# CoTS: A Scalable Framework for Parallelizing Frequency Counting over Data Streams*

Sudipto Das Shyam Antony Divyakant Agrawal Amr El Abbadi
Department of Computer Science
University of California, Santa Barbara
Santa Barbara, CA 93106, USA
{sudipto, shyam, agrawal, amr}@cs.ucsb.edu

## ABSTRACT

Applications involving analysis of data streams have gained significant popularity and importance. Frequency counting, frequent elements and top-$k$ queries form a class of operators that are used for a wide range of stream analysis applications. In spite of the abundance of these algorithms, all known techniques for answering data stream queries are sequential in nature. The imminent ubiquity of Chip Multi-Processor (CMP) architectures requires algorithms that can exploit the parallelism of such architectures. In this paper, we first explore the challenges in parallelizing frequent elements and top-$k$ queries in the context of the inherent parallelism available in multi-core processors, evaluate different naive techniques for *intra-operator* parallelism, and summarize the insights obtained from the different parallelization efforts. Our experimental analysis of the naive designs implemented in the paper shows that *intra-operator* parallelism is not straightforward and requires a complete redesign of the system. Based on the lessons learnt from this analysis, we design an efficient and scalable framework for parallelizing frequency counting, frequent elements and top-$k$ queries over data streams. The proposed *CoTS (Co-operative Thread Scheduling)* framework is based on the principle of threads *co-operating* rather than *contending*. Our experiments on a state-of-the-art quad-core chip multiprocessor architecture and synthetic data sets demonstrate the scalability of the proposed framework, and the efficiency is demonstrated by peak throughput of more that 60 million elements per second. In addition, for skewed data distributions, despite using heavy weight synchronization primitives, the implementation of the proposed framework outperforms the sequential implementation by a factor of 2–4X.

## Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous.

## General Terms

Parallelism, Stream Algorithms, Design, Performance.

## Keywords

Data Streams, Frequent elements queries, Top-$k$ queries, *intra-operator* parallelism, Chip Multi Processor.

*This work is partly supported by NSF Grants IIS-0744539, IIS-0223022 and CNS-0423336

## 1. INTRODUCTION

Data stream analysis forms an important class of applications where the data is streaming in, and processing has to be done in real time [2]. An important distinction of data stream applications compared to database applications is that stream algorithms can only make a single pass through the stream of tuples and since the stream can be potentially infinite, only a summary of the stream is stored instead of the entire stream. Analysis of click streams in internet advertising, and network monitoring for detecting malicious activities are examples of stream processing and its requirements. In Internet advertising, for determining the *effective Cost Per Click* (CPC) for an advertisement and in turn deciding which advertisements to display, a *publisher* (web site hosting the advertisements) needs to have an estimate of the number of *impressions* (the number of times an advertisement is rendered on the web page), and the *Click Through Rate* (i.e. the number of times the advertisement was clicked). This analysis requires real time frequency counting on the stream of clicks seen by the *publisher*. Irrespective of whether an exact or an approximate estimation is sought, the answers need to be updated online. An *advertising commissioner* also needs to keep track of the number of *clicks* and *impressions* and is also responsible to detect any fraud by the *publisher* or the *advertiser*. Again, this requires real-time processing of the streams for accounting purposes and for detection of frequent patterns and correlations.

Frequent elements [15, 9, 16] and top-$k$ [7, 17] queries are an important class of queries for stream analysis applications, and the research community has proposed several algorithms for answering these queries efficiently. A frequent elements query returns all the elements whose frequency of occurrence is above a certain threshold. For example, a query of the form "advertisements that are clicked more than 0.1% of the total clicks" is a frequent elements query. On the other hand, a top-$k$ query returns the $k$ elements with the highest frequency. Again, a query of the form "top-25 most clicked advertisements" is a top-$k$ query.

Even though numerous algorithms have been proposed in the literature to answer these queries, all the proposed algorithms are serial in nature. However, processor architectures have seen a recent shift in design where a single processor now consists of multiple cores, which can execute instructions in parallel. The ubiquitous presence of these processors in almost all commodity as well as high-end computers necessitates the algorithms to be concurrent, so that multiple threads executing in parallel can effectively exploit the

available parallelism. In addition to *inter-operator* parallelism (or scheduling as in [1]), where multiple operators execute independently and in parallel on different cores, *intra-operator* parallelism – where a single operator aims to utilize the available cores to improve the throughput of processing – is also important for long standing queries operating on huge amounts of data. Data stream queries are typical examples of these long standing queries and are thus candidates for possible *intra-operator* parallelism. It must be realized that frequency counting over the stream forms the basis for answering frequent elements and top-$k$ queries as in [15, 9, 16]. So these queries can be considered as queries ensuing the frequency counting operator. Therefore, for making query answering scalable, scalability of frequency counting is the primary goal.

In this paper, we explore and thoroughly analyze the challenges in *intra-operator* parallelism of frequency counting over data stream, in the context of the inherent parallelism available in multi-core processors. Even though it might seem that parallelizing these operators would be straightforward, our analysis and evaluation of the naive parallelization schemes reveal that these designs are not scalable and efficient. Based on the insights gained from this analysis, we design a scalable and efficient framework for parallelization of these operators. The framework is referred to as *CoTS (Cooperative Thread Scheduling)* and is based on the principle of threads *cooperating* rather than *contending* for shared resources. This framework is general enough to accommodate different frequency counting algorithms where the element's frequency increases monotonically. In this paper, we adapt the *Space Saving* algorithm [16] and our experiments on a state-of-the-art quad-core chip-multiprocessor architecture demonstrate the scalability of the proposed framework, and the efficiency is established by peak throughput of more that 60 million elements per second.

The main contributions of the paper are as follow:

- This is the first work exploring *intra-operator* parallelism of data stream operators in the context of the parallelism offered by multi-core architectures. We analyze and identify the challenges for *intra-operator* parallelism of frequency counting, frequent elements and top-$k$ queries over data streams.
- We propose *CoTS*, an efficient and scalable framework for parallel frequency counting of stream elements. We implement the *Space Saving* [16] algorithm in the *CoTS* framework and our experiments demonstrate scalability and efficiency. Despite a not very scalable *summary* structure and a not very easily parallelizable problem, the framework shows good scalability and efficient performance for skewed data.
- We further analyze and identify the *system-level* and *design-level* challenges that need to be addressed for further improving the performance of parallel algorithms and to efficiently exploit the parallelism of modern processor architectures.

The rest of the paper is organized as follows: Section 2 summarizes related work, and Section 3 explains the details of the *Space Saving* algorithm and the modern processor architectures and formalizes the query model which the system supports. Section 4 analyzes the naive parallelization schemes, experimentally evaluates them and analyzes the challenges in *intra-operator* parallelism. Section 5 explains the proposed *CoTS* framework and how the *Space Saving* algorithm can be adapted into the framework, Section 6 provides experimental evaluation and analyzes the different challenges that still remain unsolved and must be addressed to efficiently exploiting the parallelism of modern processor architectures, and Section 7 concludes the paper.

## 2. RELATED WORK

Frequent elements and top-$k$ queries constitute an important class of queries for stream analysis applications and numerous algorithms have been proposed in the literature for answering such queries. The algorithms for answering frequent element queries are broadly divided into two categories: *sketch based* and *counter based*. The *sketch based* techniques such as [3, 6] try to represent the entire stream's information as a "sketch" which is maintained and updated as the elements are processed. Since the "sketch" does not store per element information, the error bounds of these techniques are not very stringent. In addition, these techniques generally process each stream element using a series of hash functions, and hence the processing cost per element is also high. Even though these techniques can answer frequent elements queries, these are not very well suited for the class of applications that require frequency counting.

On the other hand, the *counter based* techniques such as [16, 15, 9] monitor a subset of the stream elements and maintain an approximate frequency count of the elements. Different approaches use different heuristics to determine the set of elements to be monitored and to limit the amount of space. The goal is to give high accuracy with a small memory footprint. For example, in *Lossy Counting* [15], the stream is divided into rounds, and at the end of every round potentially non-frequent elements are deleted. This $\epsilon$-approximate algorithm has a space bound of $O(\frac{1}{\epsilon}\log(\epsilon N))$, where $N$ is the length of the stream. The *Space Saving* algorithm [16], on the other hand, uses a different heuristic to limit space and details are provided in Section 3.3. Cormode et al. [5] provide an experimental evaluation of the different algorithms for finding the frequent elements on a stream.

Different solutions have also been suggested for answering top-$k$ queries [7, 17]. Mouratidis et al. [17] suggest the use of geometrical properties to determine the $k$-skyband and use this abstraction to answer top-$k$ queries, whereas Das et al. [7] propose a technique which is capable of answering ad-hoc top-$k$ queries, i.e., the algorithm does not need *apriori* knowledge of the attribute on which the top-$k$ queries have to be answered.

With the growing data rates and faster processing speed requirements, researchers are also striving for accelerating these queries. For example in [8] Content Addressable Memories (CAM) have been used for accelerating frequent elements and top-$k$ queries. The constant time lookups of CAM is leveraged to accelerate counter based techniques. Similarly, other novel architectures have also been explored to accelerate the processing of elements. In [12], graphics processors have been used for accelerating different data management operations, while in [10] the parallelism of a cell broadband engine is exploited for acceleration.

The advent of modern Chip Multiprocessor architectures [18, 13] have opened new frontiers and their ubiquitous presence calls for algorithms that can efficiently exploit the parallelism offered by these architectures. Although much re-

search has been done in the database arena for exploiting the parallelism [4, 11], very little or no research has focussed on stream operators. In this paper we analyze the challenges in *intra-operator* parallelism of frequent elements and top-$k$ queries and propose a scalable and efficient framework for parallelizing these stream operators.

