A Heaviest Hitters Limiting Mechanism with $O(1)$ Time Complexity for Sliding-Window Data Streams

Remous-Aris Koutsiamanis, Pavlos S. Efraimidis
Department of Electrical and Computer Engineering, Democritus University of Thrace, Greece
{akoutsia,pefraimi}@ee.duth.gr

Abstract

In this work we address the problem of identifying and limiting the heaviest hitters in a sliding-window data stream. We propose the first, to our knowledge, *exact* (i.e., not approximate) algorithm which achieves $O(1)$ with high probability time complexity in both update and query operations. Additionally, it tracks the first and last item of any itemset in the window in $O(1)$ time complexity as well as the lightest hitters with no additional computational costs. These properties allow us to efficiently implement a mechanism to limit the heaviest hitters by evicting them from or not allowing them in the window. We describe the algorithms and data structure which implement this functionality, we explain how they can be used to accomplish the goal of limiting the heaviest hitters and perform experiments to produce quantitative results to support our theoretical arguments.

Keywords: Mining, Heaviest hitters, Data streams, Sliding window, On-Line algorithms.

1 Introduction

In this paper, we aim to combine a novel algorithm for identifying the heaviest hitters in a sliding-window data stream with the ability to track the items in that sliding window in order to implement the fair rate-limiting mechanism described in [1-2]. This results in a constant time algorithm which is able to fairly distribute the shared service resource to the incoming items.

The sliding-window data stream model is very similar to a traditional limited-size queue, used frequently in network routers to buffer packets while they await service. This is the motivating problem we used to implement and evaluate our algorithms and data structures. More generally, however, the problem of finding the heaviest hitters in a data stream, i.e., the problem of finding which category of items in a long succession of them are the most frequent ones, has a number of applications, some of them quite pervasive. Some applications are in financial data streams, where it is useful, for example, to know which stocks are showing the most mobility. Other applications include sensor networks (for example, helping an intrusion detection scheme [3]) and filtering sensed data, behaviour analysis on websites and trend tracking of hot topics (for example, accurately counting the hottest queries for caching [4]).

The motivating application, as mentioned, is network traffic monitoring (and shaping) on Internet routers. Being able to tell at any moment in time which set of packets is the most frequent passing through a router (collectively referred to as a flow of packets) helps in both being able to tell what may be causing problems and subsequently resolving these problem in a "fair" manner towards those not contributing to the problem. In this paper, we specifically address this issue by implementing the Prince queue policy [1-2]. This policy has been shown to be able to successfully and fairly limit aggressive flows which send service requests, in our case packets, at a rate higher than the fair share they should request in order not to disadvantage other non-aggressive flows. To solve this problem we create a data structure and a set of associated algorithms which operate on it to solve the heaviest hitters problem on the network router queue. The basic heaviest hitters problem consists of a data stream where at each moment in time one item, which belongs to some itemset, arrives for processing. The goal is to be able to provide a list of the itemsets whose item counts are above a given θ threshold. Given the unbounded number of itemsets and length of the data stream, this cannot be achieved without unbounded memory. As a result, all of the proposed solutions for this problem have provided approximate results.

We address a variant of the basic problem in this work which stems from the observation that only a section of the whole history of the data stream may be interesting. Usually, the most recent items are considered to be more important. This is one of the most common and arguably one of the most useful of these variations: finding the heaviest (and lightest) hitters in a sliding-window data stream.

In the sliding window model, at each moment in time the maximum number of items which participate in a window over the data stream is constant. This window contains at most the Q most recent items. This scenario resembles the operation of a queue with an upper limit on its capacity. As items arrive to be processed they are inserted at the end of the queue and as items are processed they are removed from the front of the queue.

*Corresponding author: Remous-Aris Koutsiamanis; E-mail: akoutsia@ee.duth.gr
DOI: 10.6138/JIT.2013.14.1.12

All the algorithms proposed for both the basic problem and the sliding window variation have in common the requirement that they be able to operate on-line. This entails being able to do only one pass over the data, i.e., each arriving item may be examined only once by the algorithm. This is usually called an update operation and the complexity of this operation must be constant time. Furthermore, querying for the heaviest hitters must also be as fast as possible, ideally proportional to the number k of the heaviest or lightest hitters that we request to be found.

