

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 93106-5110, USA
{sudipto, shyam, agrawal, amr}@cs.ucsb.edu

Abstract—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 evaluate different naive techniques for *intra-operator* parallelism, and summarize the insights obtained from the naive techniques. Our experimental analysis of the naive designs shows that *intra-operator* parallelism is not straightforward and requires a complete redesign of the system. We then propose 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 processing throughput of more than 60 million elements per second.

I. INTRODUCTION

Frequent elements [1], [2] and top- k [2] 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. On the other hand, a top- k query returns the k elements with the highest frequency.

Even though numerous algorithms have been proposed in the literature to answer these queries, all the proposed algorithms are sequential 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. In addition to *inter-operator* parallelism, 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.

In this paper, we explore and thoroughly analyze the challenges in *intra-operator* parallelism of frequency counting over data stream. 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* (*Co-operative Thread Scheduling*) and is based on the principle of threads *co-operating* 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 [2] 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 processing throughput of more than 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 propose *CoTS*, an efficient and scalable framework for parallel frequency counting of stream elements. We implement the *Space Saving* [2] algorithm in the *CoTS* framework and our experiments demonstrate scalability and efficiency of the proposed framework.

II. BACKGROUND

A. Related Work

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], 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.

On the other hand, the *counter based* techniques such as [2], [1] 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* [1], the stream is divided into rounds, and at the end of every round potentially in-frequent

Algorithm 1 *Space Saving* algorithm

```
for each element  $\langle e \rangle$  in the stream do
  /*Check if already being monitored*/
  if (LOOKUP( $\langle e \rangle$ )) then
    IncrementCounter( $\langle e \rangle$ )
  else
    if (numCountersMonitored < maxCounters) then
       $e \rightarrow$ frequency  $\leftarrow 1$ 
      AddElementToBucket(minFreq,  $e$ )
    else
      Overwrite(minFreq,  $e$ )
    end if
  end if
end for
```

elements are deleted. This ϵ -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 [2], on the other hand uses a different heuristic to limit space, and details are provided in Section II-B.

With the growing data rates and faster processing speed requirements, researchers are also striving for accelerating data stream queries. For example in [4], 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. In addition, the advent of modern Chip Multiprocessor architectures have opened new frontiers and their ubiquitous presence calls for algorithms that can efficiently exploit the parallelism offered by these architectures. Although much research has been done in the database arena for exploiting the parallelism [5], 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.

B. 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 ϵ . Algorithm 1 gives an overview of *Space Saving*. 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*). In a recent experimental study [6], it was demonstrated that *Space Saving* has highest processing throughput amongst a number of algorithms, and so we select this algorithm for parallelization of frequency counting.

For overwriting, this algorithm needs to have knowledge of the minimum frequency element. The *Stream Summary* structure [2] 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. For lookup, the algorithm needs to have an efficient *search structure* (generally a hash table) that can be integrated with the *Stream Summary* structure. The *Monitored Set* is thus represented by a combination of *Search Structure* and *Stream Summary*.

III. 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 II-B, it is evident that the *Monitored Set* is the point of interest. Due to space limitations, this paper provides an overview of the designs, details can be found in the related technical report [7].

A. 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 *Monitored Set*. These local structures need to be merged into a global structure so that queries can be answered from the global structure.

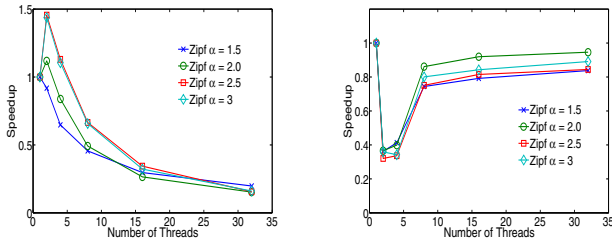
B. Shared Structure

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. 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.

C. 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 with varying zipfian factor α . More details about the experimental set up can be found in Section V and in [7]. Figure 1 shows the results for a data set of 5 million elements.



(a) Independent Structures with a query every 50000 elements. (b) Shared Structure with synchronization using Pthread mutex.

Fig. 1. Evaluation of the naive parallelization techniques for a stream of 5 million elements.