# 3. BACKGROUND

## 3.1 Chip Multi-Processor Architectures

Over the last decade, the transistor density of a chip has increased considerably and processor designers utilized the increased density coupled with increased the clock speed to increase processor performance. But as the transistor sizes become smaller and smaller, this processor design paradigm has hit physical limitations of heat dissipation, leakage current and so forth. As a result, there has been a paradigm shift in processor design, and now chip manufacturers are packing more execution units into the same die. Each of these cores do not operate at frequencies as high as the traditional single core CPUs, but these multiple execution units can execute instruction in parallel and thus provide a huge amount of parallelism, the rationale being that the performance advantage will be realized via parallelism instead of faster clock speeds. These processors are broadly categorized as "Multi-core processors" and examples include Cell Broadband Engine [10], and Chip Multi-Processors (CMP) [11]. CMP is a class of multi-core processors that have been designed for general purpose computing, and the popularity of these processors is evident from their ubiquitous presence in all modern low-end as well as high-end computers. CMPs can be further classified into two broad categories, which the authors in [11] term as "Lean Camp" and "Fat Camp" processors. This categorization is based on the architectural differences of the processors and the type of workload for which they are suited.

*Lean Camp*: These processors (e.g. Sun UltraSPARC $T2$ [18]) are characterized by simple cores which support many "hardware thread contexts" for high Thread Level Parallelism (TLP). It must be noted that "hardware" threads are different from conventional "software threads" as the hardware threads are equivalent to independent CPU's that can execute in parallel. Each hardware thread has its own set of registers and local resources, and sharing the execution unit between hardware threads does not require a costly context switch. Since the cores are simpler, a large number of cores/hardware threads can fit on a single chip, but the clock frequencies are low (about 1.2 – 1.4 GHz). For example, the Sun UltraSPARC $T2$ processor (Niagara2) contains 8 cores and each core contains 8 hardware threads, which gives it a total of 64 hardware threads.

*Fat Camp*: These processors (e.g. Intel Quad Core [13]) are characterized by relatively complex cores that employ wide-issue, out of order execution and aggressive speculation for Instruction Level Parallelism (ILP) to optimize single thread performance. Since each core involves complex logic and design, the core sizes are relatively larger and thus there are lesser number of cores/hardware contexts on a single chip. For example, the Intel Core2 Quad series of processors have 4 cores on a single processor, and each core operates at about 2–3 GHz.

To summarize, the "lean camp" processors provide high TLP and low ILP and are suitable for workloads where there is enough computation to be performed to efficiently hide I/O access latencies, while the "fat camp" processors are characterized by high ILP and low TLP and are suitable for workloads which are more compute intensive.

## 3.2 Query Model

In this section, we define the queries to be supported by the system. The queries can vary based on the type of answers sought (Queries 1, 2) or the frequency at which the queries need to be answered (Queries 3, 4).

QUERY 1. POINT QUERY: *This type of query is interested in a single element and is a boolean query of the form* IsElementFrequent(e) *or* IsElementInTopk(e).

QUERY 2. SET QUERY: *These queries report all the elements that are frequent, or all elements that are in the top-$k$. A frequent elements set query can be expressed formally in a language similar to SQL as:*

```
Select S.element
From Stream S
Where IsElementFrequent(S.element)
```

Even though a set query can be visualized as a combination of point queries for all the elements in the input alphabet, more efficient techniques can be employed provided the elements are sorted by their frequency.

QUERY 3. INTERVAL/DISCRETE QUERY: *These queries are posed as independent queries and consecutive queries are spaced out either with respect to time or the number of updates. A frequent elements interval set query can be expressed formally in a language similar to SQL as:*

```
Select S.element
From Stream S
Where IsElementFrequent(S.element)
Every 0.001s
```

QUERY 4. CONTINUOUS QUERY: *These queries are posed with "every update", i.e., as soon as a stream element is processed, the answer should be updated.*

When the stream elements are processed in parallel, the notion of "every update" is not as clear as in sequential processing of stream elements. If the sequential continuous query is mapped into the parallel processing scenario, there will be multiple concurrent queries at a particular instant, and the result from one query will be immediately updated by the next result. Most of the applications require the answer sets to be updated periodically and therefore, we only consider "Interval/Discrete" queries which can either be "point" or "set" queries.

## 3.3 Space Saving

The *Space Saving* algorithm provides an elegant technique for frequency counting on a stream of elements. An interesting property of the algorithm is that it is deterministic and provides tight space bounds corresponding to the user specified error bound. *Space Saving* monitors only $O(\frac{1}{\epsilon})$ counters for providing answers within error bound of $\epsilon$. Algorithm 1 gives an overview of *Space Saving*. The main operations performed by the algorithm have been tabulated

**Algorithm 1** *Space Saving* algorithm

```
for each element ⟨e⟩ in the stream do
  /*Check if already being monitored*/
  if (LOOKUP(⟨e⟩)) then
    IncrementCounter(⟨e⟩)
  else
    if (numCountersMonitored < maxCounters) then
      e→frequency ← 1
      AddElementToBucket(minFreq, e)
    else
      Overwrite(minFreq, e)
    end if
  end if
end for
```

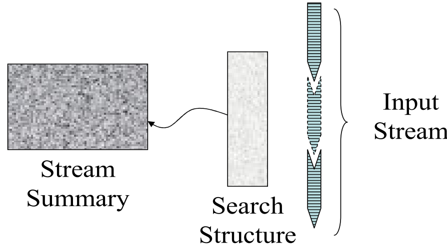


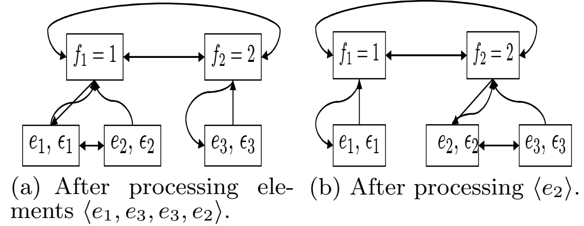

**Figure 1: System Design for Space Saving.**

in Table 1. The algorithm monitors a subset of the stream elements (*Monitored Set*). If the element being processed is already being monitored, then its count is incremented (*IncrementCounter*). Otherwise, if the number of elements is less than the maximum bound, then the element is added to the monitored set (*AddElementToBucket*), else the current element overwrites the element with minimum frequency (*Overwrite*). Since an element is overwritten only when the upper bound of space is reached, if the alphabet is small, the algorithm can give exact counts. Thus, the space bound of the algorithm is $min(O(\frac{1}{\epsilon}, |A|))$, where $|A|$ is the size of the alphabet of the stream, and $\epsilon$ is the user specified error bound.

For overwriting, this algorithm needs to have knowledge of the minimum frequency element. The *Stream Summary* structure [9, 16] is used for maintaining the minimum frequency element. The *Stream Summary* structure consists of a doubly-linked list of frequency buckets which are sorted by frequency. Each bucket contains a list of elements which has the same frequency as that of the bucket. A nice property of this structure is that it maintains the elements sorted by frequency and in constant time per element. Figure 2 demonstrates this for an example stream. For lookup, the algorithm needs to have an efficient *search structure* that can be integrated with the *Stream Summary* structure. Figure 1 provides an abstract representation of the overall structure of the *Space Saving* algorithm. The algorithm looks up an element in the *Search Structure* (LOOKUP), and then moves to the *Stream Summary* structure to update the element. Since the elements are sorted by their frequency of occurrence, the *Stream Summary* structure can also be used to efficiently answer both frequent elements and top-$k$ queries. The *Monitored Set* is thus represented by a combination of *Search Structure* and *Stream Summary*.

**Table 1: Main Operations in Space Saving**

| Operation | Description |
|---|---|
| LOOKUP($e$) | Check whether element $e$ is being monitored |
| *IncrementCounter*($e$) | Increment the frequency of $e$ |
| *AddElementToBucket*( $freq$, $e$) | Add an element $e$ to bucket with frequency $freq$ |
| *Overwrite*($minFreq$, $e$) | Overwrite the minimum frequency element with $e$ |




(a) After processing elements $\langle e_1, e_3, e_3, e_2 \rangle$. (b) After processing $\langle e_2 \rangle$.

**Figure 2: This figure illustrates the *Stream Summary* data structure for an example stream of elements $\langle e_1, e_3, e_3, e_2, e_2 \rangle$. The elements can be kept sorted in constant time per element.**

# 4. NAIVE PARALLELIZATION SCHEMES

In this section, we analyze the different naive schemes for parallelizing the *Space Saving* algorithm. From the description of the algorithm in Section 3.3, it is evident that the *Monitored Set* is the point of interest. Referring back to Figure 1, the *Search Structure* can be efficiently implemented using a hash table, and the sort-order is maintained using *Stream Summary*. The design of the system depends on how the threads share these counters, but each design needs to implement the set of operations listed in Table 1. We now discuss two approaches for naive parallelization, namely *Independent Structures* and *Shared Structure*.

## 4.1 Independent Structures

This design corresponds to the *shared nothing* paradigm, where the threads do not share any data or state information. The idea is to simulate sequential execution, and run multiple copies of the same algorithm executing on different partitions of data and operating on local structures. Each thread has a local copy of the combination of *Stream Summary* and the *Search Structure*. These local structures need to be merged into a global structure so that queries can be answered from the global structure. The *Space Saving* algorithm has two parts, the frequency counting part, which counts the number of occurrences of an element, and the query part, where the frequency counts are used to answer the queries. In this design, even though the frequency counting part can execute in parallel, the local structures need to be merged to answer queries, and the frequency of merging the counters depends on the query frequency required by the application. As the number of parallel threads increases, the cost of frequency counting decreases, but the cost of merging increases. The merge can be performed using two different approaches. In *Serial Merge* a single thread merges all the structures, while *Hierarchical Merge* refers to a parallel merge similar to the merge phase of the *Merge*

*Sort* algorithm.