Our algorithm supports the ability:

(1) To provide *exact* results in the query operation and at the same time maintain constant time update and query operations.

(2) To provide not only the heaviest but also the lightest hitters in the sliding window with the same performance and no overhead.

In the following sections we first describe the related work (Section 2) and then move on to describe the proposed abstract data type of HL-HITTERS and the building blocks out of which it is constructed (Section 3). We then describe the data structure itself and the algorithms which implement the HL-HITTERS operations. Subsequently, we present the results of the experimental evaluation of the proposed solution (Section 4) and discuss its results (Section 5). Finally, we propose some interesting possible extensions to this work (Section 6).

2 Related Work

This work merges the results from two separate fields to achieve our goals. The first field relates to the fair and balanced distribution of resources (and in this case specifically network router resources) to competing entities. In this field, network congestion has been described game-theoretically by Nagle [5] and the solution put forth used a market wherein the rules of the game would lead to the optimal strategy for the individual entities also being the optimal solution for the system. In a later work, Shenker [6] describes the relation between the selfish entities and the switch service mechanisms and proposes a method of guaranteeing efficient and fair operating points. Since then, the coordination of Internet entities has been modelled through various game definitions [7-8]. We use the model proposed by [1-2] in order to achieve the fair and balanced distribution of resources.

The second field relates to the heaviest hitters problem and its solution in a sliding-window data stream context. This problem was first posed by Moore in 1980 and together with Boyer they presented the solution (in [9]) for finding the majority hitter in the basic version of the problem, i.e., non-window-based data streams. This problem was studied and approximate solutions were proposed much later and concurrently by [10-11]. Since, a significant body of work has been performed on both the basic problem and on its numerous variations. A good presentation of this work can be found in [12-13].

This work builds on our previous effort [14] to implement an efficient heaviest hitters tracking algorithm by extending the data structure to handle the tracking of individual items in the queue, the ability to add a new tracked item and remove one in constant time. We have also performed a more extensive evaluation of the performance of the augmented data structure, improved on the previously reported performance achieved and verified the fairness of the rate-limiting algorithm.

3 Proposed Abstract Data Type

In order to provide an accurate description of our algorithm and the accompanying data structure we describe here its interface. The abstract data type which we define supports the operations shown in Table 1. All the operations in our HL-HITTERS implementation have constant time complexity.

Table 1 The HL-HITTERS Abstract Data Type

Operation	Input	Output	Description
Initialize	-	-	Initializes the ADT
Append	Item	-	Records a new item into the counts
Expire	Item	-	Removes an item from the counts
QueryHeaviest	k: Int	Array[k]	Gets the heaviest-k ItemSets
QueryLightest	k: Int	Array[k]	Gets the lightest-k ItemSets
GetOldestItem	ItemSet	Item	Finds oldest item
GetNewestItem	ItemSet	Item	Finds newest item

3.1 Building Blocks

To implement the data structure we use common basic building blocks. More specifically, we use exactly one array of fixed size, multiple doubly linked lists and one hash table. With each of these data structures we only use the constant time operations. Thus, for example, we never iterate over the nodes of the linked list to reach a sought entry, rather we keep references to the node itself. We will proceed by describing exactly which operations will be used on each data structure and its time complexity.

3.1.1 Array

The array must be of size Q, the same as the size of the window, and its size remains constant during the execution

of the algorithm. We only perform the operations Get and Set on the array, which execute in constant time. The elements of the array are never iterated over.

In the implementation for our experiments we used the standard vector provided by the C++ STL (Standard Template Library) std::vector class.

3.1.2 Doubly-Linked List

The linked lists start out empty and as the algorithm executes nodes are added and removed. We only use the *Head* and *Tail* fields of the doubly-linked list to access the respective nodes in constant time. As far as the inserts and deletes are concerned, they are always executed with respect to a reference node and as such are constant time as well. To be more specific, InsertBefore and InsertAfter require two arguments: the new node to insert and a reference node before or after which to insert the new node. Similarly, Delete requires a direct reference to the node to delete. Furthermore, the maximum number of nodes is known a priori to be Q, and thus we can eliminate the overhead of dynamic memory allocation for the nodes by using a preallocated node pool.

