CAM Conscious Integrated Answering of Frequent Elements and Top-k Queries over Data Streams*

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

ABSTRACT

Frequent elements and top-k queries constitute an important class of queries for data stream analysis applications. Certain applications require answers for both frequent elements and top-k queries on the same stream. In addition, the ever increasing data rates call for providing fast answers to the queries, and researchers have been looking towards exploiting specialized hardware for this purpose. Content Addressable Memory(CAM) provides an efficient way of looking up elements and hence are well suited for the class of algorithms that involve lookups. In this paper, we present a fast and efficient CAM conscious integrated solution for answering both frequent elements and top-k queries on the same stream. We call our scheme *CAM conscious Space Saving with Stream Summary (CSSwSS)*, and it can efficiently answer continuous queries. We provide an implementation of the proposed scheme using commodity CAM chips, and the experimental evaluation demonstrates that not only does the proposed scheme outperforms existing CAM conscious techniques by an order of magnitude at query loads of about 10%, but the proposed scheme can also efficiently answer continuous queries.

Categories and Subject Descriptors

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

General Terms

Stream Algorithms, Design, Performance.

Keywords

Data Streams, Frequent elements queries, Top-k queries, Content Addressable Memory, Network Processor.

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

Proceedings of the Fourth International Workshop on Data Management on New Hardware (DaMoN 2008), June 13, 2008, Vancouver Canada

1. INTRODUCTION

Data stream applications, such as click stream analysis for fraud detection and network traffic monitoring, have gained in popularity over the last few years. Common queries posed by the users include frequent elements [16, 4, 7, 19, 17], top-k queries [6, 18, 17], quantile summarization [11], heavy distinct hitters [22] and many more. The frequent elements query looks for elements whose frequency is above a certain threshold. For example, a network administrator interested in finding the IP addresses that are contributing to more than 0.1% of the network traffic will issue a frequent elements query. On the other hand, the 100 most popular search terms in a stream of queries constitute a top-k query. Quantile queries are used for stream summarization such as percentiles and medians, whereas a heavy distinct hitter query is used for detecting malicious activities such as spreading of worms in the network.

The frequent elements and top-k queries are used by different analysis applications such as network and web traffic monitoring, click stream analysis, financial monitoring and so on. Besides applications where the users are either interested in frequent elements *or* top-k elements, there are certain applications where the user is interested in both frequent elements *and* top-k elements on the same stream of tuples. As an example, if we consider the case of a search engine, in order to optimize the performance of the engine, the designer might decide to cache the answers to the 1000 most popular queries. This requires a top-k query on the stream of query terms. On the other hand, for auctioning the search keywords, the designer will be interested in the queries which are above a certain threshold, and accordingly assign bidding price to these keywords. In this case a frequent elements query is used for some support threshold of 0.5%.

Even though frequent elements and top-k queries seem to be very similar, there is a fundamental difference. In frequent elements computation, there is a notion of the minimum possible frequency of an element but no ordering information is necessary, whereas in answering top-k queries, the exact frequencies of the elements might not be of interest as long as an ordering of the elements is known. As a result, answering frequent elements cannot be used as a pre-processing step for top-k queries, and

vice versa. A specialized solution is therefore sought for the applications that need frequent elements and top-k queries on the same stream of elements. Metwally et. al. [17] suggest an efficient integrated solution, known as *Space Saving*, for answering both frequent elements and top-k queries. The authors propose a counter based technique for frequency counting, and a data structure to order the elements by their frequencies so that top-k queries can be easily answered.

Besides the need for having an integrated solution for frequent elements and top-k queries, the ever increasing data stream rates call for fast and efficient processing. In addition, new hardware paradigms have opened up new frontiers for efficient data management solutions leveraging specialized hardware features [5, 10, 8, 12, 3, 25, 23, 2]. A Content Addressable Memory (CAM) can also be exploited for accelerating stream processing. An interesting property of CAM is that in addition to normal read and write operations, it supports constant time lookup operation in hardware. A CAM obviates the need for complex data structures, such as Hashtables or search trees, to process efficient lookups. Hence, a CAM can be efficiently used by algorithms that perform frequent lookups or searches. Bandi et al. [1] propose a CAM conscious adaptation for the *Space Saving* algorithm and demonstrate acceleration compared to a software implementation. A disadvantage of the proposed adaptation is that the elements are not sorted by their frequencies. As a result, the algorithm cannot answer top-k queries efficiently. Additionally, since the elements are not sorted, adapting the approach in [1] for continuous queries, or even moderately high query loads, is not straightforward. In this paper, we propose a CAM conscious version of the *Space Saving* algorithm that also maintains the ordering of the elements and hence can answer both frequent elements and top-k queries on the same stream. Our scheme, *CAM conscious Space Saving with Stream Summary (CSSwSS)*, can efficiently answer continuous queries. The major contributions of this paper are summarized as follows:

- This is the first approach of using CAM for providing an "integrated" solution to frequent elements and top-k queries.
- We propose a CAM based data structure to count occurrences of the elements and efficiently maintain the sorted order of the elements in terms of their frequency. We explore the possible design alternatives and analyze their advantages and disadvantages.
- We provide an implementation of the proposed algorithm using a commodity CAM chip, and report the performance of this algorithm on an experimental prototype using synthetic data sets.

The rest of the paper is organized as follows: Section 2 summarizes the work that has been carried in frequent elements and top-k computation, and in using specialized hardware for data management operations, Section 3 explains the hardware prototype used for our implementation, Section 4 explains in detail our proposed algorithm, Section 5 provides a thorough experimental evaluation and analysis of the results and Section 6 concludes the paper.

2. RELATED WORK

Frequent elements and top-k queries are among the most common queries in data stream processing applications, and a large number of approaches have been suggested to answer these queries. The algorithms for answering frequent elements queries are broadly divided into two categories: *sketch based* and *counter based*. The *sketch based* techniques such as the one proposed by Charikar et al. [4] try to represent the entire stream's information as a "sketch" which is 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. These approaches are therefore not suitable for providing fast answers to queries.