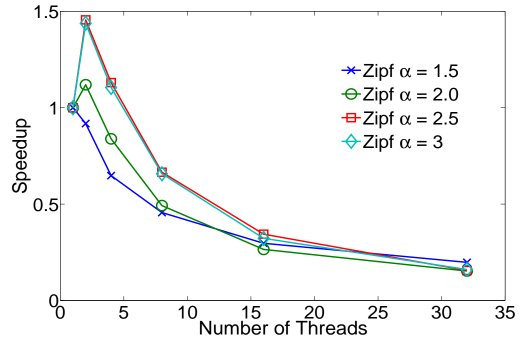
## 4.2 Shared Structure

This design corresponds to the other extreme where all the threads share a common *Stream Summary* structure. Since multiple threads are accessing the same structure, the threads must be synchronized. Synchronization is achieved using locks and atomic operations supported by the underlying architecture, and this synchronization needs to be done at two levels:

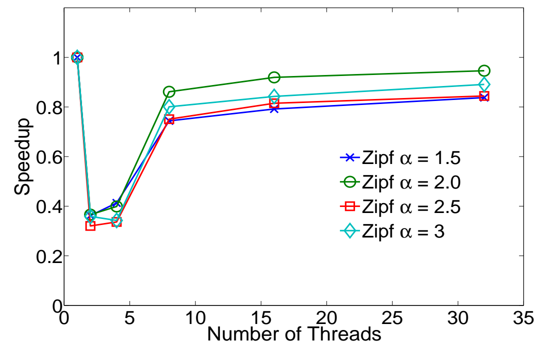
**Element Level Synchronization**: Multiple threads operating on the same element must be serialized so that there is only one thread inside *Stream Summary* that is operating on the element.

**Bucket Level Synchronization**: Since an increment or overwrite operation needs to move an element from one frequency bucket to another, a thread that is performing this operation needs to obtain a lock on the source and the destination bucket (for example, in Figure 2(b), when $e_2$ is promoted from bucket $f_1$ to $f_2$, the thread must acquire locks on both these buckets). Since there can be several elements within a frequency bucket, a lock on a bucket prevents other threads from operating on any element belonging to that bucket. So this bucket-level locking serializes accesses to a frequency bucket.

The pointer to the minimum frequency bucket should also be protected by a lock. The queries (which are only readers) also need to obtain locks so that the writes are blocked while a reader is inside a bucket. Additionally, in the original sequential design, the queries traverse the list of frequency buckets from maximum frequency towards minimum frequency, while the updates would traverse the list in the opposite direction. Therefore, additional locks need to be obtained to synchronize the queries with updates traversing the structure in the opposite direction.




(a) Independent Structures with a query every 50000 elements.




(b) Shared Structure with synchronization using Pthread mutex.

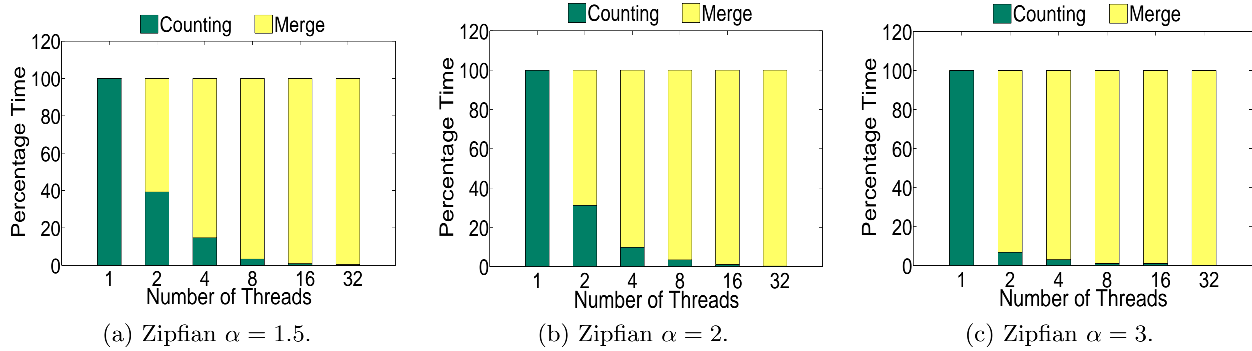
**Figure 3: Evaluation of the naive parallelization techniques for a stream of 5 million elements.**

## 4.3 Analysis of Naive Techniques

In this section, we experimentally evaluate the naive parallelization schemes and analyze their performance. The experiments were performed on an Intel Quad Core processor and the data set used is a synthetic zipfian data set. More details about the experimental set up can be found in Section 6. Figure 3 shows the results for a data set of 5 million elements. The number of threads that are concurrently processing the data is plotted along the x-axis while the speedup obtained compared to the execution time of a single thread is plotted along the y-axis. The different lines in the graphs correspond to different zipfian factors $\alpha$.

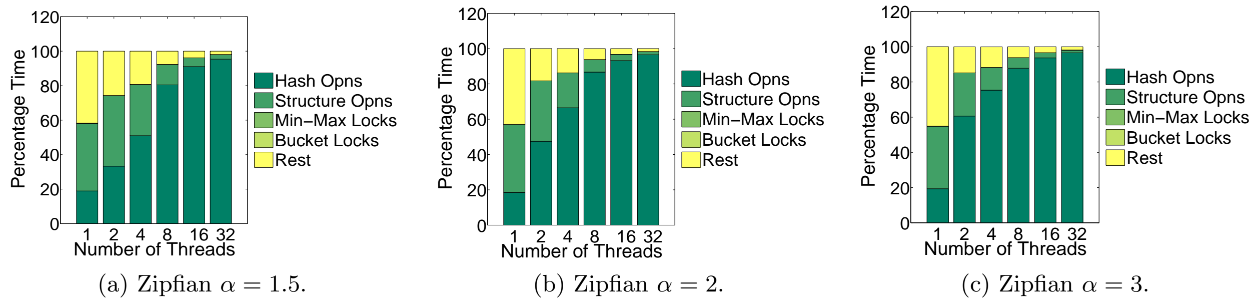
From Figure 3(a), it can be seen that the independent design does not scale as the number of threads increase. Figure 3(a) plots the results for a query every 50000 updates, and the scalability will be worse if the query frequency increases. This figure shows results for serial merge invoked every 50000 elements processed; hierarchical merge also shows a similar pattern, and the actual cost of merge varies as the number of monitored counters varies. Even though it seems that hierarchical merge should perform better, in practice it does not because of the overhead of threads synchronizing at the end of merge at each level. The reason for poor scalability would be evident from the break-up of where the time is spent by the algorithm, and Figure 4 shows a break-up of the time taken for the different sub-parts of the *Space Saving* algorithm, i.e. the frequency counting part (represented by "Counting" in the figures) and the merging part (represented by "Merge" in the figures) where the individual structures are merged to find the final result. The y-axis shows the percentage of the total time spent for the operation, and the different bars correspond to different number of threads processing the input. The number of threads is plotted corresponding to the bar representing that particular run. In these experiments, one query (and thus a merge) was executed every 50000 elements. The different sub figures correspond to different values of zipfian $\alpha$. As is evident from the figures, even though the frequency counting part scales very well and takes lesser amount of time with increase in number of threads, the counters need to be merged periodically to answer the queries, and as the number of threads increases, the merge cost increases considerably. The merge cost would increase further if the merges become frequent, i.e. if the query frequency is high, and if the number of elements being monitored increases. In addition to the merge overhead, it must also be noted that independent structures would incur a high space overhead due to local repetition of the structure.

From Figure 3(b), it can be seen that the shared design also does not scale. Even though the shared design does not have the overhead of merges, but since the threads share a common structure, the synchronization overhead is pretty high. In addition, as pointed out in Section 4.2, there are two levels of synchronization (the element level and the bucket level), and even if there are multiple threads, they are seri-




(a) Zipfian $\alpha = 1.5$. (b) Zipfian $\alpha = 2$. (c) Zipfian $\alpha = 3$.

**Figure 4: Profiling of the technique using *Independent Structures* for a stream of 5 million elements and varying Zipfian $\alpha$ with query every 50000 elements.**

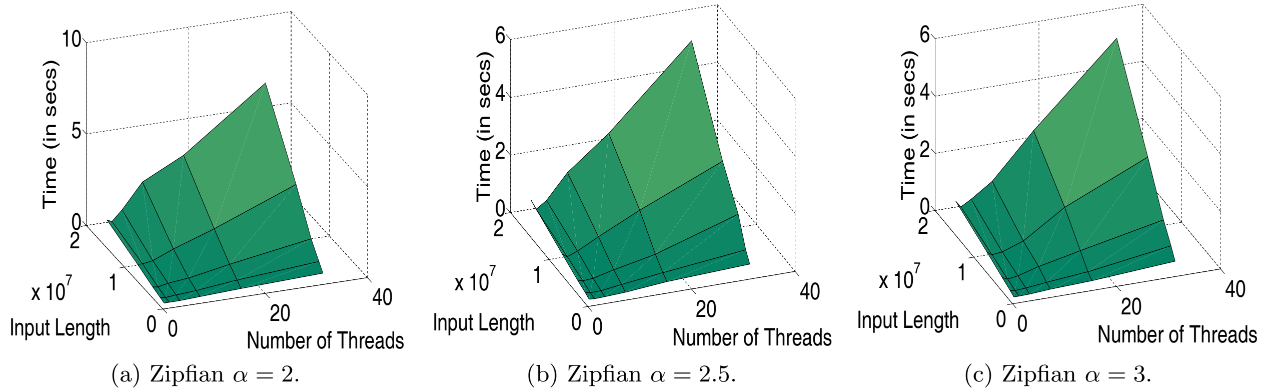



(a) Zipfian $\alpha = 1.5$. (b) Zipfian $\alpha = 2$. (c) Zipfian $\alpha = 3$.