In the implementation for our experiments we used the a custom doubly-linked list implemented by using the Boost intrusive list [15] and a simple pool allocator to avoid all list node memory allocations and deallocations during the operation of the algorithm.

3.1.3 Hash-Table

In the HL-HITTERS data structure the id of each itemset with at least one item in the window, is stored in a dynamic dictionary. A hash-table is used to implement the dynamic dictionary. Hashing is commonly assumed to require $O(1)$ amortized time for the operations Get, Set and Delete or at least for one of these operations. However, there are at least two examples of hashing schemes which achieve worst case $O(1)$ time with high probability (whp): the early work of [16] and the recent algorithm of [17]. Consequently, we can assume that an efficient, $O(1)$ hashing scheme can be used in the HL-HITTERS data structure.

There is an additional reason why we can assume $O(1)$ time for our hashing scheme. Given that our original motivation were router queues, we can assume that the maximum size of a window does not typically exceed 1000 items (packets in this case). The most common values are a few hundred items. This fact admits us the luxury to run the hashing data structure with a very low load factor. For example, even a hash table with 1 million entries would not be a significant cost for a modern router.

Consider the following naive approach with chained hashing using a uniform hashing function with n hash table entries, $m << n = cm$ packets, and k, the constant upper bound on the number of collisions. The probability ρ of experiencing more than k collisions in any of the n table entries is

$$\rho \le n.\binom{m}{k+1}\left(\frac{1}{n}\right)^{k+1} \le n\left(\frac{e}{k+1}\right)\left(\frac{1}{c}\right)^{k+1} \quad (1)$$

For $n = 10^6$, $m = 10^3$ and $k = 10$ the first inequality gives that $\rho \le 2.38 \times 10^{-35}$. Consider now a router which serves 10^9 packets per second (a bit unrealistic today but allows for future enhancements) and operates continuously for 20 years. This router can serve not more than $Z = 10^9 \times 60 \times 60 \times 24 \times 366 \times 20 \le 6.34 \times 10^{17}$ packets during its lifetime. Even if we consider the case where every one of these Z packets is unique, i.e., the router never receives two packets from the same flow and thus maximizes the potential for collisions to appear, the probability of a "bad" collision event occurring during its lifetime is $\rho * Z \le 2.38 \times 10^{-35} \times 6.34 \times 10^{17} = 1.51 \times 10^{-17}$. This probability is thus practically negligible. Consequently, even the naive approach seems to meet the requirements for a router. In addition to this naive implementation there are many, very efficient, hashing schemes which will perform much better.

Unfortunately, however, in practice a standard cuckoo hash table occasionally experiences insertion operations that take significantly more time than the average. The question of which of the published hashing schemes offers the optimal trade-off between space redundancy and worst case bounds could be an interesting problem to investigate. However, for our purposes, any lightweight hashing scheme will be sufficient if sufficient memory is provided. Moreover, for our main motivation application, special hardware-based memory is available in many routers which can achieve de-amortized $O(1)$ performance [18].

Based on the above arguments, we plausibly assume that we can employ an efficient $O(1)$ whp hashing scheme for our data structure in a modern network router. Additionally, we believe that the arguments used for the router case can apply to other applications of window-based heaviest and lightest hitter problems. In the implementation used for the experiments of this work, we used chained hashing provided by the C++ boost::unordered_map class [19].

3.2 Data Structure

We now proceed to describe how the data structure is composed out of the basic building blocks. An overview of the layout used is presented in Figure 1. It should be noted that the Queue is not part of the HL-HITTERS data structure itself but is displayed in order to illustrate the pointers to the items it contains stored in the data structure.