On the other hand, the *counter based* techniques such as [17] 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. For example, Manku et al. [16] propose a technique called *Lossy Counting* in which the stream is divided into rounds, and at the end of every round, potentially non-frequent 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. Panigrahy et al. [19] suggest a sampling based counting technique which monitors a subset of elements and manipulates the counters based on whether a sampled element is already being monitored or not. For a bursty stream this approach has a space bound of $O(\frac{F_2}{t})$, where F_2 is the 2^{nd} frequency moment and t is the minimum frequency to be reported. Most of these *counter based* algorithms [16, 17, 19] are a generalization of the classic *Majority* algorithm [9], and the goal is to minimize the space as well as reduce the error in approximation.

Different solutions have also been suggested for answering top-k queries. Mouratidis et al. [18] suggest the use of geometrical properties to determine the k-skyband and use this abstraction to answer top-k queries, whereas Das et al. [6] 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. Bandi et al. [1] suggested the use of Content Addressable Memories (CAM) for accelerating the frequent elements queries. The authors leverage the constant time lookups of CAM to accelerate a couple of counter based techniques. As pointed out earlier, this approach cannot efficiently answer continuous top-k and frequent elements queries. Bandi et al. [2] also proposed the use of CAM for Database operations. Other approaches for data management on new hard-

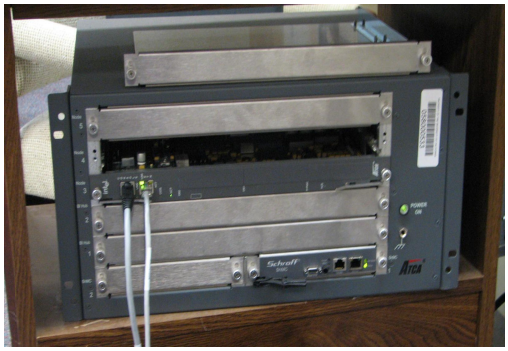


Figure 1: Hardware prototype.

ware paradigms have also been proposed. For example, Gold et al. [10] leveraged Network Processors for accelerating database operators, while Cieslewicz et al. [5] propose the use of Chip Multiprocessors for answering aggregation queries. The use of Graphics processors has also been proposed by Fang et al. [8]. For this work, we concentrate on using CAM for efficient answering of frequent elements and top-k queries.

3. HARDWARE PROTOTYPE

The hardware prototype (Figure 1) used for our implementation consists of two major constituents: the Network Processing Unit (NPU) and the Ternary Content Addressable Memory (TCAM). In this section, we explain the features of the two constituent parts.

3.1 NPU Architecture

For the implementation in this paper, we use the Intel IXP2800 network processor [15]. This network processor consists of a Control Plane Processor (CPP) and 16 independently operating data plane processors referred to as Micro Engines (ME). The CPP is a 32-bit XScale core that runs Monta Vista linux, operates at a maximum clock speed of 700MHz, and is designed to work as a "master" assigning tasks to the ME. On the other hand, the ME is designed to perform simple data plane operations very fast. The MEs have a very simple design and instruction set and have been optimized to quickly perform simple operations. Each ME operates at a maximum clock speed of 1.4GHz and has 8 hardware thread contexts. Therefore, the NPU provides $128(16 \times 8)$ hardware threads. A hardware thread is different from a software thread since each thread has its own set of registers and hence switching contexts between threads incurs minimal overhead. Each ME has a set of general purpose registers and a small instruction cache.

In addition to the Thread Level Parallelism (TLP), the NPU also provides a form of pipeline parallelism. The MEs are arranged in a pipelined fashion and an ME shares a set of registers with its immediate next ME. These registers are called nearest neighbor registers and are designed for fast inter-communication between the MEs. For our implementation we use only a single ME and do not use either of the above mentioned forms of parallelism.

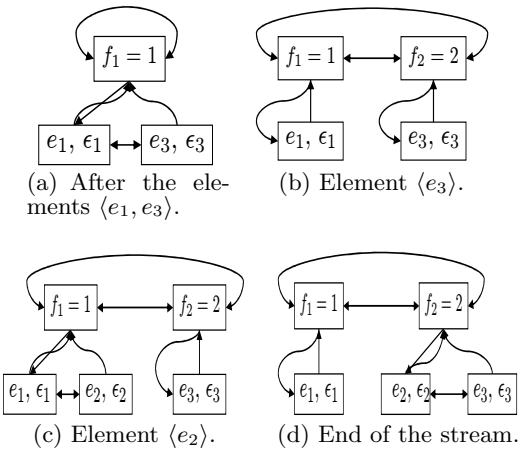
The CPP as well as the 16 MEs share some common on-chip resources like the PCI unit, hashing unit, the on-chip shared memory known as scratchpad, as well as the industry standard DRAM and SRAM interfaces. Main memory is available in the form of DRAM and SRAM and is present on-board. A TCAM chip can be efficiently interfaced with the NPU through the SRAM interface.

Some important features of this architecture are as follows: *First*, even though the NPU provides a lot of parallelism, the architecture is so simple that only a small set of instructions are supported and is suited for simple operations. For example, floating point operations are not supported by the ME. So the applications for which the NPU can be used are very limited. *Second*, the CPP, which acts as a master, runs at a speed slower than the ME. Therefore, when running parallel threads, the master should also perform simple tasks or else the master might become a bottleneck. *Finally*, the MEs do not have any support for a memory hierarchy. This is both an advantage as well as a disadvantage. The programmers have the freedom to decide precisely where their data resides, but this design increases the overhead for the application design.