**Figure 5: Profiling of the technique using *Shared Structures* for a stream of 5 million elements and varying Zipfian $\alpha$.**

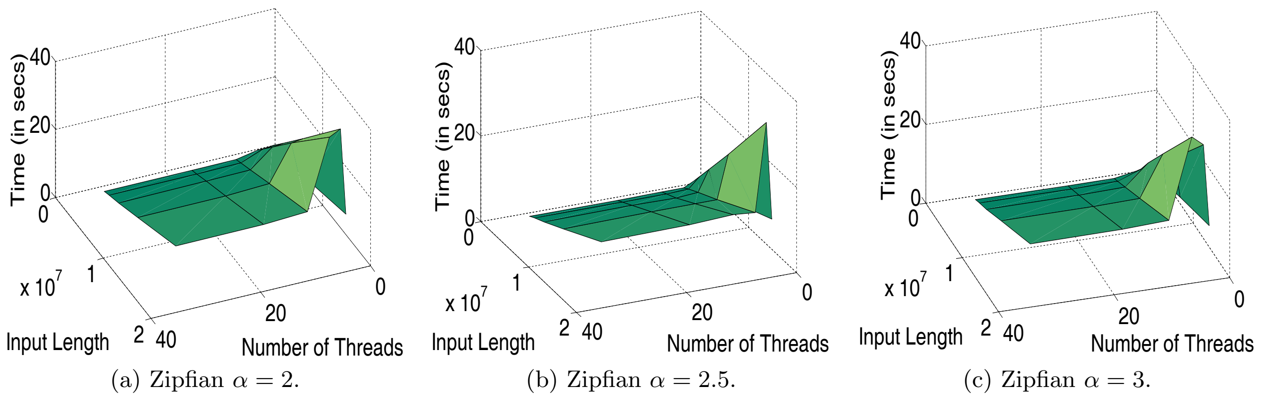
alized at these contention points. Figure 5 shows a break-up of where the time in spent by the *Space Saving* algorithm using a shared structure. The y-axis shows the percentage of the total time spent for the operation, and the different bars correspond to different number of threads processing the input. The number of threads is plotted corresponding to the bar representing that particular run. The different sub figures correspond to different values of zipfian $\alpha$. "Hash Opns" represents the time taken to complete the hash table operations and this includes the time when a thread blocks for an element while some other thread is processing the same element. "Structure Opns" is the time taken to complete the different operations in the *Stream Summary* structure, and correspond to adding a new element to the structure, incrementing an existing element or overwriting a minimum frequency element. For all these operations, the threads need to contend for locks on the shared resources. As pointed out in Section 4.2, the pointers to the minimum frequency and maximum frequency buckets need to be protected by locks, and "Min-Max Locks" represents the time taken to acquire these locks whenever needed. "Bucket Locks" represents the time taken to acquire a lock on a frequency bucket, other than the cases of the structure operations, and "Rest" corresponds to time for all the remaining operations performed by the algorithm.

As is evident from Figure 5, with the increase in number of threads, a high percentage of the time is spent for the hash table operations because threads are blocked in the hash table as some other thread is operating on the same element (element level synchronization). With increasing number of threads, the growth rate of time spent on hash operations is higher for more skewed data sets (Figure 5(c)) compared to somewhat lesser skewed data (Figure 5(a)). This is because for skewed data sets, the same element reappears frequently in the stream and hence multiple threads try to concurrently process the same element, therefore being serialized. Thus due to the element level synchronization requirement, adding more threads does not improve performance for skewed data sets. When the threads are not blocked in the hash table, they concurrently access the *Stream Summary* and again threads accessing the same bucket are serialized there due to bucket level synchronization. This a evident from the high percentage of time spent for structure operations and for lesser skew, more time is spent for the structure operations. From these figures, it is evident that high contention for shared resources is the reason preventing improved throughput of processing with more threads added to the system. The performance was worse with *Spin Locks* (busy-wait) as not only were the threads waiting for shared resources, they were busy-waiting, and hence were also contending for the CPU. An important point to be noted from Figure 3(b) is that the performance of the system degrades when the number of threads is increased from 1 to 4, and beyond that, the performance remain almost steady. Since the processor used for the experiments has 4 cores, only 4 threads can run in parallel, so in 1 – 4 threads, the threads were operating really in parallel, and the effect of contention is evident. Beyond 4, the threads share CPU time and this




(a) Zipfian $\alpha = 2$. (b) Zipfian $\alpha = 2.5$. (c) Zipfian $\alpha = 3$.

**Figure 6: Time taken using independent structures with varying input sizes and queries every 50000 elements. As the number of threads increases, the execution time also increases. The increase is even more noticeable for larger inputs.**




(a) Zipfian $\alpha = 2$. (b) Zipfian $\alpha = 2.5$. (c) Zipfian $\alpha = 3$.

**Figure 7: Time taken using shared structures with varying input sizes. Note that the horizontal axes are flipped in this figure for better understanding and visibility.**

limits the amount of contention in the system and the execution time depends on how the thread executions are interleaved.

In the last experiment, we evaluate the scalability of the naive algorithms with varying input length. Figures 6 and 7 show the plots for the independent and the shared designs and different sub figures correspond to different values of zipfian $\alpha$. The input size was varied from 1 million to 16 million elements and the number of threads was varied from 1 to 32. In both Figures 6 and 7, along one horizontal axis we vary the input size, along the other horizontal axis we vary the number of threads, and the vertical axis represents the execution time in seconds. The time reported is the average time for completion of each thread, and is further averaged over multiple repeated runs. The zipfian $\alpha$ was varied from 2.0 to 3.0 in steps of 0.5. Even though we do not report the times for $\alpha = 1.5$, since frequent elements and top-$k$ queries are generally targeted towards skewed distributions, they also show a similar trend.

As is evident from Figure 6, for independent structures as the input size increases, more merges are required (since the query is posed every 50000 elements) and this prevents scalings, and the effect of adding more threads is worse with larger input sizes. Figure 7 shows similar trends as Figure 3(b) and with increase in the length of the input, the time increases almost linearly, but even with large inputs, there is no improvement in the scalability.

## 4.4 Need for Re-designing the System

As the analysis in Section 4.3 reveals, straightforward parallelization schemes do not scale well. Ideally, we would like a system that has the good properties of the shared design (small memory footprint and no merge overhead) and the independent design (no contention for locks). In a shared structure, *bucket level synchronization* (Section 4.2) serializes all threads accessing the same bucket, and things get worse when there are many overwrites, as all threads are trying to find the minimum frequency element and hence are serialized at the minimum frequency bucket. This is the limitation imposed by the *Stream Summary* structure. But if two threads are operating on different buckets, then they can operate in parallel. Again, *element level synchronization* (Section 4.2) prevents parallel processing in a skewed stream where multiple threads are processing the same element and hence are serialized.

One possible extension can be to maintain a combination of local and global counters (i.e. a **Hybrid** Structure) to limit the contention (by hitting local counters frequently)

as well as space overhead (no need to replicate relatively infrequent elements). This design would not be scalable as well because on the two extremes of the input distribution (relatively uniform and relatively skewed), this technique would degenerate into one or the other parent technique explored here. Therefore, the conventional locking models or models of parallelizing operations using the shared-nothing paradigm does not scale and a somewhat unconventional approach is needed to effectively leverage the parallelism inherent in modern processors.

Based on the insights from this analysis, we design the *CoTS* framework that has the good properties of shared design (small memory footprint and no merge overhead), and reduces the amount of contention by having the threads *cooperate* rather than *contend*. Since the *CoTS* framework uses the shared structure, the two levels of synchronization mentioned in Section 4.2 still remain and the framework provides a scheme to reduce this synchronization overhead.

## 5. COOPERATIVE THREAD SCHEDULING

In this section, we provide details of the *Cooperative Thread Scheduling (CoTS)* framework. As observed in Section 4.3, instead of the sibling threads contending with each other for shared resources, as they are part of the same system, they can rather *cooperate* with each other thereby boosting each other and in turn boosting the overall system performance. In formal terms, when two threads are contending for a shared resource, only one of the threads will acquire the resource. Instead of the other thread waiting for the resource to be available, this thread can "log" its request and leave, provided that it is guaranteed that the thread which is currently holding the resource will eventually complete the "logged" request before relinquishing control over the resource. The manner in which the request is "logged" varies on where the request is being delegated, and details are provided in Section 5.2. This design principle can be easily generalized to a system with multiple threads leading to the following design principles:

**Delegation Model**: If $Thread_i$ is trying to acquire a shared resource $R$ and it succeeds in acquiring that resource, it will go ahead and complete its job using the resource. If it fails to acquire a resource, it will "delegate" its request to the thread that is currently holding exclusive access for $R$, and move to the next request. All threads trying to acquire $R$ will delegate their requests to $Thread_i$. Once $Thread_i$ finishes its own request, and before it relinquishes control over $R$, it will check for any pending requests on $R$ and will relinquish $R$ only when all pending requests have been processed. The implementation should ensure that no request is lost after it has been "logged" or "delegated".

**Minimal Existence**: Once a thread has acquired a resource, live minimally by abstaining from "blocking" for any other shared resource, thereby allowing it to make unhindered progress. As we will see later, a thread will never block on any resource and the *Delegation Model* is used to obviate the need for acquiring multiple shared resources.