Before proceeding with the description of the data structure further, we need to describe two types of simple record-like structures which are used:

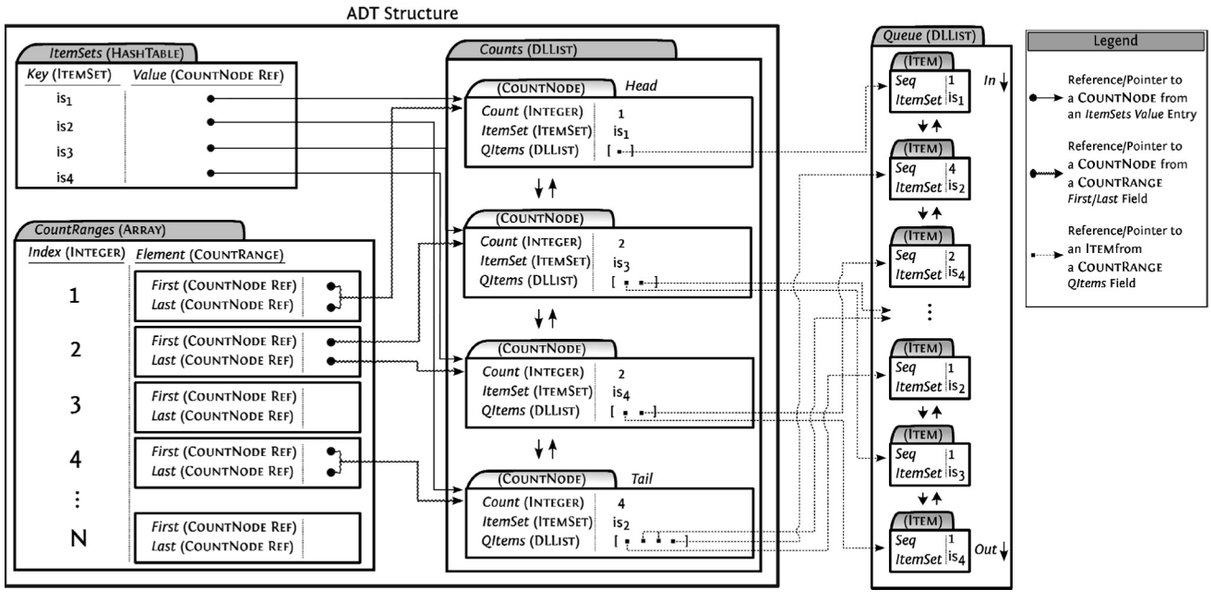

Figure 1 The ADT's Structure

- CountNode, which is the type of the list node used in the doubly-linked list. The data stored (besides the *Previous* and *Next* fields) is an integer named *Count*, the identifier of an ItemSet named *ItemSet* and a linked list of references to items in the queue named *QItems*.
- CountRange, which has two fields, named *First* and *Last*, both of which are references to a doubly linked list node of type CountNode. This structure is meant to store the endpoints of a sub-range of the *Counts* DLList. To support this, it supports two simple operations: Insert (a new node in range) and Remove an existing node from the range. Both are $O(1)$ operations as they manipulate only the *First* and *Last* fields and do not iterate over the nodes in the range.

Layout of the Data Structure: Itemsets that have no items in the window, i.e., a count of zero, will not have any entries in any of the data structures. Conversely, each itemset which has at least one item in the window, i.e., a count ≥ 1, will have one entry in the *ItemSets* HashTable. Additionally, for each itemset, there will exist one node of type CountNode in the *Counts* DLList, with a *Count* field corresponding to its exact count of items in the window and a *QItems* field containing pointers to its items in the queue. Finally, for each group of itemsets which have the same item count there will be one entry in the *Ranges* Array, in the position of the array which is equal to the itemset group's count.

3.3 Algorithms

We now present the operations which are supported by the data structure using pseudo-code and describe their operation and computational complexity in detail.

3.3.1 Initialization

The Initialize operation is shown in Algorithm 1. While its functionality is simply to initialize the *ItemSets* hash table, the *Counts* doubly linked lists and the *Ranges* array, it is useful nevertheless to illustrate that initialization is straightforward and that only memory allocations are performed. For the DLList, the allocation of the node pool is also performed here.

Algorithm 1: The `Initialize` operation

```
1: procedure Initialize
2:     ItemSets ← new HashTable
3:     Counts ← new DLList
4:     Ranges ← new Array
5: end procedure
```

3.3.2 Append

In Algorithm 2 we present the Append operation. It receives the item which is to be appended as a parameter. The itemset of the item is looked up in the *ItemSets* hash table. If it is found, then the itemset is already being counted, i.e., has other items in the window, and therefore its count must be increased by one. If not, then it is a new itemset, i.e., it has no other items in the window, and thus must be recorded with a count of one and a pointer to item in the queue has to be stored.

