, and the count in the bucket is incremented by 1 (Step I3). Since the frequency phase of a stream item can vary dynamically during the stream ingestion process, we also keep track of the average number of hash functions used by the unique items ingested so far in the sketch, d_{avg}.

Once the data stream has been analyzed and a sketch has been constructed, we query the sketch to answer point query estimates. Given a query item q, using the same hash

functions, we compute its offset and gap, and using the same round robin fashion, d_{avg} hash tables are chosen from the computed offset(Step Q1). For each table, the appropriate bucket i is selected using the corresponding hash function employed during ingestion. The value in the bucket i is then returned as an estimate of the count of the item. Among the d_{avg} counts obtained, we return the *minimum* value as the final count estimate for the item q from the sketch (Step Q2). This approach differs from the CM sketch where the final result is computed as the minimum of d counts.

4.1.3 Error Analysis

Let $H = \{h_i | 1 \leq i \leq d\}$ be the set of the d hash functions of the sketch, each with range w. Consider a data stream with n distinct items, say $\{1, \ldots, n\}$. Let items $1 \leq i \leq h$ be the high frequency items and $h+1 \leq i \leq n$ be the low frequency items. We refer to the set of high frequency items as X_h and the set of low frequency items as X_l, where $\|X_l\| + \|X_h\| = n$. Let a_i be the exact count of occurrences of item i. Let $A = (a_1, a_2, \ldots a_n)$ be a vector of counts of all distinct items in the stream. Let $A_h = (a_1, a_2, \ldots a_h)$ be a vector of counts of distinct items in X_h. Let $A_l = (a_{h+1}, a_{h+2}, \ldots a_n)$ be a vector of counts of distinct items in X_l. Let $S_i \subseteq H$ denote the set of hash functions chosen for item i. d_h denotes the number of hash functions from H chosen to fill in the sketch for items in X_h. The notation d_l denotes the number of hash functions from H chosen to fill in the sketch for items in X_l. We visualize the sketch as a $d \times w$ table where rows correspond to the hash functions and columns correspond to the bucket number. Let $count[j, h_j(i)]$ be the value in the sketch in cell $(j, h_j(i))$. $X_{i,j}$ is a random variable which denotes the error in $count[j, h_j(i)]$ for estimating a_i. $I_{i,j,k}$ is an indicator function defined as follows:

$$I_{i,j,k} = \begin{cases} 1 & \text{if } i \neq k,\ h_j(i) = h_j(k) \text{ and } j \in S_i \text{ and } j \in S_k; \\ 0 & \text{otherwise.} \end{cases}$$

Thus $I_{i,j,k}$ is an indicator function for collision between items i and k on hash function h_j. $I_{i,j,k} = 1$ depends on three independent events as follows:

1. $j \in S_i$. This event denotes that h_j is chosen for item i. Choosing the elements of the subset S_i uniformly from the set H, this probability is equal to $|S_i|/|H|$.
2. $j \in S_k$. This event denotes that h_j is chosen for item k. Choosing the elements of the subset S_k uniformly from the set H, this probability is equal to $|S_k|/|H|$.
3. $h_j(i) = h_j(k)$. This is the event that item i and k collide on hash function h_j. Since we consider uniform hash functions with range w, the probability of this event is $1/w$.

By definition, $X_{i,j} = \Sigma_{k=1}^{k=n} a_k.I_{i,j,k}$. $E(X_{i,j}) = E(\Sigma_{k=1}^{k=n} a_k.I_{i,j,k}) \leq \Sigma_{k=1}^{k=n} a_k.E(I_{i,j,k})$. If k is a low frequency item $|S_k| = d_l$; and if k is a high frequency item $|S_k| = d_h$. Therefore,

$$E(I_{i,j,k}) = \begin{cases} 1/w.(\frac{d_l}{d})^2 & \text{if } i \in X_l \text{ and } k \in X_l; \\ 1/w.(\frac{d_h}{d})^2 & \text{if } i \in X_h \text{ and } k \in X_h; \\ 1/w.(\frac{d_l}{d}.\frac{d_h}{d}) & \text{otherwise.} \end{cases}$$

Let $\hat{a_i}$ be the estimate for a_i derived from the sketch. Let $w = \frac{e}{\epsilon}$. For a vector $A = (a_1, \ldots, a_n)$, we denote its L1-norm as $|A|_1$, and define it as $|A|_1 = \Sigma_{i=1}^{i=n} |a_i|$. For $i \in X_l$, $E(X_{i,j}) \leq 1/w.((\frac{d_l}{d})^2.|A_l|_1 + \frac{d_l}{d}.\frac{d_h}{d}.|A_h|_1) = \frac{e}{\epsilon}.((\frac{d_l}{d})^2.|A_l|_1 + \frac{d_l}{d}.\frac{d_h}{d}.|A_h|_1)$. Similarly for $i \in X_h$, $E(X_{i,j}) \leq = \frac{e}{\epsilon}.((\frac{d_h}{d})^2.|A_h|_1 + \frac{d_l}{d}.\frac{d_h}{d}.|A_l|_1)$. Let $C_1 = ((\frac{d_l}{d})^2.|A_l|_1 + \frac{d_l}{d}.\frac{d_h}{d}.|A_h|_1)$ and $C_2 = ((\frac{d_h}{d})^2.|A_h|_1 + \frac{d_l}{d}.\frac{d_h}{d}.|A_l|_1)$.