The above principles of *Delegation* and *Minimal Existence* rely on the *thread cooperation*. There are multiple benefits of this approach:

*Remove Waits*: The threads do not wait for any shared resource and waste useful computational resources either spinning for the lock or sleeping. In this model, the thread that is deprived of the resource can "log" the request and proceed to process the next request.

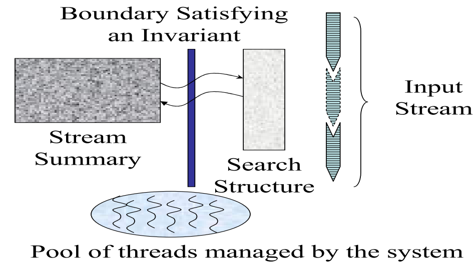



**Figure 8: Overview of the system using the model of cooperating threads.**

*Remove Overhead of Arbitration*: Since the threads do not contend for resources, this technique would save the overhead of arbitration of the locks among the contending threads, the cost of which varies based on the implementation of the locks.

### 5.1 System Overview

Figure 8 provides a high level overview of the system architecture of the co-operative thread model. The system design is very similar to sequential *Space Saving* (Figure 1). As earlier, the system has a stream of elements which is being processed, and the *Stream Summary* structure is updated to reflect the stream element that was processed. The boundary conceptually separates the *Stream Summary* and the *Search Structure*, and these structures interact with each other through a well-defined interface and need not be aware of the implementation details. In this concurrent processing model of the stream, the system should guarantee that the following invariant holds:

INVARIANT 5.1. *If $Thread_i$ processing element $\langle e \rangle$ has crossed the boundary into the* Stream Summary *structure, then it is the* only *thread active in the* Stream Summary *that is processing the element $\langle e \rangle$.*

The *Search Structure* should guarantee that Invariant 5.1 holds and this provides the element level synchronization as discussed in Section 4.2. The *Stream Summary* structure can be optimized with the knowledge of Invariant 5.1. This framework is independent of the choice of different structures involved and the actual algorithm used for processing the stream elements, as long as the desired properties and the invariant holds. The system has at its disposal a *Thread Pool* from where it can select threads to perform different tasks as required by the system. Depending on the amount of parallelism supported by the *Stream Summary* structure and the data distribution, the system can adaptively wake-up threads from *Thread Pool* and return them to the pool when they are not required. Ideally, the system should try to determine the number of threads required to optimize the throughput of the system depending on the characteristics of the data stream being processed.

Thus the problem of parallelization of a stream operator can be viewed as a scheduling problem, where the threads are scheduled in a way that utilizes the maximum parallelism allowed by the underlying structures and input data, with the goal of optimizing performance and resource utilization

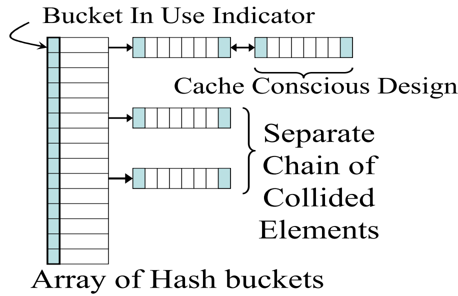



**Figure 9: Cache conscious hash table design using separate chaining.**

**Algorithm 2** *Hash table Delegation*

**Procedure** *Delegate*($\langle e \rangle$)
entry ← `LOOKUP`($\langle e \rangle$)
newlyInserted ← **false**
**if** (entry = **NULL**) **then**
  entry ← `INSERT`($\langle e \rangle$)
  newlyInserted ← **true**
**end if**
result ← `ATOMIC_INC_AND_FETCH`(entry→count)
**if** (result = 1) **then**
  /*No other thread is processing this element.*/
  *CrossBoundary*(entry, newlyInserted)
**end if**
**end Procedure** *Delegate*

of the system. The following sections demonstrate how this general framework applies to the *Space Saving* algorithm.

## 5.2 Adapting Space Saving

As depicted in Figure 8, the system can be viewed as two different components which interact with each other through an interface that ensures Invariant 5.1 holds. It must be recalled than when processing the elements, the algorithm first accesses the *Search Structure* and this structure directs the algorithm to the appropriate place in the *Stream Summary*. In the implementation, the hash table (search structure) points to the element in the *Stream Summary* structure, and the element in turn points to the bucket to which it belongs. The two different components of the system are now discussed in detail.

### 5.2.1 Search Structure

As pointed out earlier in Section 4, a hash table is an appropriate choice for the *Search Structure*. The hash table lookup is equivalent to the `LOOKUP` operation in Table 1.

*Thread Safe Cache Conscious Chained Hash table*: We use a hash table where separate chaining is used to resolve hash collisions. This application has a constant churn in the set of elements being monitored, and therefore, there are a lot of deletions in the hash table. In such a case, a hash table using open addressing [14] will have to resize often to remove the garbage which has accumulated due to the deletions, and designing an efficient and scalable thread safe open hash table is quite complex. On the other hand, minor modifications to the conventional design of a chained hash table [14] can make it cache conscious as well as efficient and scalable, while guaranteeing thread safety. Figure 9 gives an overview of the chained hash table. As depicted in the figure, the design is made cache conscious by grouping the elements in separate chains into a *Block* and the size of the block is made a multiple of the size of the cache line of the underlying architecture.

The hash table is designed in a scalable manner such that minimal locking is needed. The "readers" in the hash table do not need locks, deletions are performed lazily and thus need no synchronization. Locks are needed only to serialize insertions to the same hash bucket. Once a thread has acquired a lock on a bucket, it will *Garbage Collect* all deleted entries in the bucket and add the new entry to the chain belonging to the bucket. Therefore the design is mostly *wait free*, as the locks are held for small intervals, and threads are blocked only when two threads are trying to insert into the same hash bucket. If a moderately robust hash function (such as *Multiplicative Hashing* [14]) is used, then the likelihood of two "writers" mapping to the same hash bucket is very rare. This design also leverages the fact that the frequency counting algorithms limit the number of counters monitored at all times, and thus if a suitable hash table size is chosen, the hash table will not require a resize.

*Delegation and Guaranteeing Satisfaction of the Invariant*: As explained in Section 5.1, the *Search Structure*, should guarantee that Invariant 5.1 is satisfied at all times. This implies that if $Thread_i$ has exclusive rights on stream element $\langle e \rangle$, then any other $Thread_j$, which is also processing $\langle e \rangle$, should not cross the boundary. Instead of making and such $Thread_j$ wait for completion of $Thread_i$, the *Delegation* model is used. $Thread_j$ will delegate its request for processing $\langle e \rangle$ to $Thread_i$ and before $Thread_i$ (or any other $Thread_k$ to which $Thread_i$ delegates its request) relinquishes $\langle e \rangle$, it will complete any other pending request on $\langle e \rangle$. To capture delegation, and since the sole operation on $\langle e \rangle$ is an increment, we associate with each entry in the hash table a *count* which is initialized to 0 and incremented by the threads In the shared design in Section 4.2, $Thread_j$ would be blocked. Algorithm 2 explains how this delegation can be performed efficiently using atomic primitives supported by the underlying architecture. Once the count corresponding to the entry has been incremented, the request for that element has already been "logged", and now it has to be decided if the thread will go ahead with the request (in case it is the sole thread processing the element), or some other thread will eventually process this request (in case of delegation). Since the requests are additive and commutative, so the ordering of the requests does not make any difference, as long as they are logged atomically. Because requests are getting accumulated, the adaptation of *Space Saving* in the *CoTS* framework needs to handle *Bulk Increments*, and the algorithm for handling these is explained in Section 5.2.2.

*Crossing the Boundary*: The action which a thread performs after "crossing the boundary" depends on whether the element was newly inserted or not. If the element was already being monitored, it is an *IncrementCounter* request to the bucket in which the element is residing. If the element was newly inserted and space is still left in the structure, then the request is an *AddElementToBucket* to the minimum frequency bucket, otherwise it is a *OverwriteElement* request to the minimum frequency bucket. Recall that this is in accordance with Algorithm 1 and operations listed in Table 1.

*Relinquishing an element*: Once a thread has processed the

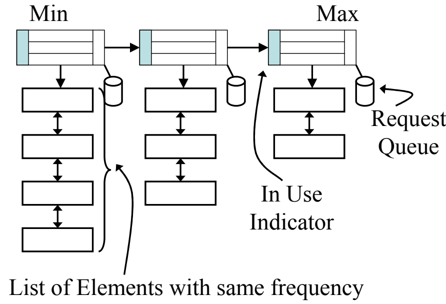



**Figure 10: *Concurrent Stream Summary* structure based on thread "cooperation".**

**Algorithm 3** Processing the *AddElementToBucket* Request

```
Procedure AddElementToBucket(element, bucket)
if (element→frequency = bucket→frequency) then
  bucket→addElementToList(element)
else if (element→frequency < bucket→frequency) then
  if (bucket = minFreq) then
    newBucket ← getNewBucket(element→frequency)
    newBucket→addElementToList(element)
    newBucket→next ← bucket
    minFreq ← newBucket
  else
    DelegateRequestToBucket(minFreq)
  end if
else
  findDestBucket(bucket, element)
end if
end Procedure AddElementToBucket
```

request for an element, it has to relinquish the element. This is again done by a combination of atomic *compare and swap (CAS)* and atomic *swap*. To relinquish, the thread first performs a *CAS* with 1 and 0. If this succeeds, no other request was pending, and the element has been relinquished. A failure implies pending requests and the thread will do a *swap* with 1. This prevents other threads from crossing the boundary, and the swap will return the number of "logged" requests. The thread then subtracts 1 from this value (which corresponds to the request for this thread in the earlier round) and crosses the boundary with the subtracted value as increment for the *IncrementCounter* request.