For the case of being already counted, only the *Counts* and the *Ranges* structures will be modified. The idea is to move the count node corresponding to the itemset to the position in the *Counts* linked list where it will be the first linked list node with the new count. In order to do this, the count node of the itemset is looked up via the Get operation

Algorithm 2: The `Append` operation

```
1: procedure Append(item: ITEM)
2:   itemset ← item.GetItemSet()
3:   cn ← cn′← null
4:   if itemset ∈ ItemSets then
5:     cn ← ItemSets.Get(key:itemset)
6:     cn′← Ranges.Get(index:cn.Count).Last.Next
7:     Ranges.Remove(node:cn)
8:     Counts.Remove(node:cn)
9:     cn.Count ← cn.Count + 1
10:    cn.QItems.Push(item)
11:    Counts.InsertBefore(before:cn′, ins:cn)
12:    Ranges.Insert(node:cn)
13:  else
14:    qi ← new DLLIST
15:    qi.Push(item)
16:    cn←new
       COUNTNODE (ItemSet:itemset, Count:1,QItems:qi)
17:    Counts.InsertBefore(before:Counts.Head, ins:cn)
18:    Ranges.Insert(node:cn)
19:    ItemSets.Set(key:itemset, value:cn)
20:  end if
21: end procedure
```

on the hash table and a reference to it is stored in *cn*. Before removing the *cn* node from the list, the position in the linked list where it will be moved to is recorded in *cn′*, with help from the *Ranges Last* field. This will point to the immediately next linked list node after the last node with the old count. Subsequently, the count node *cn* is removed from the linked list and the corresponding *Ranges* count range entry is updated with the Remove operation. Finally, the *cn* node is inserted in the linked list before the *cn′* node, the new *Ranges* count node entry is updated to include it and a pointer to the item in the queue is pushed at the end of the *QItems* queue (in $O(1)$).

For the case of not being already counted, all of the structures will be modified. A new count node will be created to hold the count for the new itemset. Since allocating a new object on the heap may not be $O(1)$, we can take advantage of the fact that the maximum number of itemsets is Q, as explained in Section 3.1.2, and as such we can just take out a preallocated count node out of a preallocated pool in $O(1)$. A new DLList is created to store the pointers to items in the queue which belong to this itemset and is used in the new count node. This node is then inserted in the position of the *Counts* linked list indicated by the *First* field in the first count range entry of the *Ranges* array and then it is recoded in the same count range entry. Finally, the *itemset* hash table is updated by creating an entry that maps the new itemset to the count node which was created previously using the Set operation.

3.3.3 Expire

In Algorithm 3 we present the Expire operation. It receives the item which is to be removed as a parameter. The item's itemset is looked up in the *ItemSets* hash table via the Get operation and the reference to the count node in the *Counts* linked list representing it is stored in *cn*.

Since the count of the itemset will be decremented by one, we need to move the *cn* count node to the position in the *Counts* linked list where it will be the first linked list node with the new (old minus one) count. Similarly to the Append operation, before removing the *cn* node from the list, the position in the linked list where it will be moved to is recorded in *cn′*, with help from the *Ranges First* field. This will point to the immediately previous linked list node after the first node with the old count. Subsequently, the count node *cn* is removed from the linked list and the corresponding *Ranges* count range entry is updated with the Remove operation. The first item in the count node's *QItems* queue is popped and the count node *Count* field is decremented by one. If the count has not reached zero a check is made to see whether the position to be moved is valid:

- The reference in *cn′* must be not null, which would indicate that the previous count range was the first in the linked list, and
- The count of the *cn′* referenced node must be the same as the new count of the moving node, i.e., the target count node must belong to the correct count range.

If this check succeeds, the new corresponding *Ranges* count range entry is fetched with the Get operation. Its *First* field is set as the new *cn″* insertion position. Afterwards the moving node is inserted there. If the check fails, then there is no CountRange entry in the *Ranges* array corresponding to the new count and the count node is inserted right where the original *cn′* reference pointed to.

In both cases, the moving count node will be inserted in the *Ranges* entry with the new count using the Insert operation.