Let $D_i \subseteq H$ be the set of d_{avg} hash functions used to query the count of item i. By Markov inequality, for $i \in X_l$, $Pr[\hat{a_i} > a_i + \epsilon.C_1] = Pr[\forall j \in D_i.count[j, h_j(i)] > a_i + \epsilon.C_1]$
$= Pr[\forall j \in D_i.a_i + X_{i,j} > a_i + \epsilon.C_1]$
$= Pr[\forall j \in D_i.X_{i,j} > e.E(X_{i,j})] < e^{-|D_i|} = e^{-d_{avg}}$. Hence for $i \in X_l$, we have $Pr[\hat{a_i} > a_i + \epsilon.(\frac{d_l}{d})^2.|A_l|_1 + \frac{d_l}{d}.\frac{d_h}{d}.|A_h|_1)] < e^{-d_{avg}}$. Similarly for $i \in X_h$, $Pr[\hat{a_i} > a_i + \epsilon.(\frac{d_h}{d})^2.|A_h|_1 + \frac{d_l}{d}.\frac{d_h}{d}|A_l|_1)] < e^{-d_{avg}}$.

In CM sketch, there is no differential filling of the sketch for high and low frequency items. Hence $d_l = d_h = d$, $d_{avg} = d$ and $C_1 = C_2 = |A_l|_1 + |A_h|_1 = |A|_1$. Hence $Pr[\hat{a_i} > a_i + \epsilon.(|A_l|_1 + |A_h|_1)] < e^{-d}$. From the above equations, we observe that the error bound in FCM has fractional weight assigned to the terms $|A_l|_1$ and $|A_h|_1$, which makes it a tighter bound compared to that for the CM sketch. Further, the weight assigned to the count of high frequency terms, i.e., $|A_h|_1$, is smaller as $d_h < d_l < d$. Hence, the effect of high-frequency items polluting the sketch estimates for other items is reduced.

4.2 Reducing errors in estimating counts of absent items

One of the fundamental problems in all existing probabilistic counting algorithms is their lack of preserving precise history of the processed stream items. This results in their inability to identify absent (zero-frequency) items, *irrespective of the size of the sketch being used*, and leads to significant errors while estimating counts for such items. Existing approaches for detecting absent items use probabilistic data structures like the Bloom Filters [24]. While Bloom filters are fairly accurate, they consume far more space than the FCM sketch. (A well-implemented Bloom filter with 1% false positive rate requires on average 9.6 bits per item)

Given limited memory availability, the FCM algorithm aims to *reduce* the *magnitude* and *frequency* of errors in estimating counts of zero-frequency items. For this purpose, the FCM algorithm uses an additional sketch data structure called the *Zero-frequency Table* (Figure 7). The key idea is to ensure that a hash table collision between two items in the FCM sketch does not *repeat* in the zero-frequency table. For example, if items x and y map to the same hash bucket i using a hash function h_j in the FCM sketch, they don't map to the same bucket when using a different hash function h_k in the zero-frequency table.

The zero-frequency table is also a sketch with d uniform pairwise independent hash functions, each with the range w', where $w' = w + \delta$. Let the i^{th} hash function in the FCM sketch be $h_i(x) = ((a_i.x + b_i) mod(P)) mod(w)$. Then the i^{th} hash function in the zero-frequency table is chosen as $h'_i(x) = ((a_i.x + b_i) mod(P)) mod(w')$. δ is chosen as $O(w)$, and δ and w are co-prime. This implies w and $w' = w + \delta$ are relatively prime as well. While ingesting a stream, the FCM algorithm updates both the sketch and zero-frequency table. However, unlike in the sketch, irrespective of its frequency phase, all d hash functions of the zero-frequency table are used. For every row j, $0 < j \leq d$, the item is hashed into a bucket i, $0 < i < w + \delta$, and the count in the bucket is incremented by 1 (Step I4 in Figure 6). Now consider

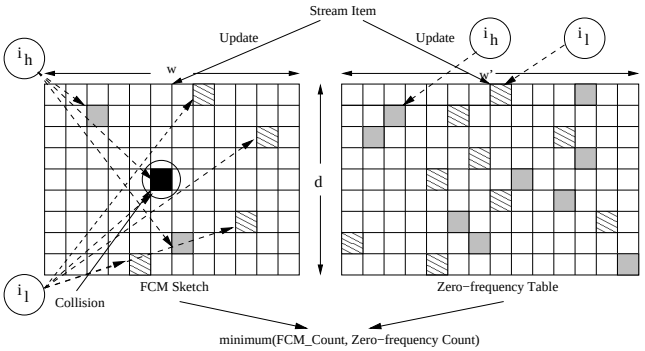

Figure 7: Reducing estimation errors due to absent items using the Zero-frequency Table.

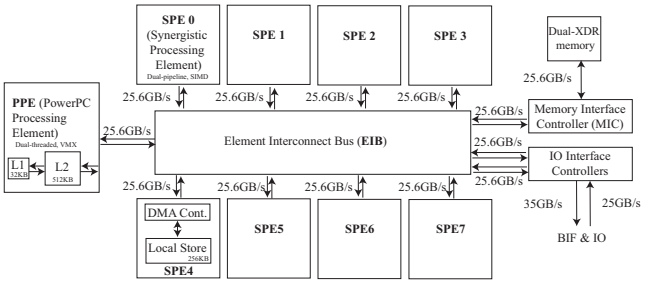

Figure 8: Architecture of the Cell Processor.

a hash function $h_1(x) = ((a_1x + b_1) \bmod w)$ for the data points x_1 and x_2 in the FCM sketch. A collision occurs in the sketch only if $a_1 \cdot (x_1 - x_2)$ is divisible by w. For the same data points, a collision occurs in the zero-frequency table only if $a_1 \cdot (x_1 - x_2)$ is divisible by w'. Since w and w' are relatively prime, this implies that for a collision in both tables, $a_1 \cdot (x_1 - x_2)$ is divisible by $w \cdot w'$. For uniform hash functions, this happens with probability $1/(w \cdot w')$. We note that we are using space proportional to $w+w'$ and instead of collisions with probability $1/(w+w')$, we are now getting at least one table without a collision with probability $1/(w \cdot w')$.