3.2 TCAM Architecture

Content addressable memories (CAM) provide efficient lookup operations in hardware and have been typically used for networking applications such as IP address lookups for packet forwarding or implementation of access control lists. For our implementation, we use the IDT 75K62134 chip [13] which is a Ternary Content Addressable Memory (TCAM). In addition to the properties of a CAM, a TCAM has ternary capabilities (i.e. it can represent a "don't-care" state and hence can efficiently represent ranges) and are therefore suited for longest prefix matching in IP forwarding. For example, a TCAM can store an IP range as say 168.111.$*$, such that any IP address in the range 168.111.0.0 to 168.111.255.255 will match this entry. A typical TCAM chip, similar to the one used in this paper, consists of either 36-bit or 72-bit word-arrays and typical sizes are $128K$ or $256K$ of these entries. The proposed algorithms do not use the ternary capabilities, so a TCAM and a CAM become functionally equivalent.

A TCAM consists of a two dimensional array of bits, and the lookup is performed by a parallel comparison of the search key with all the stored words in a SIMD-like fashion. If the search key is present in the TCAM core, then a match is reported and this is referred to as a *hit*. If multiple copies of the search key are found, then the smallest index is reported by a priority encoder, and a special *multi-hit* bit is set. Since all the elements in the TCAM array are compared in parallel, the power consumption of the TCAM is pretty high. To lower the power consumption, the chip we use allows division of the TCAM into a number of search databases, so that the application designer can selectively power down the databases that are not in use and ignore them for comparison. The TCAM chip also supports *mask registers*, which can be used to selectively mask certain bits dur-



(a) After the elements $\langle e_1, e_3 \rangle$. (b) Element $\langle e_3 \rangle$. (c) Element $\langle e_2 \rangle$. (d) End of the stream.

Figure 2: This figure illustrates the *Stream Summary* data structure for an example stream of elements $\langle\ e_1, e_3, e_3, e_2, e_2\ \rangle$.

ing the `LOOKUP` and `WRITE` operations. The bits that are masked do not participate in comparison during the `LOOKUP` operations, and are not affected by the `WRITE` operation. In addition to these operations, a TCAM also supports reads and writes similar to conventional memories.

Some important characteristics of the TCAM are as follows: *First*, the chip can support a lookup throughput of 100 million lookups per second, but that is possible only through parallel access to the TCAM and pipelining of requests. The chip under consideration can support up to 128 pending requests. For our implementation, we do not consider the parallelism supported by the TCAM. *Second*, even though the TCAM efficiently interfaces with the NPU, the word sizes of two devices do not match. The TCAM core is 72-bit wide, whereas the SRAM bus of the NPU is only 32-bit wide. As a result, all communication between the NPU and the TCAM takes place through multi-word transfers. Due to this overhead, it is very difficult to attain the maximum supported throughput.

4. SYSTEM DESIGN

This section explains the proposed algorithm for the hardware prototype described in Section 3.

4.1 Background

Metwally et. al. [17] proposed an "integrated" scheme for answering frequent elements and top-k queries. The proposed counter based technique, called *Space Saving*, is based on the intuition that frequent elements are of importance only in a skewed stream, and in a skewed stream, the frequency of the elements of interest is much higher compared to the low frequency ones. This heuristic is used to limit the number of elements which need to be monitored. A nice property of this algorithm is that the number of elements to be monitored is independent of the length of the stream or the size of the alphabet, and depends on only the user specified error bound. In this algorithm, with the arrival of an element in the stream, if the element is already being monitored, then its count is incremented, otherwise the new element replaces the element with the minimum frequency. The space bound for this algorithm is $O(\frac{1}{\epsilon})$, where ϵ is the user specified error bound. An ϵ-*approximate* algorithm is one such that given a support ϕ, instead of reporting the elements above the frequency ϕN, the algorithm reports the elements whose frequency is above $(\phi - \epsilon)N$.

To answer top-k queries and keep track of the minimum frequency element, this algorithm maintains a data structure, called *Stream Summary* [7, 17], which can efficiently keep the elements sorted by their frequency. The *Stream Summary* data structure consists of "frequency buckets" which correspond to frequency values of at least one element that is being monitored, and each bucket consists of the elements which have the same approximate frequency as is represented by the bucket. Figure 2(a) provides an illustration of the *Stream Summary* structure with two elements with frequency 1. Figure 2 illustrates how the elements in the stream are processed and maintained in sorted order of their frequency. For example, in the example stream, Figure 2(a) shows the structure after elements $\langle e_1, e_3 \rangle$ have been processed. When the element e_3 appears again, its frequency is incremented to 2, and Figure 2(b) shows the state of *Stream Summary* structure. The elements are always kept sorted, and the per-element processing cost is a constant [7, 17].

Authors in [1] propose an adaptation of the *Space Saving* algorithm to use the constant time lookup of TCAM. They store the elements and their frequencies in the TCAM. As the stream is being processed, the elements can be looked up in constant time. In addition to that, the *Space Saving* algorithm needs to keep track of the minimum frequency element. This can also be done efficiently by looking up the minimum frequency from the TCAM and always keeping track of the minimum frequency. As mentioned earlier, this TCAM adaptation does not sort the frequencies and hence answering queries is costly. In this paper, we refer to this algorithm as *CAM conscious Space Saving (CSS)*.

4.2 Proposed Algorithm

Stream Summary is a powerful structure that can be used to efficiently keep the elements sorted by their frequency. This structure can be easily adapted to the CAM setting. With every stream element, the algorithm needs to determine whether the element is already being monitored. Since CAM provides efficient lookups, we can place the elements in the CAM. On the other hand, in order to keep the elements sorted by their frequency, the circular doubly-linked structure of frequency buckets is maintained in SRAM. Figure 3 provides an illustration of the *CAM adapted Stream Summary* data structure. In Algorithm 1, we provide the details of how the *Space Saving* algorithm can be adapted to use the *CAM adapted Stream Summary*. We call our algorithm *CAM conscious Space Saving with Stream Summary (CSSwSS)*. Algorithm 2 gives an overview of the supporting routines used by Algorithm 1.