### 5.2.2 Stream Summary Structure

The structure used for this adaptation is very similar to the original *Stream Summary* structure proposed in [9, 16] (Figure 2) but is slightly modified to suit the requirements of a concurrent design. This modified structure is shown in Figure 10 and is referred to as *Concurrent Stream Summary*. The structure consists of a singly linked list of *Frequency Buckets*. Each bucket maintains a list of elements which have the same frequency as that represented by the bucket. The buckets are ordered by their frequency, and the elements traverse through this structure of buckets as their frequency changes while the stream is being processed. Each bucket has a queue to "log" requests for the bucket.

*Lock Free and Wait Free Design*: As pointed out in Section 4.2, the frequency buckets are points of contention. So in this model, each bucket has an associated queue of requests (Figure 10), which is a thread safe producer-consumer FIFO structure. Whenever a thread wants to perform any operation on a bucket, it will atomically add the request to the queue of the bucket. If the bucket is not currently being held by any other thread, then this thread can acquire this bucket and go ahead with its request. Otherwise, the request has already been delegated and will be processed by the thread which is currently holding this bucket. All these tests can be performed using atomic primitives supported by the hardware and a thread does not need to wait for acquiring any shared resource. It must be noted that in the shared design (Section 4.2), the threads would wait till the resource becomes available again.

*Lock-free Reads*: The structure of frequency buckets is maintained in such a way that any "reads" (reads include reading the frequency of a bucket, or traversing through the structure) in the list of frequency buckets can be made *lock-free*. The main design characteristics that allow these *lock-free* "reads" are: *First*, the frequency of a bucket never changes, if the bucket becomes empty, it is marked as *Garbage Collected* and will be removed eventually. *Second*, whenever a bucket is *Garbage Collected*, enough time is given for "readers" of that bucket to rejoin the main list. *Third*, at no point of execution will there be any broken links in the structure. To ensure this and to adhere to the principle of *Minimal Existence*, the *Concurrent Stream Summary* structure consists of a singly linked list of frequency buckets, compared to a doubly-linked structure in the sequential structure (Figure 2). *Last*, if at any point, the "reader" determines that things have gone wrong, it will abort and restart from where it started the read.

We now describe the algorithms for implementing the basic operations listed in Table 1. These operations will be enqueued as requests in the processing queue of the bucket, and the process that acquires exclusive access to the bucket will process all pending requests for the bucket.

*AddElementToBucket*: An addition request that arrives at a bucket can either be an addition to the current bucket, or to a bucket whose frequency is greater than the present bucket (for dealing with bulk increments). Algorithm 3 provides a high level overview of how the add request is processed. If the addition is to the same bucket, it can be processed right away, while addition to a higher frequency bucket is handled as explained in Algorithm 4. When a new element is added to the set of monitored elements, this request is delegated to the current minimum frequency bucket. If the frequency of the bucket is greater than 1, then the addition request will be for a frequency less than that of the present bucket. The request will then result in creation of a new minimum frequency bucket. It must be noted that after creation of the new minimum frequency bucket, the present bucket can still have pending requests for addition to a lower frequency bucket, and these requests will then be delegated to the new minimum frequency bucket. If the add request was completed by the current thread, this implies that processing of the corresponding element is complete and the thread should relinquish the lock on the element, and this is achieved as described in Section 5.2.1.

*Finding the Destination Bucket*: Finding the next bucket in the case of counter increments is an important part of most requests and is explained in Algorithm 4. If the current next node is not the destination for the element, then

**Algorithm 4** Finding the Destination bucket for an element

**Require:** Invoking thread has exclusive access to startBkt.
**Ensure:** The element has either been added, or addition has been delegated.
**Ensure:** *Garbage Collected* any candidate bucket next to startBkt.

```
Procedure FindDestBucket(startBkt, e)
GarbageCollectCandidateBuckets(startBkt)
nextBkt ← startBkt→next
if (nextBkt = NULL || nextBkt→freq > e→freq) then
   newBkt ← getNewBucket(e→freq)
   nextBkt→addElementToList(e)
   nextBkt→next ← nextBkt
   startBkt→next ← nextBkt
else if (nextBkt→freq = e→freq) then
   DelegateRequestToBucket(nextBkt)
else if (nextBkt→freq < e→freq) then
   /*Process a Bulk Increment.*/
   repeat
      prevBkt ← startBkt
      nextBkt ← prevBkt→next
      repeat
         if (!(nextBkt→isGarbageCollected())) then
            prevBkt ← nextBkt
         end if
         nextBkt ← nextBkt→next
      until (nextBkt ≠ NULL && nextBkt→freq ≤ e→freq)
      returnStatus ← DelegateRequestToBucket(prevBkt)
      /*If returnStatus is false, then the read has to rolled back
      and restarted.*/
   until (returnStatus ≠ TRUE)
else
   startBkt→addElementToList(e)
end if
end Procedure FindDestBucket
```

either a new bucket needs to be inserted after the present bucket, or the list of buckets needs to be traversed. In either case, the design of the structure allows *lock-free* operation. If at any point during traversal through the list, the "reader" finds that it has ended up in a node that has been marked for garbage collection, then it can abort the present run, and start the traversal again. This is likely to converge quickly, as after each garbage collection phase, the destination is approaching the source. From the efficiency perspective, this failure and abort will be rare, as this case would arise when dealing with *bulk increments* and this is common only for the high frequency elements, which are generally towards the extreme right of the *Concurrent Stream Summary* structure. For the less frequent elements in the middle of the structure, the next node will be the destination in most cases. A thread can perform *garbage collection* whenever this routine is invoked.

*Garbage Collection*: This is another important aspect which makes sure that empty *frequency buckets* are removed from the list of frequency buckets in the *Concurrent Stream Summary* structure. This prevents the "readers" from making unnecessarily long traversal through the structure, as well as reduce unnecessary resource usage. An advantage of the design is that *Garbage Collection* can also be done lock free. A bucket can be garbage collected if there are no elements in that bucket, and no pending requests on it. Once this is determined, it is atomically marked as *Garbage Collected*. A thread traversing through the list can remove these marked buckets if these buckets are immediately next to the bucket which the thread owns. Once the buckets have been removed from the list, de-allocation can be done using a reference counting principle as in Java garbage collection.

**Algorithm 5** Processing the *IncrementCounter* Request

```
Procedure IncrementCounter(element, increment)
bucket→removeElementFromList(element)
element→frequency + = increment
findDestBucket(bucket, element)
if (bucket→isEmpty() && bucket = minFreq) then
   /* This is the minimum frequency bucket that had fallen
   empty. This thread should immediately update minFreq.*/
   success ← FALSE
   repeat
      newMinFreq ← findNewMinFreqBucket(minFreq)
      minFreq ← newMinFreq
      success ← newMinFreq→appendQueues(bucket)
      if (success) then
         gcStatus ← false
         while (gcStatus = FALSE) do
            gcStatus ← bucket→atomicMarkGarbageCollected()

            if (gcStatus = FALSE) then
               /*The queue has accumulated some requests and
               needs to be merged.*/
               success ← newMinFreq→appendQueues(bucket)
            end if
         end while
      else
         minFreq ← bucket
      end if
   until (success = FALSE)
   deferAllOverwrites ← FALSE
end if
end Procedure IncrementCounter
```

*Dealing with Accumulated Counts and Bulk Increments*: While an element is being processed inside the *Concurrent Stream Summary* structure, other threads might have accumulated requests for that element inside the *Search Structure*. This would result in *bulk increments*. The *FindDestBucket* function in Algorithm 4 is general enough to handle this situation.

*IncrementCounter*: A request for increment counter will be made only for an element within the current bucket. Increment results in deletion of the element from the bucket and adding it to a bucket corresponding to the desired frequency of the element. Deleting the element from the bucket is straightforward as the thread currently processing this request has exclusive control on that bucket. Finding the destination node can be done by *FindDestBucket* in Algorithm 4. Algorithm 5 shows a high level overview of how the increment request is processed. If after an increment, a bucket falls empty and there are no pending requests, it is atomically marked as garbage collected, and will eventually be removed from the structure by some other thread. If the present bucket is the minimum frequency bucket and it falls empty, it can be immediately removed from the structure, and any pending requests on the minimum frequency bucket is appended to the bucket that becomes the new minimum frequency bucket.

*Overwrite*: This is the most complex request and is specific to the *Space Saving* algorithm which uses this technique to limit the number of counters monitored. This request arrives only on the minimum frequency bucket and processing this request amounts to selecting a candidate element from the minimum frequency bucket, overwriting it with the

**Algorithm 6** Processing the *OverwriteElement* Request

```
Procedure OverwriteElement(element, increment)
bucket ← minFreq
if (deferAllOverwrites) then
   /*There are no candidates to be overwritten, so defer the
   requests.*/
   DelegateRequestToBucket(bucket)
end if
curElement ← bucket→firstElement
while (curElement != NULL) do
   hashtable→tryRemove(curElement→element, status)
   if (status = SUCCESS) then
      /*The current element was successfully removed from the
      Hashtable, so it can be overwritten.*/
      bucket→removeElementFromList(curElement)
      element→error ← bucket→frequency
      element←frequency ← element→error + increment
      findDestBucket(bucket, element)
      break
   end if
   /*The current element is busy, so move to the next ele-
   ment.*/
   curElement ← curElement→next
end while
if (bucket→isEmpty()) then
   /*The minimum frequency bucket has fallen empty. Move
   all pending requests to the next available bucket.*/
   Select new Minimum Frequency Bucket as shown in Incre-
   mentCounter (Algorithm 5)
   deferAllOverwrites ← FALSE
end if
if (curElement == NULL) then
   /*The OVERWRITE request was not processed because
   there are no candidates to be overwritten. So append the
   request to the end of the queue and defer all further over-
   writes till we have some candidate that can be overwritten.
   */
   DelegateRequestToBucket(bucket)
   deferAllOverwrites ← TRUE
end if
end Procedure OverwriteElement
```

current element being processed and then incrementing the count of that element. Algorithm 6 provides a high level overview of how the overwrite request is processed. For selecting a candidate to overwrite, the thread will follow the principle of *Minimal Existence* and will not block on any shared resource. It will start from the first element in the minimum frequency bucket, and to overwrite the element, the corresponding entry in the *Search Structure* should be deleted. This deletion is non-blocking, and a failure implies that some other thread is trying to increment that element, and the increment request should be in the request queue of the minimum frequency bucket. So the thread moves to the next element in the bucket, and the process is repeated. If all elements are busy and none of them could be overwritten, this implies that all these elements have pending increment requests. The overwrite request is thus *deferred* till all the increment requests have been processed. Since a single thread will process all these requests, the processing can be highly optimized as the thread now has much more knowledge about the requests. If at any time, the minimum frequency bucket becomes empty, the thread can immediately remove this bucket, making the next bucket as the new minimum frequency bucket. All pending requests for this bucket are transferred to the next bucket, and now the thread might have elements that can be overwritten.