During a point query, an estimate is computed from the FCM sketch by minimizing over d_{avg} hash functions. Similar estimate is computed from the zero-frequency table by minimizing over d hash functions (Step Q3 in Figure 6). The final estimate for the point query is computed as the minimum of the FCM estimate and the zero-frequency estimate (Step Q4 in Figure 6). As the final value is the smaller of the two estimates, the result can be either 0 (i.e., there was no conflict in either or both data structures) or the smaller value (i.e., there was conflict in both data structures). However, in both cases, the error in estimating the count of a zero-frequency item would be less than that for the CM sketch.

Thus, using the frequency-aware sketch updating and the zero-frequency table, the FCM algorithm improves the estimate accuracy for both the low-frequency and zero-frequency items.

5. OVERVIEW OF THE CELL PROCESSOR

The Cell processor is designed primarily for compute- and data-intensive streaming applications. It is a 64-bit single-chip heterogeneous multiprocessor with nine processing cores: one general-purpose processor, called the PPE and eight special purpose co-processors, called the SPEs. Both the PPE and the SPEs run at the same clock frequency. These processors are connected via a high-bandwidth internal bus and can access the shared, coherent main memory (Figure 8). The PPE is a PowerPC-based RISC core and runs the operating system to manage system resources of the entire multiprocessor. It has 32 KB instruction and data L1 caches, and a 512 KB integrated L2 cache. Each SPE is also a 64-bit RISC core natively supporting a short-vector 128-bit single-instruction multiple-data (SIMD) instruction set. The Cell SPE is a dual-issue, statically scheduled SIMD processor. Each SPE holds a 128-entry 128-bit register file to execute these SIMD instructions. The SPE SIMD instruction set can support multi-way (2,4,8, and 16) data parallelism. The SPE control unit can issue 2 instructions per cycle, in-order. Instead of caches, each SPE has a 256 KB private local store which is used to hold both instructions and data. The SPE load and store instructions manage the data transfer between the register file and the local store. Each SPE has dedicated single-precision floating point and integer vector units. Although designed primarily as a SIMD processor, the SPE can also execute scalar computations. However, as the SPEs lack branch prediction hardware, execution of scalar SPE code with conditionals is not optimal. Finally, there is no virtual-memory support on the SPEs and the only way to access the main memory from the SPEs is via explicit asynchronous direct memory access (DMA) commands. The DMA is also used for communicating among different SPE local stores [22]. The PPE and SPEs have different instruction-set architectures (ISAs) and the Cell compiler automatically generates the appropriately optimized native code.

The Cell's multi-core architecture can be exploited via a variety of parallel programming models [8]. A Pthreads like task-parallel programming model enables a host program executing on the PPE to spawn multiple threads which can execute different programs on different SPEs. Each SPE program can then use the SIMD instructions to exploit Cell's data parallel facilities. The SPE code performance can be further improved by using instruction-level parallelism via SPE's dual execution pipes. The Cell also supports a shared-memory programming model where multiple SPEs can share data from their local stores using DMAs over a distributed shared address space. Other supported programming models include function offloading and computation acceleration models [8].

5.1 Implementing the parallel counting algorithm on the Cell

The Cell implementation of our parallel counting algorithm uses the master-slave approach using the Cell's task-parallel programming model. This approach also used for programming network processors [7, 18]. Our implementation divides the work between the PPE- and SPE-based components. The sequential PPE code acts as the coordinating processors. It loads the SPE code into individual local stores and then invokes the sequential counting kernels on one or more SPEs. The PPE code reads the stream data and partitions it using either block- or hash-based partitioning schemes. At the query processing time, the PPE

propagates the query to the participating SPEs, collects the results from these SPEs and returns the final result.

Each participating SPE code executes the core sequential counting kernel (e.g., the FCM). The SPE implementation of the FCM uses the native 128-bit SIMD instruction set to accelerate a variety of key computational functions (e.g., the hash functions). The SPE code also uses extensive loop unrolling to enable instruction-level parallelism. The SPE kernel initializes key data structures in the 256 KB local store (e.g., sketch, zero-frequency tables, buffers for memory accesses, etc.). Once initialized, each SPE instance fetches its portion of the stream data via DMA from the main memory. The core ingestion algorithm (Figure 6) needs to be *strip-mined* as a single DMA can fetch only 16 KB of data at every invocation. The DMA memory-access calls use the double-buffering approach to hide the memory latencies. All operations on the data buffers are SIMDized as well. During the querying phase, the PPE multicasts a point query to the participating SPEs. The SPEs access their local data structures to compute local counts, and send them back to the PPE. Our parallel counting algorithm does not require any inter-SPE communication during stream ingestion or querying.