From Figure 1(a), it can be seen that the independent design does not scale as the number of threads increase. Figure 1(a) plots the results for a query every 50000 updates, and the scalability will be worse if the query frequency increases. A break-up of the time shows that even though the frequency counting part scales very well, the counters need to be merged periodically to answer the queries, and as we increase the number of threads, the merge cost increases considerably. The merge cost would increase further if the merges become frequent, i.e. if the query frequency is high.

From Figure 1(b), it can be seen that the shared design also does not scale. A profile of the algorithm reveals that the threads spend a lot of time contending for the locks they need to acquire, and the multiple levels of serialization (element level and bucket level) further degrade the performance and increase the time spent waiting for resources. Figures from the profiling experiments and experiments with varying input size can be found in [7].

IV. COOPERATIVE THREAD SCHEDULING (CoTS)

A. Design Principles

As is evident from the analysis of the naive techniques provided in Section III-C, it is evident that the *shared* design does not scale due to the high overhead of contention between the threads. The main reasons why *contention* based locking is not a good choice are: *First*, if threads need to acquire locks on a shared resources, all threads that are concurrently accessing the resource are serialized. *Second*, an even worse scenario is when a thread needs to acquire multiple locks. This scenario further degrades performance because a thread (T_i) that has acquired a shared lock (on resource \mathbb{R}), and has therefore serialized all other threads (T_j s for $j \neq i$) waiting for the lock (on \mathbb{R}), might it self be waiting for acquiring another lock (on resource \mathbb{R}'). *Third*, contention for locks also adds the overhead of lock arbitration.

We thus propose a framework, *CoTS* (*Cooperative Thread Scheduling*), to optimistically exploit the parallelism inherent in modern processor architectures. Stated formally, if a resource is busy, a thread T_i seeking the resource (\mathbb{R}) would *delegate* the request to thread T_j ($j \neq i$) currently holding the resource, provided that T_j will complete the request *delegated* to it by T_i . Thus, instead of T_i waiting for \mathbb{R} (as in a

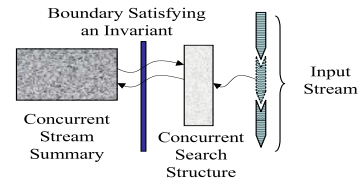


Fig. 2. Overview of the system using the model of cooperating threads.

contention based design), T_i can *make progress* towards its next request. In addition, T_j should refrain from contending for other shared resources and *delegate* wherever required. In this *cooperation based design* paradigm, even though locks are not eliminated completely, “waits” associated with locks have been eliminated.

Let us consider a system comprised of threads (hardware or software), and the task to be completed is a set of *requests* that are *associative* and *commutative*. The system is said to *make progress* if the input set of *requests* is being consumed. The following principles express the *thread cooperation* model:

Principle 1: Request Delegation: If thread T_i is trying to acquire a shared resource \mathbb{R} and it succeeds in acquiring that resource, it will go ahead and complete its request using the resource. If it fails to acquire the resource, it will “delegate” its request to the thread that currently has exclusive access to \mathbb{R} , and move to the next request. All threads trying to acquire \mathbb{R} will “delegate” their requests to T_i . Once T_i finishes its own request, and before it relinquishes control over \mathbb{R} , it will check for any pending requests on \mathbb{R} and will relinquish \mathbb{R} only when all pending requests have been processed.

Principle 2: Minimal Existence: Once a thread has acquired a resource, it will live minimally by abstaining from “blocking” for any other shared resource, thereby allowing it to make unhindered progress. Whenever this thread needs to acquire a shared resource, it will “delegate” the request. Thus the *Delegation Model* is used to obviate the need for acquiring multiple shared resources.

It must be observed that *requests* are *delegated* in two different scenarios. *First*, when a shared resource is not *available*, and *second*, when a thread has already acquired a shared resource, and needs another shared resource. In either case, the threads rely on *cooperation* to optimistically make *progress*. To guarantee correctness, an implementation should satisfy the following invariant:

Invariant 4.1: Fulfillment Guarantee: Once a thread has “delegated” a request, it is neither *lost* nor left *unfulfilled*.

The benefits of the *cooperation* based design include the removal of waits in the design as well as the removal of the overhead of arbitration of locks amongst contending threads.