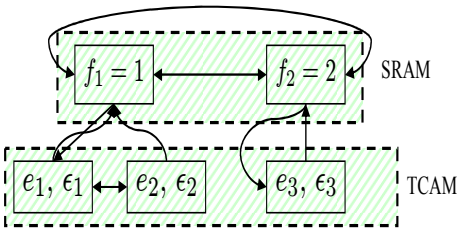


Figure 3: TCAM adaptation of the stream summary data structure.

As described in Section 3.2, the TCAM chip we use for our implementation provides support for programmable size of search keys. This is accomplished by having TCAM core of different sizes. An advantage of having a larger TCAM core is that more data can fit into a single TCAM entry; the disadvantage being that each TCAM operation becomes costlier when compared to smaller TCAM cores. Due to this programmable size, two possible implementations of the proposed approach are possible, based on what information is kept in the TCAM and what is kept outside. It must be noted that in addition to storing the elements, its frequency, the error in frequency approximation, and the link structure for the sort order also need to be stored.

Narrow TCAM Core (72-bit): The TCAM chip used in this implementation consists of an array of 72-bit wide entries. As a result, the 72-bit core size is an obvious choice. In this design, as the number of bits in the TCAM core is less than the information to be stored, part of information is stored in the TCAM and the rest of the information is stored in SRAM with a pointer to the corresponding information stored in TCAM. Since the elements need to be looked up, they must be placed in the TCAM. The placement of other items is an implementation choice. In this implementation, we place the element and the link structure of the elements in the TCAM, while the error and a pointer to the frequency bucket is placed in the SRAM. An advantage of this approach is that TCAM space is preserved as we are using the bare minimum when it comes to TCAM resources. The disadvantage is an added level of indirection which is incurred when processing a stream element. Since all information is not stored in the TCAM, a TCAM `LOOKUP` is followed by a SRAM read to obtain all information corresponding to an element.

Wide TCAM Core (144-bit): The chip being used allows programmable word size. So the device may be programmed to have a core size of 144-bits, where two consecutive 72-bit entries are combined. As the TCAM entries are wider in this design, all necessary information corresponding to an element, i.e. the error and pointer to the frequency bucket, fit into a single wide TCAM entry. The advantage of this design is that the additional level of indirection is avoided, the disadvantages being more TCAM space being utilized (which might be undesirable as the number of TCAM entries is limited), and wide core makes each TCAM operation costlier compared to the operations with a narrow core.

Algorithm 1 *CAM conscious Space Saving with Stream Summary (CSSwSS)*

```
/* mask_1 register masks the element component */
/* LOOKUP, WRITE are TCAM operations. */
/* min_fr is the pointer to the minimum frequency bucket in
Stream Summary. */
Procedure ComputeFrequentTopK(stream, table_size,
min_fr)
for each element ⟨e⟩ in stream do
  hit ← LOOKUP(e, index, mask_1)
  if (hit) then
    /* The element is already being monitored, so increment
    its counter. */
    cur_fr ← FrequencyBucketAtIndex(index)
    added_bucket ← IncrementCounter(cur_fr, index)
    added_elem ← index
  else
    /* The Element is not being monitored, so either add
    to the end of the list if there is space, or overwrite the
    minimum. */
    if (cur_size < table_size) then
      /* Space is left, so add element */
      WRITE(free_index, e, 0, ⟨link_structure⟩)
      if (min_fr = NULL || min_fr→freq > 1) then
        /* A new Frequency Bucket must be added to the
        list. */
        new_node ← AllocateNewNode()
        new_node→freq ← 1
        AddElementToList(new_node, free_index)
        added_bucket ← new_node
        added_elem ← free_index
      else
        /* A bucket with frequency 1 already exists, add
        this element to that bucket. */
        AddElementToList(min_fr, free_index)
        added_bucket ← min_fr
        added_elem ← free_index
      end if
      free_index++
    else
      /* Overwrite the minimum element. */
      index ← min_fr→elem
      WRITE(index, e, min_fr→freq, ⟨link_structure⟩)
      added_bucket ← IncrementCounter(min_fr, index)
      added_elem ← index
    end if
  end if
end for
end Procedure ComputeFrequentTopK
```

4.3 Query Processing

The *CAM adapted Stream Summary* data structure can be used to efficiently answer the frequent elements and top-k queries. In this paper we consider two different types of queries, viz. *Continuous Queries* and *Interval Queries*. *Continuous Queries* are posed with every update, i.e. with every stream element processed, and the answer cache is always up-to-date. On the other hand, *Interval Queries*, as the name suggests, are posed at regular intervals, and hence the answer cache is updated at regular intervals. In this section, we provide a detailed analysis of the algorithm for using this structure to efficiently answer the queries. First we will consider the frequent elements query, and then move on to top-k query.

Algorithm 3 provides a scheme for answering continuous frequent elements queries. Since the elements are sorted, answering this query amounts to keeping a

Algorithm 2 Supporting method for *CSSwSS*

```
/* READ is a TCAM operation. */

Procedure IncrementCounter(freq_node, element_index)
/* Increments the count of the specific element. */
new_fr ← freq_node→freq + 1
next_node ← freq_node→next
if (freq_node→count = 1 & next_node→freq ≠ new_fr) then
   /* This bucket can be promoted to become the new bucket.
   */
   freq_node→freq ← new_fr
   return freq_node
end if
RemoveElementFromList(freq_node, index)
if (next_node→freq = new_fr) then
   /* The element moves to the next node. */
   AddElementToList(next_node, index)
   return next_node
else
   /* A New Frequency Bucket need to be inserted next to the
   current node. */
   new_node ← AllocateNewNode()
   AddToNext(new_node, freq_node)
   AddElementToList(new_node, index)
   return new_node
end if
end Procedure IncrementCounter
```