6. EXPERIMENTAL EVALUATION

We have evaluated the parallel counting algorithm on the Cell using two sequential counting kernels: FCM and CM. We evaluated the implementations using 16 MB (4M integers) data with Zipf ($\lambda = 1.1, 1.5$) and Normal ($\sigma = 10, 20$) distributions under the error bound $\epsilon = 0.087$, and the probability bound $\delta = 0.0002$. Based on these parameters, for both the CM and FCM algorithms, we used a sketch with 17 hash functions (i.e., $d = 17$) with the range of 31 (i.e., $w = 31$). We used the same universal hash functions for updating the FCM and CM sketches. For these experiments, d_h was set to $\frac{d}{2}$ and d_l was set to $\frac{4}{5}d$.[2] Each sketch required 2108 bytes and the zero-frequency table required 2356 bytes. The FCM algorithm also used a MG frequency counter with the stream window size $k = 8$ (corresponding number of entries in the MG frequency counter was $klogk = 24$). Note that the additional space required by the MG frequency counter was substantially less than the CM or FCM sketch and FCM used the zero-frequency table only in the parallel scenario. For the FCM algorithm, the per SPE memory consumption was around 110 KB, which included the space of two data buffers, the sketch, zero-frequency table, the frequency counter, etc. The CM algorithm consumed slightly less SPE local space, as it did not use the MG frequency counter and the zero-frequency table. We ran our experiments on a 2.1 GHz Cell-based blade with a single Cell Broadband Engine(BE) Processor, with a single PPE and 8 SPEs.

6.1 Evaluation of the FCM Algorithm

The first three experiments evaluate the FCM algorithm against the CM algorithm using a single SPE. Figures 9 and 10 present the CM and FCM estimates for the low-frequency range of the input datasets. As these graphs illustrate, while consuming similar amounts of space as the CM algorithm

[2]To a priori determine optimal d_h and d_l for an input data distribution is currently an open problem and is being investigated.

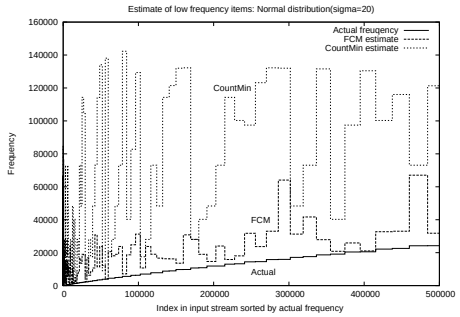

Figure 9: Comparing the estimate quality of CM and FCM algorithms for low-frequency items in a Normal ($\sigma = 20$) dataset.

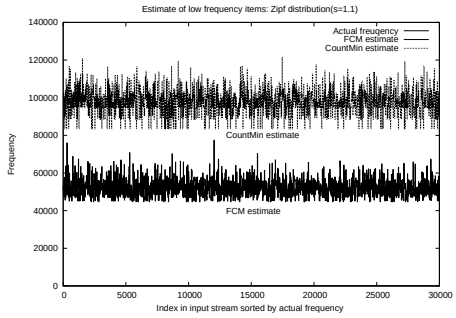

Figure 10: Comparing the estimate quality of CM and FCM algorithms for low-frequency items in a Zipf ($\lambda = 1.1$) dataset.

(FCM uses an additional 24-entry MG counter), the FCM estimates are closer to the actual frequency counts. These results validate our proposal of using frequency-aware sketch updating to avoid collisions with high-frequency items. They also validate the application of the Misra-Greis frequency counter and the choices of d_h and d_l.

Figure 11 presents the behavior of the FCM and CM algorithms while estimating counts of absent items. In this experiment, the FCM and CM sketches were populated using the 16 MB Zipf dataset and then queried using items absent from the input dataset. Figure 11 represents the frequency estimates of both algorithms calculated over the entire lifetime of the input stream. The graph demonstrates that the errors in count estimates of the FCM algorithm are *fewer* and *smaller* than the CM algorithm. In fact, the average error per unique absent item for the CM algorithm was 13 times larger than that for the FCM algorithm. This result provides an experimental validation of our approach of using a space-efficient zero-frequency table along with the sketch for reducing errors for absent items.

6.2 Evaluation of the Parallel Counting Algorithm

The following experiments evaluated the parallel counting

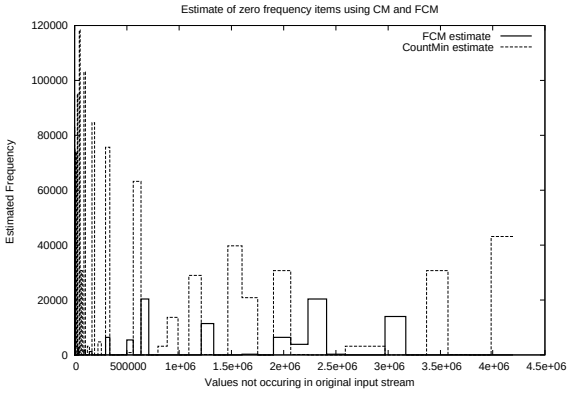

Figure 11: Evaluation of the zero-frequency table using a Zipf ($\lambda = 1.1$) dataset. The average error per unique items is calculated over the entire stream.