B. Thread Cooperation for Parallelizing Space Saving

Figure 2 provides a high level overview of the system architecture of the *CoTS* framework. This framework is based on the *shared* design and reduces the contention which is the bottleneck in the *shared* design. The boundary conceptually separates the *Stream Summary* and the *Search Structure*, and these structures interact with each other through a well-defined

interface. In this concurrent processing model of the stream, the system should guarantee that the following invariant holds:

Invariant 4.2: If thread T_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 4.2 holds and this provides the **element level synchronization**. The *Stream Summary* structure can be optimized with the knowledge of Invariant 4.2. 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.

Both the *Search Structure* (hash table), and the *Stream Summary* structures need to be redesigned to incorporate the *thread cooperation* paradigm that utilizes the principles of *request delegation* and *minimal existence*. Due to space limitation, the actual details of implementation of the concurrent structures using the *CoTS* framework is provided in [7].

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. More details about the supported query model, and the techniques for answering queries can be found in [7].

V. EXPERIMENTAL EVALUATION

In this section, we experimentally evaluate the proposed framework. The experiments have been performed on an Intel Core 2 Quad Q6600 processor. 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 *L2* 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 have been implemented in C++ and compiled using GNU C++ compiler with Level 2 optimization and code generation tuned towards the architecture. GCC built-in atomic primitives were used for the atomic operations. 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. The zipfian factor α determines whether the distribution is uniform or skewed. In all our experiments, we choose data with α in the range 1.5 to 3.0. The lower α 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. More details about the experiments can be found in [7].

Figure 3 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 5 million elements, and the number of threads was varied from 4 to 256. The different

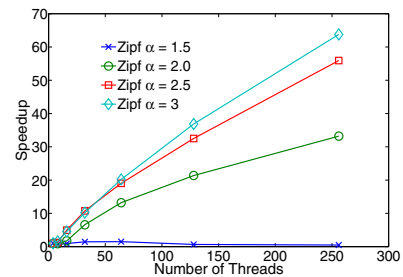


Fig. 3. Scalability of the *CoTS* framework with increasing number of threads.

plots in the graph correspond to different values of α . In this plot, we select 4 threads as the baseline for computing the speedup. It is evident from the figure that the proposed *CoTS* framework demonstrates high scalability. We also performed experiments by varying the size of the input, and the results from this experiment, and detailed analysis of all the results can be found in the related technical report [7].

VI. DISCUSSION AND CONCLUSION

In this paper, we propose the *CoTS* framework to optimistically exploit parallelism in multicore architectures. In this paper, we adapted the *Space Saving* algorithm, but other counter based algorithms that monotonically increase the element’s frequency (like Lossy Counting [1]) 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. Our experiments demonstrate scalability and efficient performance which demonstrates the effectiveness of the proposed design.

ACKNOWLEDGMENT

This work is partly supported by NSF Grants IIS-0744539 and CNS-0423336.

REFERENCES

- [1] G. S. Manku and R. Motwani, “Approximate frequency counts over data streams,” in *VLDB*, 2002, pp. 346–357.
- [2] A. Metwally, D. Agrawal, and A. E. Abbadi, “An integrated efficient solution for computing frequent and top- k elements in data streams,” *ACM Trans. Database Syst.*, vol. 31, no. 3, pp. 1095–1133, 2006.
- [3] M. Charikar, K. Chen, and M. Farach-Colton, “Finding frequent items in data streams,” in *ICALP’02*, 2002, pp. 693–703.
- [4] S. Das, D. Agrawal, and A. E. Abbadi, “CAM Conscious Integrated Answering of Frequent Elements and Top- k Queries over Data Streams,” in *DaMoN ’08*, Vancouver, Canada, 2008, pp. 1–10.
- [5] N. Hardavellas, I. Pandis, R. Johnson, N. G. Mancheril, A. Ailamaki, and B. Falsafi, “Database Servers on Chip Multiprocessors: Limitations and Opportunities,” in *CIDR*, 2007, pp. 79 – 87.
- [6] G. Cormode and M. Hadjieleftheriou, “Finding frequent items in data streams,” in *VLDB*, 2008, pp. 1530–1541.
- [7] S. Das, S. Antony, D. Agrawal, and A. E. Abbadi, “CoTS: A Scalable Framework for Parallelizing Frequency Counting over Data Streams,” UCSB, Tech. Rep. 2008-08, 2008. [Online]. Available: http://www.cs.ucsb.edu/research/tech_reports/