#### 5.2.3 Dynamic Auto Configuration

The *CoTS* framework has the capability to dynamically determine the number of threads needed for high throughput and low wastage of system resources. This can be done by a thread scheduling algorithm and the *Thread Pool* provided to the framework by the system as shown in Figure 8. The scheduler can start with the number of threads that are available. Whenever a thread crosses the boundary and enqueues a request to the queue of a bucket, it will check to see if the queue size has increased beyond a threshold $\sigma$. In this case, the system puts several threads to sleep and returns them to the *Thread Pool*. On the other hand, when a thread which is processing a frequency bucket delegates its request to another frequency bucket and might also be building up the queue for that bucket. If there is no thread that is processing requests for that bucket, and the queue size is above a threshold $\rho$, then the system wakes up a thread from the pool and this thread can start processing the pending request for this bucket.

Whenever a thread finishes processing the requests for its bucket, and has relinquished the bucket, it checks its neighbors for any pending requests and no thread processing them. In that case, this thread can acquire the bucket and start processing the requests. This checking and traversal through the list will continue till the point a bucket is reached which is acquired by some thread. At this point this thread can leave the *Concurrent Stream Summary* structure and return back to the stream. Occasionally, if there are threads available in the *Thread Pool*, the system can wake up a few threads and assign tasks for them to process. This allows the system to adapt to changes in the input distribution enabling more parallelism.

#### 5.2.4 Answering Queries

The *Concurrent Stream Summary* structure maintains the elements in a sorted order, so that queries can traverse this structure to find the appropriate elements. It must be noted, that for the target applications, queries are far less frequent than the rate of updates, so the design of the *Concurrent Stream Summary* has been optimized for "updates". Queries however can still be processed with considerable efficiency because with most data distributions, the elements of interest will have high frequencies and reside in the rightmost end of the structure, and the low frequency elements will be cluttered in the leftmost end (assuming frequencies increase from left to right). Therefore, as the queries start from the minimum frequency, they can very quickly prune out the low frequency elements and reach the region of interest. Again, queries can be answered without acquiring any locks. We now provide techniques for answering the queries explained in Section 3.2.

*Point Queries*: Frequent elements queries are straightforward and can be answered directly from the *Search Structure* without coming to the *Concurrent Stream Summary*. This is done by looking up the query point in the search structure and reporting it as frequent if its frequency is above the query threshold. For Top-$k$ queries, the frequency of the $k^{th}$ element needs to be determined, and this can be done by a traversal through the structure, reading the buckets for the number of elements and the request queue statistics, thereby counting the number of elements that are present to the right of the present bucket. With a count of number of

elements in the structure, the bucket to which the $k^{th}$ element belongs can be determined. Once this is known, if the frequency of the query point is above the $k^{th}$ frequency, then the point is in the top-$k$, otherwise the answer is negative.

*Reporting the Answer Set*: Answering these queries are costlier since they need traversal through the elements in the bucket. Again, once the appropriate buckets are determined through a traversal of the list of buckets, the delegation model can be used to report the actual elements in the answer set. The query requests can be appropriately prioritized to improve response times.

### 5.3 Generalization of the CoTS Framework

Even though we only discuss the adaptation of the *Space Saving* algorithm, the framework is general enough to be able to accommodate other counter based algorithms. For example, the *Lossy Counting* [15] algorithm divides the stream into multiple rounds and at the end of the round would remove elements which are infrequent. Therefore, for adaptation into the *CoTS* framework, only the *Overwrite* request in *Space Saving* has to be replaced by a request that removes the minimum frequency bucket at round boundaries, everything else remains unchanged.

## 6. EXPERIMENTAL EVALUATION

In this section, we experimentally evaluate the proposed framework. The experiments have been performed on an Intel Core 2 Quad Q6600 processor, which belongs to the "fat camp" as described in Section 3.1. This processor has 4 cores, each corresponding to a hardware thread and operating at a clock speed of 2.4GHz, and the cores share a *L*2 Cache of 4MB. The machine has 4 GB main memory and runs Fedora Core Linux with kernel 2.6.24.5-85.fc8. All algorithms and the framework have been implemented in C++ and compiled using GNU C++ compiler with Level 2 optimization and code generation tuned towards the architecture. The four algorithms (shared, independent, *CoTS*, and sequential) were implemented on the same platform. The data set is synthetically generated and follows zipfian distribution which is very close to realistic data distribution [19]. The zipfian factor $\alpha$ determines whether the distribution is uniform or skewed. The frequency of the elements in the distribution varies as $f_i = \frac{N}{i^\alpha \zeta(\alpha)}$ where $\zeta(\alpha) = \sum_{i=1}^{|A|} \frac{1}{i^\alpha}$ where $N$ is the length of the stream, $|A|$ is the size of the alphabet, and $f_i$ represents the frequency of the $i^{th}$ frequent element. Smaller values of $\alpha$ represent lesser skew in the distribution with $\alpha = 0$ representing uniform distribution. As the value of $\alpha$ increases, the skew of the data distribution also increases. The data set has a total of 100 million elements and an alphabet of 5 million. Different experiments have been performed by varying the size of the data set, the zipfian factor $\alpha$, and the number of threads processing the stream elements. Even though the *CoTS* framework can be adaptive, we do not use this feature for experiments as here we are more interested in the scalability of the system when all threads are working in parallel. GCC built-in atomic primitives were used for performing the atomic operations. In all our experiments, we choose data with $\alpha$ in the range 1.5 to 3.0. The lower $\alpha$ values have not been evaluated because the frequent elements and top-$k$ elements are more interesting and meaningful in a skewed distribution, than in a uniform distribution.

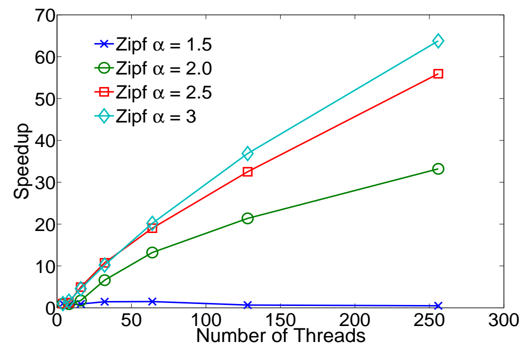
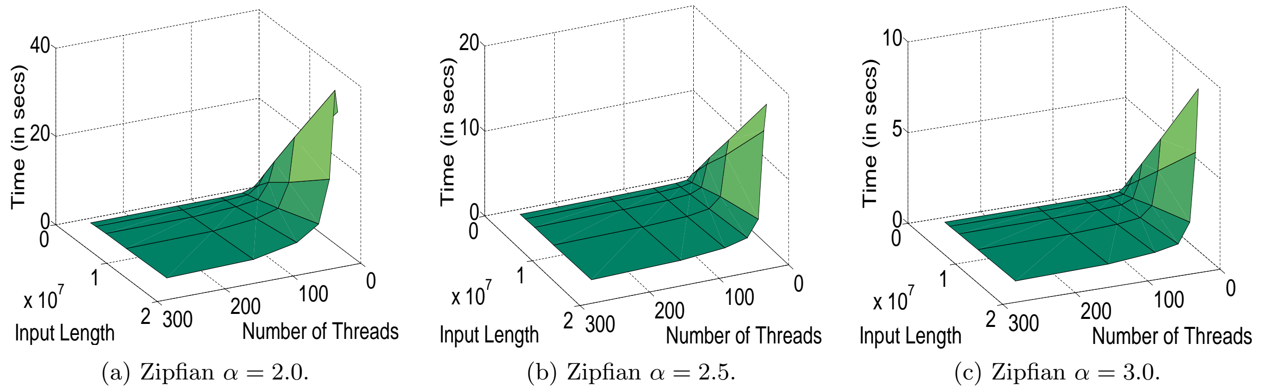



**Figure 11: Scalability of the *CoTS* framework with increasing number of threads.**

Figure 11 shows the scalability of the proposed framework with increasing number of threads, and plots the number of threads along the x-axis, and speed-up along the y-axis. In this experiment, the data size was set to 1 million elements, and the number of threads was varied from 4 to 256. The different plots in the graph correspond to different values of $\alpha$. The *CoTS* framework has been designed with a view towards multiple threads cooperating with each other and delegating requests wherever possible. Because of this assumption, the system performance will be bad when there are not sufficient threads working in parallel. So we select 4 threads as the baseline for computing the speedup. Another reason for this number 4 is that the processor has 4 cores and so below 4 threads the system will not leverage the parallelism offered by the processor. Even though we do not report these values in this experiment, the throughput increases almost by 30 times when the number of threads was increased from 1 to 4.