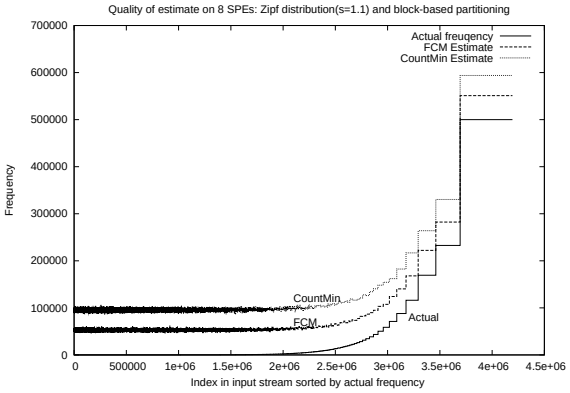

Figure 13: Quality of estimating a Zipf ($\lambda = 1.1$) input data using 8 SPEs via block-based data partitioning.

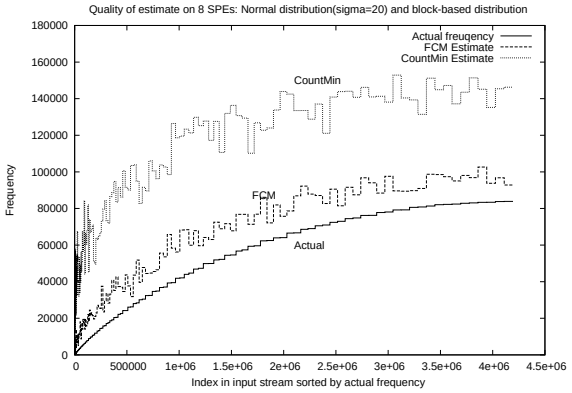

Figure 12: Quality of estimating a Normal ($\sigma = 20$) input data using 8 SPEs via block-based data partitioning.

algorithm using the FCM and CM counting kernels. We scaled the number of SPEs from 1 and 8, and partitioned the data using block-based partitioning with 16 KB blocks and hash-based partitioning over 8 processor groups (each processor group had a single member). For the hash-based partitioning, we used the last three bits of the item value as the function for partitioning the input data set into 8 buckets. Each bucket was then assigned to a SPE. In this section, for evaluation purposes, we use average error per unique items normalized over a window of 1024 items.

Figures 12, 13, 14, and 15 illustrate the estimates for the Normal and Zipf datasets using the parallel algorithm over 8 SPEs. Each experiment is run using the CM and FCM algorithms as the sequential kernel with the block-based and hash-based data partitioning. As these graphs demonstrate, in most cases, the FCM algorithm substantially improves the estimation quality over the CM algorithm for the *entire frequency range*. The exception being the low-tailed Normal ($\sigma = 10, 20$) datasets (Figure 15), where FCM and CM estimates are very accurate due to fewer collisions among low- and high-frequency items. A key point to note is that both CM and FCM algorithms are using the same space for storing their sketches. Therefore, the improvement is mainly due to FCM's better estimation of low-frequency items and zero-frequency items. The reduced number of collisions among high- and low-frequency items in the FCM algorithm also improves the estimates of high-frequency items.

Further, for both algorithms, the overall quality of estimation improves when the stream data is partitioned using the hash-based partitioning scheme. There are three reasons for this improvement. First, when the stream data is partitioned using a hash-based partitioning scheme, all occurrences of a particular value are grouped together. Thus the *relative* frequency of the items observed by a processor group increases. As the CM and FCM algorithms both provide good estimation of high-frequency items, the per-processor estimates improve substantially (This effect was particularly prominent for the Normal datasets as displayed in Figure 15). Second, as the data is partitioned only across a subset of processors, the error in the final estimate is bound by the number of processors in the processor group (in our experiment, one processor). Finally, when the processor group had only one processor, there were no errors due to zero-frequency items (In the general case, when the processor group has more than one processor, hash-based partitioning would experience additional errors due to zero-frequency items). The combination of improved local estimates and aggregation over a smaller number of processors leads to substantial improvement in the final estimate.

To further examine the effect of hash-based partitioning on the estimation errors, we measured the normalized estimation error for both block- and hash-based partitioning for both Zipf and Normal datasets while scaling the number of SPEs from 1 to 8 (Figures 16 and 17). As Figure 16 illustrates, for the block-based partitioning, the normalized estimation error does not decrease as the number of SPEs is scaled up from 1 to 8. In fact, in one case, the error *increases* as the number of SPEs is increased. On the contrary, while

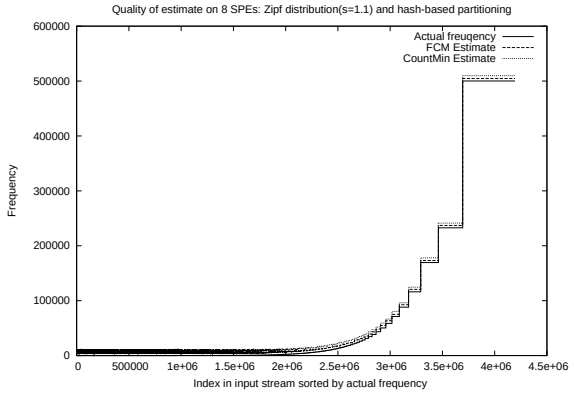

Figure 14: Quality of estimating a Zipf ($\lambda = 1.1$) input data using 8 SPEs via hash-based data partitioning. The normalized estimation error was reduced from 2.13 (for CM) to 1.13 (for FCM).