Algorithm 3 *Answering continuous frequent elements queries*

```
/* added_bucket and added_elem are the bucket added and
elements added.*/
Procedure ContinuousQueryFrequent(φ)
/*ptr_φ is the pointer to the bucket with minimum frequency
above support φ*/
/* φ is the minimum frequency to be reported. */
if (ptr_φ ≠ NULL) then
   next_node ← ptr_φ→next
   min_freq ← ptr_φ→freq
   if (min_freq < φ) then
      /* At least one element has become infrequent. */
      if (next_node→freq > min_freq) then
         ptr_φ ← next_node
      else
         ptr_φ ← NULL
      end if
   end if
   if (added_bucket→freq ≥ φ & added_bucket→freq <
   min_freq) then
      /* An infrequent element has become frequent. */
      ptr_φ ← added_bucket
   end if
else
   /* No frequent element yet. */
   if (added_bucket→freq ≥ φ) then
      ptr_φ ← added_bucket
   end if
end if
end Procedure ContinuousQueryFrequent
```

pointer to the bucket that has the minimum frequency above the support ϕ. We can use the *CAM adapted Stream Summary* structure to efficiently maintain this pointer (ptr_ϕ). The intuition behind this algorithm is that after processing an element, a bunch of elements might become infrequent, while only one element can become frequent. These elements can be determined from Theorems 4.1 and 4.3 and Corollary 4.2.

THEOREM 4.1. *When processing continuous frequent elements queries, if element $e_i \in bucket_i$ becomes infrequent then all elements $e_j \in bucket_i$ and only elements e_j become infrequent.*

PROOF. The proof is of the theorem consists of two parts.

- *All elements $e_j \in bucket_i$ becomes infrequent if $e_i \in bucket_i$ becomes infrequent.* This is intuitive from the structure of the *CAM adapted Stream Summary* structure as all elements in the same frequency bucket have the same frequency.

- *Only elements e_j become infrequent.* Since we are answering continuous queries, the length of the stream increases by 1 at each step. Now, since $\phi = \lambda N$, where $0 < \lambda < 1$ and N is the length of the stream, and if f_i is the frequency of e_i, as e_i was reported as frequent in the previous step, and considering the fact that N increased by 1 after the previous step, all $f_k > f_i$ must be reported as frequent. The *CAM adapted Stream Summary* structure ensures that all buckets to the right of $bucket_i$ (refer to Figure 2 for illustration) will have $f_k > f_i$. Therefore, only the elements e_j become infrequent if e_i become infrequent.

□

COROLLARY 4.2. *Only the elements with the minimum frequency amongst the set of reported frequent elements can become infrequent.*

PROOF. The proof follows from Theorem 4.1, since for each new element, only the elements from a single frequency bucket can become infrequent, and since the buckets in *CAM adapted Stream Summary* are in sorted order. □

THEOREM 4.3. *When processing continuous frequent elements queries, only the element seen last can become frequent.*

PROOF. An element e_i with frequency f_i will be reported as frequent $\iff$ $f_i > \phi$ where $\phi = \lambda N$ and $0 < \lambda < 1$. Since the number of elements N is monotonically increasing, therefore ϕ is also monotonically increasing. If e_l is the last element processed, then f_l is the only frequency that increases over the previous step. Hence e_l can be the only element that might become frequent. □

From the theorems, it is evident that updating ptr_ϕ induces a constant cost per element being processed: reporting an element becoming frequent is constant and reporting p elements as infrequent is $O(p)$. This is independent of the number of elements in the stream or the number of elements monitored in the *CAM adapted Stream Summary* structure. This is a drastic improvement from the *CAM conscious Space Saving (CSS)* in [1], where the cost of query answering is $O(n)$, n being the number of elements currently being monitored, and evidently, $p \ll n$ in most iterations.

Since answering continuous queries is not efficient for *CSS*, in our experiments we compare *CSS* with *CSSwSS* using varying query load, i.e., instead of the queries being continuous, now the queries are issued at regular

Algorithm 4 *Answering frequent elements queries at regular intervals*

```
/* added_bucket and added_elem are the bucket added and
elements added.*/
Procedure IntervalQueryFrequent(φ)
/*ptr_φ is the pointer to the bucket with minimum frequency
above support φ*/
/* φ is the minimum frequency to be reported. */
if (ptr_φ ≠ NULL) then
   cur_fr ← ptr_φ
else
   cur_fr ← added_bucket
end if
next_node ← cur_fr→next
prev_node ← cur_fr→prev
freq ← cur_fr→freq
if (freq ≥ φ) then
   /*Check if any infrequent element has become frequent.*/
   while (prev_node→freq < cur_fr→freq & prev_node→freq
   ≥ φ) do
      cur_fr ← prev_node
      prev_node ← cur_fr→prev
   end while
   ptr_φ ← cur_fr
else
   /* Some frequent elements have become infrequent, so up-
   date the cache. */
   while (next_node→freq > cur_fr→freq) do
      cur_fr ← next_node
      next_node ← cur_fr→next
   end while
   if (next_node→freq < cur_fr→freq) then
      ptr_φ ← NULL
   else
      ptr_φ ← cur_fr
   end if
end if
end Procedure IntervalQueryFrequent
```