As is evident from Figure 11, for skewed data, as more and more threads are added to the system, the throughput of the system increases almost linearly. This is because the cooperation model allows two-level delegation (at the bucket level as well as at the element level as explained in Section 5.2), and as a result what used to *points of contention* in the shared design (Section 4.2) has become points of improved performance in this design. The two-levels also explain the super-linear speedup observed in certain cases. For skewed data, since multiple threads are processing the same element concurrently, they can delegate the request in the *search structure* itself, and the thread which is processing the element inside the *stream summary* can perform bulk increments resulting from delegation. These bulk increments (Section 5.2) improve the system throughput considerably and this is evident from the speedup obtained for higher values of $\alpha$. As the value of $\alpha$ decreases and the stream tends towards uniformity, the gains of this delegation model diminishes because there are not many threads processing the same element. In most cases, different threads are processing different elements and thus the delegations do not add up. In addition, as observed in Section 4.3, the *Stream Summary* structure does not allow much parallelism because all requests to the same bucket have to be serialized, and generally there are only two *hot spots* in the structure – the minimum frequency bucket and the buckets near the maximum frequency. As a result, the *stream summary* structure




(a) Zipfian $\alpha = 2.0$. (b) Zipfian $\alpha = 2.5$. (c) Zipfian $\alpha = 3.0$.

**Figure 12: Scalability with varying input size for different values of $\alpha$.**

**Table 2: Comparison of different techniques**

| | $\alpha = 2.0$ | $\alpha = 2.5$ | $\alpha = 3.0$ |
|---|---|---|---|
| Sequential | 0.43861 | 0.520246 | 0.506345 |
| Shared | 13.404 | 12.649 | 12.3309 |
| *CoTS* | 0.662688 | 0.227706 | 0.1115 |

limits the amount of parallelism that can be achieved. So for $\alpha = 1.5$, the system throughput does not scale much beyond 8 to 16 threads. In this case, adding more threads is likely to give diminishing returns, but the low contention in *CoTS* system prevents the system throughput from degrading heavily when more and more threads are added. Therefore, it is evident that the *cooperation* based model performs really well when the data is skewed, and when the skew is less, the system can effectively eke out whatever parallelism is supported by the underlying *stream summary* structure.

In another experiment, the input size was varied from 1 – 16 millions to demonstrate the scalability of the system with increasing input size, and that the execution time increases linearly with a linear increase in length of the input. Figure 12 shows the results of this experiment. In each of the sub-figures, along one horizontal axis we vary the input size, along the other horizontal axis we vary the number of threads, and the vertical axis represents the execution time in seconds. The time reported is the average time for completion of each thread, and is further averaged over multiple repeated runs. The different sub-figures correspond to different values of $\alpha$ and in this experiment we concentrate only on the skewed distributions. As is evident from each of the Figures 12(a), 12(b), and 12(c), the execution time increases linearly with the input stream length, and scalability remains the same irrespective of the size of the input stream. This is particularly important as the input streams can be potentially infinite. This experiment reinforces the scalability results in Figure 11.

In the last experiment, we compare the best case execution time of the two implementations which share the *Stream Summary* structure, with a sequential implementation which does not need any locks or synchronization, and processes the stream sequentially. Table 2 shows this comparison in terms of the absolute execution times (in seconds) obtained from the experiments. The length of the stream is set to 16 million elements. The independent structures technique has not been compared here because the actual cost depends on the merge frequency used. Only the single thread version does not have this overhead, and can be compared directly, but then it is equivalent to sequential. As can be seen from the table, not only does the *CoTS* implementation outperforms the best implementation of *shared* by two orders of magnitude, but it outperforms the sequential implementation for $\alpha$ values 2.5 and 3.0 by 2–4 times. It must be noted that this performance gain is in spite of multiple levels of overhead in the *CoTS* implementation. *First*, the *CoTS* implementation relies on system calls to synchronize between the threads, and each of these calls incur a significant overhead. *Second*, memory allocations in the *CoTS* framework is much higher because of request logging and related book keeping, and these allocation calls again invoke system routines thereby adding to the overhead. *Third*, any calls made to the pthread library for thread management and synchronization involves high overhead. These overheads are even more noticeable as these calls are invoked for every stream element (in certain cases multiple calls per element) and the per element processing cost is not high enough to hide this latency. *Above all*, the underlying structure and the problem semantics do not support high parallelism and as seen for somewhat uniform distributions, the amount of parallelism is limited by the *stream summary* structure.

It should be noted that the time reported in different experiments only involves the time for frequency counting. Since queries are read-only and do not require locks, they will not affect the scalability of the system and if frequency counting scales, so will the queries. Separate threads can be devoted for processing ad-hoc queries and the performance of the threads performing frequency counting will not suffer.

## 7. DISCUSSION AND CONCLUSION

In this paper, we analyze different challenges in *intra-operator* parallelism in the context of the inherent parallelism in modern processor architectures. Based on the analysis and insights from naive parallelization techniques, we propose an efficient and scalable framework for parallelizing frequency counting algorithms. In this paper, we adapted the *Space Saving* algorithm, but other counter based algorithms that monotonically increase the element's frequency can also be adapted to the framework. Our experiments show that the proposed *CoTS* framework is highly scalable

and efficient for skewed distributions. Since the frequent elements and top-$k$ queries are generally interested in skewed data, these results show a positive sign of improvement towards parallel designs which have gained importance with the changing landscape of the processor architectures. For completeness of the framework, it should also scale for less skewed data, but in this implementation, the scaling is limited by the underlying search structure. Therefore, more scalable concurrent structures need to be designed to effectively utilize the parallelism supported by modern processors. In addition, the insights gained from the experiments assert the need for efficient and lightweight system primitives for threading and synchronization, so that the parallel algorithms and frameworks can efficiently leverage the available parallelism. The analysis and techniques proposed in this paper serve as a step forward in this world of multicore processors, but further steps need to be taken to effectively exploit the available parallelism. In the future, we plan to analyze the performance of the *CoTS* framework on the "lean camp" CMP architectures.

## 8. ACKNOWLEDGEMENT

The authors would like to thank Sun Microsystems Inc. for providing the state-of-the-art UltraSPARC T2 processor powered Sun SPARC Enterprise T5120 Server (`http://www.sun.com/servers/coolthreads/t5120/index.xml`). Eventhough the experiments on that architecture have not been reported in this paper, we are currently working towards an efficient implementation of the proposed algorithm in the UltraSPARC architecture.

## 9. REFERENCES

[1] Babcock, B., Babu, S., Datar, M., Motwani, R., and Thomas, D. Operator scheduling in data stream systems. *VLDB J. 13*, 4 (2004), 333–353.

[2] Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Seidman, G., Stonebraker, M., Tatbul, N., and Zdonik, S. B. Monitoring Streams - A New Class of Data Management Applications. In *VLDB* (2002), pp. 215–226.

[3] Charikar, M., Chen, K., and Farach-Colton, M. Finding frequent items in data streams. In *ICALP'02* (2002), pp. 693–703.

[4] Cieslewicz, J., and Ross, K. A. Adaptive Aggregation on Chip Multiprocessors. In *VLDB 2007* (2007), pp. 339–350.

[5] Cormode, G., and Hadjieleftheriou, M. Finding frequent items in data streams. In *VLDB '08* (2008).

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

[7] Das, G., Gunopulos, D., Koudas, N., and Sarkas, N. Ad-hoc top-k query answering for data streams. In *VLDB* (2007), pp. 183–194.

[8] Das, S., Agrawal, D., and Abbadi, A. E. CAM Conscious Integrated Answering of Frequent Elements and Top-k Queries over Data Streams. In *DaMoN '08* (Vancouver, Canada, 2008), pp. 1–10.

[9] Demaine, E. D., López-Ortiz, A., and Munro, J. I. Frequency estimation of internet packet streams with limited space. In *ESA* (2002), vol. 2461, pp. 348–360.

[10] Gedik, B., Yu, P. S., and Bordawekar, R. Executing Stream Joins on the Cell Processor. In *VLDB '07* (2007), pp. 363–374.

[11] Hardavellas, N., Pandis, I., Johnson, R., Mancheril, N. G., Ailamaki, A., and Falsafi, B. Database Servers on Chip Multiprocessors: Limitations and Opportunities. In *CIDR* (2007), pp. 79 – 87.

[12] He, B., Yang, K., Fang, R., Lu, M., Govindaraju, N., Luo, Q., and Sander, P. Relational joins on graphics processors. In *SIGMOD* (2008).

[13] Intel core 2 quad series. `http://download.intel.com/design/processor/datashts/318726.pdf`, 2008.

[14] Knuth, D. E. *The Art of Computer Programming, Sorting and Searching*, vol. 3. Addison-Wesley, Cambridge, MA, 1997.

[15] Manku, G. S., and Motwani, R. Approximate frequency counts over data streams. In *VLDB* (2002), pp. 346–357.

[16] Metwally, A., Agrawal, D., and Abbadi, A. E. An integrated efficient solution for computing frequent and top-k elements in data streams. *ACM Trans. Database Syst. 31*, 3 (2006), 1095–1133.

[17] Mouratidis, K., Bakiras, S., and Papadias, D. Continuous monitoring of top-k queries over sliding windows. In *SIGMOD* (2006), pp. 635–646.

[18] Sun UltraSPARC T2 Processor – The world's first true system on chip. `http://www.sun.com/processors/UltraSPARC-T2/brochure.pdf`, 2007.

[19] Zipf, G. K. *Human Behavior and The Principle of Least Effort.* Addison-Wesley, Cambridge, MA, 1949.