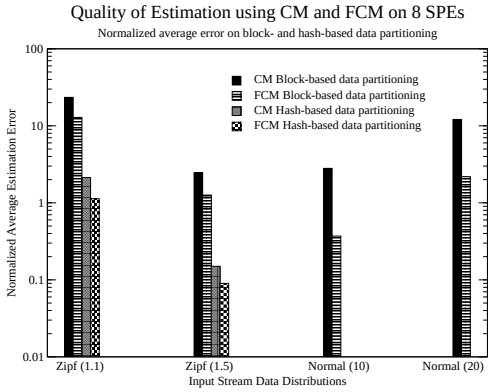

Figure 15: Comparison of estimation quality for Zipf and Normal data distributions using block- and hash-based data partitioning. Note the logarithmic Y-axis. Errors in estimating the Normal datasets using hash-based data partitioning too small for representation.

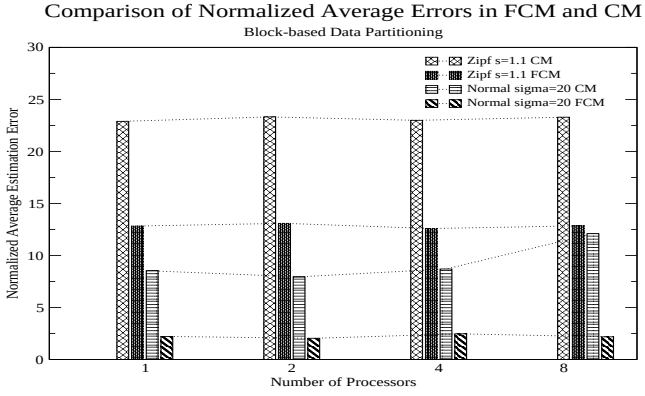

Figure 16: Comparison of errors in FCM and CM using block-based data partitioning from Normal ($\sigma = 20$) and Zipf ($\lambda = 1.1$) datasets. The number of SPEs is increased from 1 to 8.

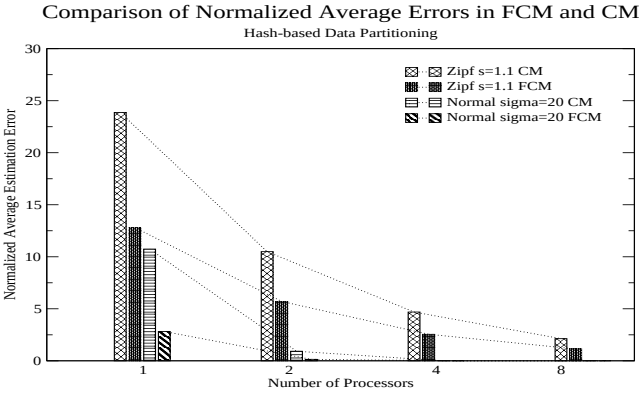

Figure 17: Comparison of errors in FCM and CM using hash-based partitioning as the number of SPEs is increased from 1 to 8. Errors in estimating Normal ($\sigma = 20$) datasets on 4 and 8 SPEs are too small for representation.

using hash-based partitioning (Figure 17), the normalized error decreases significantly as the number of SPEs is increased (for the Normal dataset, after 2 SPEs, the error is too small to be represented on the graph.).

Unlike the hash-based partitioning, the block-based partitioning does not group together items of the same value. Hence, the quality of local estimate is not as high as that for the high-frequency items (i.e., local errors are high). Also, the final result is calculated via aggregating local results over all SPEs in the system. Therefore, the local errors get accumulated, resulting in a significant degradation in estimation quality. Finally, for block-based partitioning, as the number of SPEs is increased, the impact of zero-frequency items on the estimation error increases as each point query is broadcast to all SPEs. FCM's ability to identify zero-frequency items mitigates this impact and unlike the CM algorithm, it's estimate quality does not degrade as the number of SPEs is increased (Figure 16).

Figures 18 and 19 compare the frequency estimation for the hash-based data partitioning while using the FCM algorithm on 1 SPE and 8 SPEs. In both cases, the frequency estimation improves substantially as the number of SPEs is increased to 8. These results experimentally demonstrate the benefits of using hash-based data partitioning as our sequential kernel.

Finally, Figure 20 illustrates the scalability of the parallel algorithm under different runtime configurations. We use the execution time of the algorithm for ingesting 16 MB data stream on a single SPE as a baseline and compute the relative performance for 2, 4, and 8 SPEs. As shown in Figure 20, in all cases, the parallel algorithm demonstrates linear scalability as the number of SPEs was increased. The hash-based data partitioning has very similar performance to the block-based data partitioning. In both cases, the cost of processing a point query was insignificant. We also did not find any significant performance overheads due to the additional PPE-side processing or unbalanced data partitioning across the SPEs.

The results presented in this section conclusively demonstrate that on a single processor, FCM's key features, namely, the frequency-aware sketch processing using the Misra-Gries

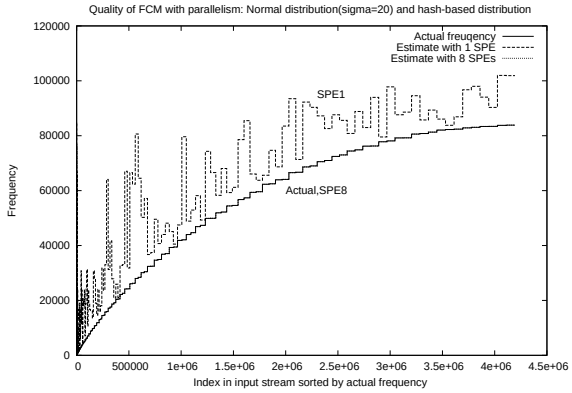

Figure 18: Comparison of the quality of estimation of a Normal ($\sigma = 20$) data using hash-based data partitioning over 1 and 8 SPEs.