Algorithm 5 *Answering continuous top-k queries*

```
/* added_bucket and added_elem are the bucket added and
elements added.*/
Procedure ContinuousQueryTopK(k)
/* Set_k is the set of elements in top-k cache with minimum
frequency. */
/* ptr_k is the pointer to the bucket containing elements in
Set_k*/
if (num_elems_topk < k) then
   /* There are not enough elements in the top-k set. */
   if (added_elem ∉ top-k set) then
      Report added_elem ∈ top-k
      num_elems_topk++
   end if
   if (num_elems_topk = k) then
      /* k elements have now been reported, enter maintenance
      mode. */
      ptr_k ← min_freq
      Set_k ← ElementsInBucket(ptr_k)
   end if
else
   /* Check for updates, if any. */
   ptr_k^+ ← ptr_k→next
   if (added_bucket = ptr_k^+ & ptr_k^+→freq − ptr_k→freq = 1)
   then
      if (added_elem ∈ Set_k) then
         Set_k ← Set_k − added_elem
      else
         Select elem ∈ Set_k
         Set_k ← Set_k − elem
         Report elem ∉ top-k
         Report added_elem ∈ top-k
      end if
      if (Set_k is empty) then
         ptr_k ← ptr_k^+
         Set_k ← ElementsInBucket(ptr_k)
      end if
   end if
end if
end Procedure ContinuousQueryTopK
```

interval. Algorithm 4 gives an overview of the scheme used for answering queries at regular intervals. Since the queries are not issued after processing every single element, Theorems 4.1 and 4.3 do not hold. Hence Algorithm 3 cannot be used for this purpose. It can be seen that Algorithm 4 uses an idea similar to that of Algorithm 3, except that Algorithm 4 scans through the structure, because the number of elements that have been processed since the last invocation is not known. But it can be easily proved that the time taken by Algorithm 4 is $O(m)$, where m is the number of elements processed since the last invocation. So, in the worst case it might reduce to a continuous query, but as demonstrated in the experiments in a later section, the performance is much better for most practical cases.

The *CAM adapted Stream Summary* structure can be used to efficiently answer continuous top-k as well. The idea is the same as with continuous frequent elements queries and the top-k set is selected using the layout of the *CAM adapted Stream Summary* structure. The algorithm is very similar to the corresponding algorithm in [17] and has been adapted to efficiently leverage the *CAM adapted Stream Summary*. An overview of the algorithm for continuous top-k monitoring is provided in Algorithm 5.

Again, it is straightforward to see that the per-element cost of continuous maintenance of set of top-k elements is pretty small and the *CAM adapted Stream Summary* structure can be used to efficiently maintain the top-k set. It is therefore evident that *CSSwSS* can be used to efficiently answer frequent elements and top-k queries.

5. EXPERIMENTAL EVALUATION

We implement the *CAM conscious Space Saving with Stream Summary (CSSwSS)* algorithm on a commodity TCAM chip IDT75K62134 [13] which interfaces efficiently with the IXP2800 NPU [15]. The TCAM chip has $128K$ 72-bit wide entries, supports programmable word size, and up to 128 parallel request contexts. Development is done using the Teja NP ADE [21] and Intel Development Workbench [14]. Implementation of the algorithms involve coding in $TejaC^{TM}$ and $MicroC^{TM}$. Times reported are actual execution times of the algorithms on the MEs (Micro Engine), and are obtained from the *Timestamp* register common to all the ME's. The experiments have been repeated multiple times and the values have been averaged over multiple runs.

The experiments have been performed with synthetic Zipfian data which has been shown to closely resemble realistic data sets [24]. The data set used for experiments consists of 1 million hits, taken from an alphabet of size $10,000$. The alphabet is the number of distinct elements in the stream. Since the performance of the *Space Saving* algorithm is not dependent on the size of the alphabet, we expect similar results for smaller or larger alphabet sizes. The zipfian factor is varied from 0 to 3 in steps of 0.5, where zipfian factor 0 represents uniform distribution while 3 is a highly skewed distribu-

TCAM Operation	72-bit	144-bit
READ	0.32	0.64
LOOKUP	0.2971	0.3673
WRITE	0.32	0.3314

Table 1: Time (in secs) for a million TCAM operations for different sizes of TCAM core.

tion. The error bound ϵ is set to 0.001. We compare the performance of the proposed algorithms with the CAM conscious *Space Saving* adaptation in [1], which we refer to as *CAM conscious Space Saving (CSS)*.

5.1 Cost of Frequency Counting

In this section, we experimentally evaluate the two possible design alternatives. First we need to determine the cost of the primitive operations in narrow and wide TCAM cores. The operations of interest are `LOOKUP`, `READ` and `WRITE`. We evaluate this using an experiment where we time only the operations of interest, and Table 1 summarizes the results.

There are a few interesting observations about the statistics in Table 1. `LOOKUP` is one of the most important operations, and this operation is done at least once for each stream element. As we increase the width of the TCAM core, the cost of `LOOKUP` increases, but the increase in the cost of `WRITE` does not increase significantly. The highest increase is for the `READ` operations where the time almost doubles, as the chip used only supports up to 72-bit wide `READ` operations. As a result, a 144-bit `READ` comprises of two 72-bit `READ` operations. But this is not very alarming as the layout of elements can be designed in a manner that would never require a 144-bit `READ`, i.e. the algorithm will need either the first or the second 72-bit entry but not both, so that only the cost of 72-bit wide `READ` operations is incurred.

Another interesting observation about the times in Table 1 is that the `LOOKUP` throughput is nowhere near the peak throughput of about 100 million `LOOKUP`s per second as stated in Section 3.2. This is primarily because we use only a single ME for implementing our algorithm and accessing the TCAM, and the high throughput can be obtained by parallel accesses to the TCAM. We will exploit this parallelism in future work. In addition, it must be noted that we are using a single ME that runs at a speed of only 1.4GHz.

Before evaluating the performance of the algorithms for answering queries, we evaluate the cost of counting the frequencies and keeping them sorted. Since the authors in [1] provide an efficient frequency counting algorithm (*CSS*), we compare the performance of the proposed schemes with *CSS*. Figure 4 provides a comparison of time taken for frequency counting. From the figure it can be seen that for uniform data, keeping the elements sorted is almost twice as costly as simple frequency counting. But as the data becomes skewed, the elements can be kept sorted almost for free. This difference in performance is due to the fact that the proposed scheme involves managing the structure of frequency buckets, which becomes costly when the sort order of elements is changing rapidly, which is the case for the uniform distribution (zipfian factor close to 0). On the other hand, when the data is skewed, the sort order does not change much, and the proposed scheme performs better as the cost of maintaining the minimum frequency element in *CSS* now becomes significant, which is not present in *CSSwSS*.

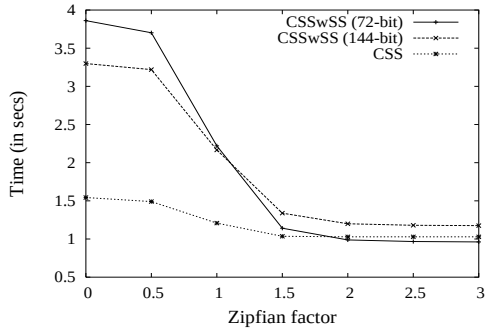


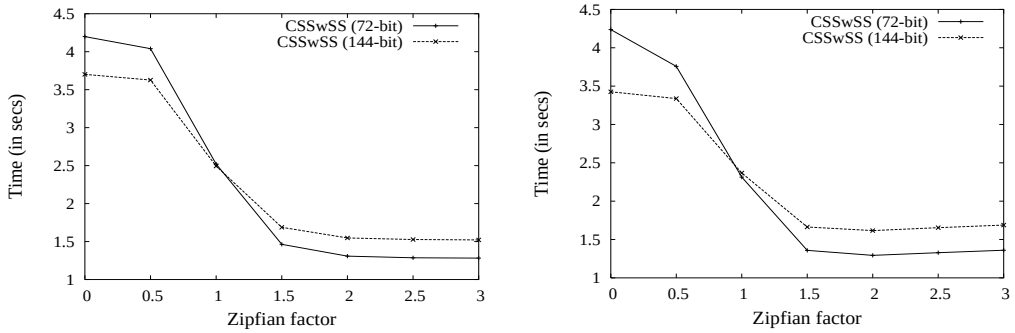
Figure 4: Comparing the performance of *CSS* with *CSSwSS* for 72 & 144 bit TCAM core sizes. This figure reports only the time for frequency counting.

Now analyzing the performance of the two alternative designs, it can be seen from Figure 4 that the approach using 72-bit TCAM core performs better for moderate and heavily skewed distributions, whereas the 144-bit design performs better for uniform distribution. This is because for uniform data, the *CAM adapted Stream Summary* structure undergoes a lot of changes, and the overhead due to the added level of indirection for the 72-bit implementation dominates. However, for skewed data, the increased cost of the 144-bit TCAM operations makes the 72-bit implementation cheaper.

Since the zipfian factor ranging between 1 and 2 represents most realistic data sets, from Figure 4 we can conclude that the implementation using 72-bit wide cores would perform better for real data. In addition to the savings in time, another advantage of the 72-bit implementation is that it saves TCAM space which can be a scarce resource.

5.2 Cost of Answering Queries

In this section, we analyze the cost of answering frequent elements and top-k queries using the proposed algorithm. Since the proposed algorithm keeps the elements sorted, it can therefore be easily adapted to answer queries efficiently and incrementally. As *CSS* does not maintain the elements in sorted order, there is no straightforward technique for the incremental reporting of frequent elements. Every time a query arrives, the *best-effort* adaptation of *CSS* would scan through all the counters to report the frequent elements. On the other hand, the *CAM adapted Stream Summary* structure in *CSSwSS* can be leveraged to incrementally report the frequent elements. Section 4.3 provides efficient algorithms to answer queries and also provides a thorough analysis of the cost incurred by these algorithms. In this section, we evaluate these algorithms experimentally.



(a) Frequent elements (b) Top-k

Figure 6: Comparing performance of implementations of *CSSwSS* answering continuous queries.

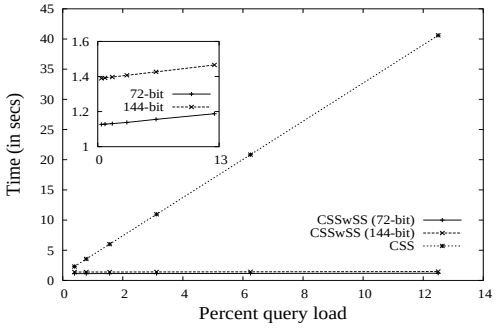


Figure 5: Comparison of times for answering continuous frequent elements queries for input data of Zipfian factor 1.5.

Figure 5 compares the time taken to answer frequent elements queries by the two implementations of *CSSwSS*, and by *CSS*. For this experiment, we vary the query load, and the load is measured as a percentage of queries per update. The efficiency of *CSSwSS* is evident from the fact that even at moderately low query loads of about 10%, it is an order of magnitude faster than *CSS*. We repeated the above experiment for the same range of zipfian factors (0.0 – 3.0), but report the one with zipfian factor of 1.5 as this data set is illustrative of real data. We observed similar behavior for other values of zipfian factor. The graph in the inset compares the two implementations of *CSSwSS* and it can be seen that the 72-bit implementation is a clear winner.

Our proposed algorithms can efficiently answer continuous queries and this is illustrated in Figure 6 where we evaluate the performance of the proposed scheme when answering continuous queries. Figure 6(a) represents continuous frequent elements queries whereas Figure 6(b) illustrates continuous top-k queries. Again, as *CSS* does not sort the elements, it cannot efficiently answer continuous frequent elements and top-k queries, so we do not consider it for this comparison. These results corroborate the analysis performed in Section 4.3. Figures 4 and 6 have similar trends for the two implementations of *CSSwSS*. This is because answering the queries only involves processing the linked structure of frequency buckets, which is identical for both the implementations. Therefore, the 72-bit implementation would be suited for most practical data sets.