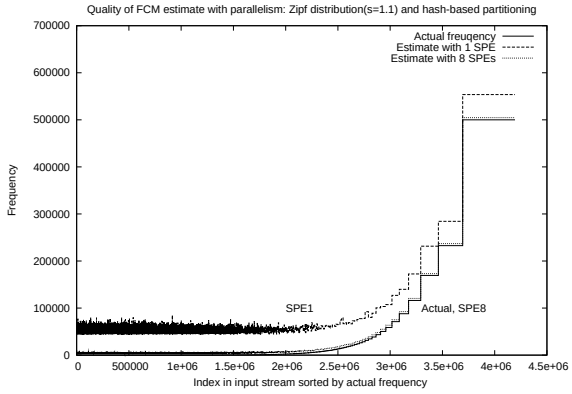

Figure 19: Comparison of the quality of estimation of a Zipf ($\lambda = 1.1$) data using hash-based data partitioning over 1 and 8 SPEs.

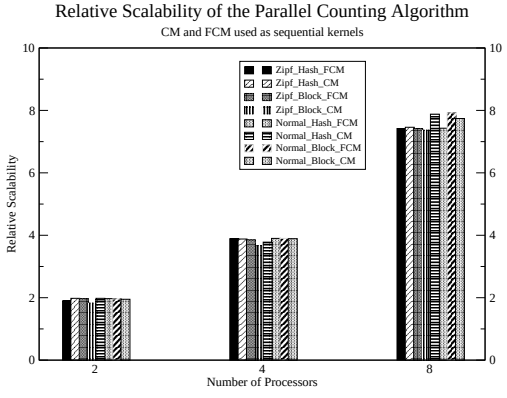

Figure 20: Linear scalability of the Parallel Counting Algorithm as compared to the single SPE execution time for ingesting data stream.

Counter, improved the estimation quality for the entire frequency range. These results also illustrate that simply partitioning data across multiple processors does not lead to an improvement in estimation quality as the number of processors is increased. To achieve performance scalability and estimation improvement, one needs to use hash-based data partitioning, along with the zero-frequency table.

7. RELATED WORK

The problem of synopsis construction has always been considered an important one in the context of data streams. Typical methods for stream synopsis construction include reservoir sampling, histograms, wavelets and sketches. A broad overview of synopsis construction methods in data streams may be found in [14].

The sketch based method was first proposed in [2] as an application of random projection techniques for space-efficient determination of aggregate characteristics of the stream such as frequency moments and frequent items. The broad idea of the original sketch technique is that a stream can be viewed as a very high dimensional vector whose dimensionality is determined by the number of items in the stream, and a random projection [20] of this stream can be used to determine aggregate functions of the frequencies of the individual items. A pseudo-random number generator is used to dynamically generate the components of the random projection vector rather than explicitly storing them [2]. The sketch based method is particularly useful for the space-constrained scenario (such as a cell processor) since its space-requirements are logarithmic in terms of the number of distinct items in the data stream. Subsequently a variety of other sketch based methods were proposed which vary from the random projection idea for sketch computation. These include hash-based structures such as that proposed in [4] (Count Sketch), and the count-min sketch [5]. Some the sketch based structures such as the count-min sketch [5, 6] are particularly suited to tracking frequent items in the streams, whereas others such as random projection based sketches [2] are more suited to aggregate computations such as determining the frequency moments. However, both sketch based structures are generally better suited to estimating the behavior of items with high frequency, and are not particularly well suited to lower frequency items or determining the number of distinct elements. A different kind of sketch based structure has been proposed [12] for counting the number of distinct elements in the stream. Subsequently, sketch based structures have been extended to a variety of other problems such as join estimation [1] and aggregate query estimation [10, 11, 16].

Metwally et al [13] also use a counter data structure to identify top-k items from data streams. However, its space complexity is a function of user-defined error ϵ and the range of the input data, which is significantly larger than our MG counter.

The problem of sketch based processing has also been used for distributed mining of data streams. Sketch based processing is especially useful in distributed sensor network processing [26, 21], in which sensors have limited storage and processing ability. In [26, 21], a number of techniques have been discussed for aggregate computation of the sensor network statistics with the use of sketch based methods. A number of different techniques for distributed sketch based processing have been proposed for effective query process-

ing. For example, the technique in [3] discusses the problem of distributed top-k monitoring, in which the k largest values from a set of distributed data streams are continually maintained. This method is able to achieve this to within a user-specified tolerance. Similarly, the method in [9] discusses the problem of distributed set expression cardinality estimation. In [17], the problem of power conserving computation of order statistics in sensor networks is discussed. One critical difference between the distributed algorithms on sensor networks and that for the cell processor is that in the case of sensor networks, power conservation is the primary criterion for algorithm design, whereas in the case of the cell processor, the algorithms need to be designed with storage and processing methodology as the primary consideration.

8. CONCLUSIONS

In this paper, we investigated the problem of counting items from data streams using multi-core processors that have multiple processing engines with low on-chip memories. We evaluated existing stream counting algorithms and found that they are not suitable for such processors due to their inability to correctly estimate low-frequency and zero-frequency items using low-memory probabilistic data structures. We proposed a new stream counting algorithm called Frequency-aware Counting (FCM) that improves over the Count-Min algorithm using: (1) A space-conserving frequency counter to dynamically predict relative frequency phases, (2) A sketch data structure that is updated using a frequency-aware approach, and (3) A zero-frequency data structure that reduces the estimation errors due to absent items. We used the FCM algorithm as the kernel in our parallel counting algorithm and implemented it over a commercial multi-core processor, the Cell processor. Our experiments validate our key ideas and efficacy of our data structures. The results demonstrate that in a sequential mode, the FCM kernel provides better quality estimates over the CM algorithm both for the high- and low-frequency items. Further, our parallel algorithm improves both performance and quality as the number of processors is scaled up. Although the FCM algorithm is evaluated using the Cell processor, it can be implemented on similar multi-core processors [7, 18]) or using software libraries like Pthreads.

This work could be extended in many directions. It is an open problem to determine the optimal values for d_h and d_l for an input data distribution. Clearly, the hash-based data partitioning used in this study is an example of a generalized value-based partitioning scheme. It will be interesting to evaluate a value-based partitioning scheme which can exploit statistical characteristics of input data streams. It will also be interesting to examine extensions to the Misra-Greis counter to capture more fine-grained frequency phases for computing top-k elements from data streams.

9. REFERENCES

[1] Alon, N., Gibbons, P., Matias, Y., and Szegedy, M. Tracking Joins and Self Joins in Limited Storage. In *ACM PODS Conference* (1999).

[2] Alon, N., Matias, Y., and Szegedy, M. The Space Complexity of Approximating the Frequency Moments. In *ACM Symposium on Theory of Computing* (1996), pp. 20–29.

[3] Babcock, B., and Olston, C. Distributed Top-k Monitoring. In *ACM SIGMOD Conference 2003* (2003), pp. 28–39.

[4] Charikar, M., Chen, K., and Farach-Colton, M. Finding Frequent Items in Data Streams. In *Automata, Languages and Programming, 29th International Colloquium, ICALP 2002* (2002), pp. 693–703.

[5] Cormode, G., and Muthukrishnan, M. An Improved Data Stream Summary: the Count-Min Sketch and its Applications. *Journal of Algorithms 55*, 1 (April 2005).

[6] Cormode, G., and Muthukrishnan, S. What's Hot and What's Not: Tracking Most Frequent Items Dynamically. *ACM Transactions on Database Systems 30*, 1 (March 2005), 249–278.

[7] Intel Corporation. Intel IXP Network Processors. Online Documentation, www.intel.com, 2000.

[8] IBM Corporation. Cell Processor Technical Documentation. IBM Developerworks, 2006.

[9] Das, A., Ganguly, S., Garofalakis, M., and Rastogi, R. Distributed Set-Expression Cardinality Estimation. In 30^{th} *International Conference on Very Large Data Bases* (2004), pp. 312–323.

[10] Dobra, A., Garofalakis, M., Gehrke, J., and Rastogi, R. Processing Complex Aggregate Queries over Data Streams. In *SIGMOD Conference* (2002).

[11] Dobra, A., Garofalakis, M., Gehrke, J., and Rastogi, R. Sketch-based Multi-query Processing over Data Streams. In *EDBT Conference* (2004).

[12] Flajolet, P., and Martin, G. N. Probabilistic Counting Algorithms for Data Base Applications. *Journal of Computer and System Sciences 31*, 2 (1982), 182–209.

[13] Metwally, A., Agrawal, D., and Abbadi, A. E. An Integrated Efficient Solution for Computing Frequent and Top-k Elements in Data Streams. *ACM Transactions on Database Systems 31*, 3 (2006), 1095–1133.

[14] Garofalakis, M., and Gehrke, J. Querying and Mining Data Streams: You only get one look. In *VLDB Conference* (2002).

[15] Gibbons, P. B., and Matias, Y. Synopsis Data Structures for Massive Data Sets. In *Proceedings of Symposium on Discrete Algorithms* (1999).

[16] Gilbert, A., Kotidis, Y., Muthukrishnan, S., and Strauss, M. Surfing Wavelets on Streams: One Pass Summaries for Approximate Aggregate Queries. In *VLDB Conference* (2001).

[17] Greenwald, M., and Khanna, S. Power-conserving Computation of Order-Statistics over Sensor Networks. In *ACM PODS Conference* (2004).

[18] Cisco Systems, Inc. Parallel Express Forwarding on the Cisco 10000 Series. White Paper, 2006.

[19] Mercury Computer Systems, Inc. Cell Broadband Engine: Products Online Documentation, www.mc.com.

[20] Johnson, W., and Lindenstrauss, J. Extensions of Lipshitz Mapping into Hilbert Space. *Comtemporary Mathematics 26* (1984), 189–206.

[21] Kempe, D., Dobra, A., and Gehrke, J. Gossip Based Computation of Aggregate Information. In *ACM PODS Conference* (2004).

[22] Kistler, M., Perrone, M., and Petrini, F. Cell Multiprocessor Interconnection Network: Built for Speed. *IEEE Micro 26*, 3 (2006).

[23] Misra, J., and Gries, D. Finding Repeated Elements. *Science of Computer Programming 2* (1982), 143–152.

[24] Mitzenmacher, M., and Upfal, E. *Probability and Computing*. Cambridge University Press, 2005.

[25] Muthukrishnan, S. Data Streams: Algorithms and Applications. Tech. Report, Computer Sciences Department, Rutgers University, 2005.

[26] Yao, Y., and Gehrke, J. E. Query Processing in Sensor Networks. In *First Biennial Conference on Innovative Data Systems Research (CIDR 2003)* (January 2003).