5.3 Discussion

In Section 4 we propose efficient CAM conscious algorithms and in Sections 5.1 and 5.2 we evaluate these algorithms to demonstrate the gains over the existing CAM conscious technique. But there are some important implications of the experimental results. *First*, in the experimental prototype used for the experiments in this paper, the cost of a single TCAM operation is about 300ns, which is an order of magnitude higher than the typical TCAM speeds used in simulation of different TCAM algorithms. For example, authors in [20] report typical TCAM `LOOKUP` operation taking around 20ns. The TCAM chip used for our experiments, the IDT75K62134 chip from IDT [13], can support about a 100 million lookups per second, which gives per operation cost of 10ns. From our analysis of the cost of TCAM operations we inferred the following as primary reasons for the difference in performance: the added level of abstraction provided by Teja ADE which is used for implementing the algorithms, the difference in the word sizes of the NPU and the TCAM, and the use of a single Micro Engine to access the TCAM. *Second*, a simple analysis of the algorithms reveals that on an average, for a uniform distribution, every element incurs $5-7$ TCAM operations while for a skewed distribution it incurs $1-3$ TCAM operations. From the times reported in Figure 4 and Table 1 it can be seen that about $50-70\%$ of the frequency counting time is spent on the TCAM operations. Therefore, a decrease in the cost of TCAM operations would considerably reduce the time taken by the CAM conscious algorithms.

Another important point to note is that these times cannot be compared directly with the times of a software hash-based technique running on a standard computer. The primary reason is the difference in the two architectures. As pointed out in Section 3.1, the NPU has a very simple design, whereas most standard processors have higher clock speed and are highly optimized for

single thread performance. But as demonstrated in [2], TCAM does not interface well with conventional processors and requires an Application Specific IC (ASIC) which becomes the communication bottleneck. Therefore, even though TCAM provides good potential for designing efficient stream management algorithms, practical limitations restrict the gains obtained from a real experimental setup such as the NPU-TCAM prototype used in this paper.

6. CONCLUSION

In this paper, we propose a CAM conscious integrated solution for answering frequent elements and top-k queries. We provide an implementation of the proposed algorithm using commodity CAM chip. Evaluation using realistic synthetic data sets show that CSSwSS performs an order of magnitude better compared to the existing technique even with moderate query loads of about 10%. Therefore we can see that the proposed scheme can efficiently answer frequent elements queries. In addition, CSSwSS can efficiently answer continuous queries as well. We analyze two design alternatives for implementing our proposed scheme, and from the experiments, we conclude that the implementation using 72-bit TCAM core performs better for non-uniform data distributions.

Although we use the NPU for our implementation, we do not leverage the parallelism provided by the NPU and supported by the TCAM. In the future, we plan to exploit this parallelism to further improve the processing rates. In addition, we plan to explore the possibility of using TCAM for other stream management operations.

7. REFERENCES

[1] N. Bandi, A. Metwally, D. Agrawal, and A. E. Abbadi. Fast data stream algorithms using associative memories. In *SIGMOD*, pages 247–256, Beijing, China, 2007.

[2] N. Bandi, S. Schnieder, D. Agrawal, and A. E. Abbadi. Hardware Acceleration of Database operations using Content Addressable Memories. In *DaMoN*, 2005.

[3] B. Bhattacharjee, N. Abe, K. Goldman, B. Zadrozny, V. R. Chillakuru, M. del Carpio, and C. Apte. Using secure coprocessors for privacy preserving collaborative data mining and analysis. In *DaMoN '06*, 2006.

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

[5] J. Cieslewicz and K. A. Ross. Adaptive Aggregation on Chip Multiprocessors. In *VLDB 2007*, pages 339–350, 2007.

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

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

[8] R. Fang, B. He, M. Lu, K. Yang, N. K. Govindaraju, Q. Luo, and P. V. Sander. Gpuqp: query co-processing using graphics processors. In *SIGMOD '07*, pages 1061–1063, 2007.

[9] M. Fischer and S. Salzberg. Finding a majority among n votes: Solution to problem 81-5. In *Journal of Algorithms 3(4)*, pages 376–379, 1982.

[10] B. T. Gold, A. Ailamaki, L. Huston, and B. Falsafi. Accelerating database operators using a network processor. In *DAMON '05*, 2005.

[11] M. Greenwald and S. Khanna. Space-efficient online computation of quantile summaries. In *SIGMOD*, 2001.

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

[13] Integrated Devices Technologies, Integrated IP Co-processor IDT 75K62134. `http://idt.com/?genID=75K62134&source=products_genericPart_75K62134`, 2006.

[14] Intel Internet Exchange Architecture for Network Processors. Technical report, Intel Corp., 2002.

[15] Intel IXP 2800 Network Processor Product Brief. `http://www.intel.com/design/network/prodbrf/279054.htm`, 2006.

[16] G. S. Manku and R. Motwani. Approximate frequency counts over data streams. In *VLDB*, 2002.

[17] 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.*, 31(3):1095–1133, 2006.

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

[19] R. Panigrahy and D. Thomas. Finding Frequent Elements in Non-Bursty Streams. In *ESA*, pages 53 – 62, October 2007.

[20] D. Shah and P. Gupta. Fast updating algorithms for tcams. *IEEE Micro*, 21(1):36–47, 2001.

[21] Teja networking systems and teja np application development platform. `http://www.teja.com`, 2006.

[22] S. Venkataraman, D. Song, P. Gibbons, and A. Blum. New streaming algorithms for fast detection of superspreaders. In *NDSS*, 2005.

[23] J. Zhou, J. Cieslewicz, K. A. Ross, and M. Shah. Improving database performance on simultaneous multithreading processors. In *VLDB '05*, pages 49–60, 2005.

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

[25] M. Zukowski, S. Héman, and P. Boncz. Architecture-conscious hashing. In *DaMoN '06*, 2006.