If the new count after decrementing by one is zero, the count node is deleted. Before doing that, the count node's *QItems* DLList is also deleted and returned to the preallocated pool. If a preallocated pool was used it is returned to the pool in $O(1)$. Finally, the *itemset* hash table is updated by deleting the entry that maps the itemset to the count node which was previously deleted.

3.3.4 Query

In Algorithm 4 we present the QueryHeaviest and the QueryLightest operations simultaneously. The basic algorithm is the same; only the start of the iteration and its direction is different. In the algorithm, the left side of the ↔ symbol corresponds to the QueryHeaviest operation while the right side to the QueryLightest operation.

Algorithm 3: The `Expire` operation

```
1:procedure Expire(item: ITEM)
2:   itemset ← item.GetItemSet()
3:   cn''← null
4:   cn ← ItemSets.Get(key:itemset)
5:   cn'← Ranges.Get(index:cn.Count).First.Previous
6:   Ranges.Remove(node:cn)
7:   Counts.Remove(node:cn)
8:   cn.QItems.Pop(item)
9:   cn.Count ← cn.Count - 1
10:  if cn.Count ≥ 1 then
11:    if cn'≠null and cn'.Count = cn.Count then
12:      cn''← Ranges.Get(index:cn'.Count).First
13:      Counts.InsertBefore(before:cn'', ins:cn)
14:    else
15:      Counts.InsertAfter(after:cn', ins:cn)
16:    end if
17:    Ranges.Insert(node:cn)
18:  else
19:    delete cn.QItems
20:    delete cn
21:    ItemSets.Delete(key:itemset)
22:  end if
23:end procedure
```

Algorithm 4: `Query Heaviest ↔ Lightest` operation

```
1:function QueryHeaviest(k: INTEGER)
2:   results ← newARRAY[k]
3:   cn ←Counts.Tail ↔Counts.Head
4:   i ← 1
5:   while i ≤ k and cn ≠ null do
6:     results[i] ← cn.ItemSet
7:     cn ← cn.Previous ↔cn.Next
8:     i ← i+1
9:   end while
10:  return results
11:end function
```

The algorithm receives the threshold k as a parameter. Initially, a new *results* array of size k is created to hold the results. In some cases, there may be less than k itemsets available, therefore a number of positions at the end of the array will have null entries.

The count node reference *cn* is set to point to the last (for QueryHeaviest) or the first (for QueryLightest) node in the *Counts* linked list via its *Head* or *Tail* fields. Afterwards, an iteration is performed up to k times. In each step, the current itemset stored in the node referenced by *cn* is stored in the current (the i-th) index of the array. Finally, the result is returned.

The whole operation makes up to k iterations, at each one adding a different itemset to the result. This makes this operation have a time complexity of $O(k)$ and as such is constant time as well. The operation of the query algorithm can easily be extended without changing the computational complexity to also return the actual count of each itemset along with each itemset. In addition it is possible instead of specifying a k parameter to return all the itemsets with the highest/lowest count. To implement this, retrieve the *Tail*/*Head* count node of *Counts*, get the highest/lowest count, access the *Ranges* entry corresponding to that count and get the range of count nodes between the *First* and *Last* fields with the max/min count. This algorithm's computational complexity will depend on the number of itemsets which will be the max/min count. As it is possible to have Q itemsets each with a count of one, this algorithm will have a worst case complexity of $O(Q)$. However, in practice in many applications this will seldom be the case. Another extension would be to return the heaviest-θ/lightest-θ hitters, where θ is relative, expressed as a proportion of the window size (e.g., θ = 10%). However, here the QueryHeaviest and the QueryLightest operations will have different complexities. Since there is an upper bound on the number of itemsets which can have a frequency more than or equal to θ equal to $1/\theta$, one can just execute QueryHeaviest with $k = 1/\theta$ and the complexity will be as originally $O(k)$. However, no such bound exists for the QueryLightest case, and therefore its worst case complexity will be $O(Q)$. Finally, if one is willing to accept an $O(Q)$ worst case complexity it is possible to create cumulative versions of both the original and the relative version of the query operations, where the k or θ parameters denote the cumulative count or proportion of the window. This would return the first itemset whose counts together add up to the specified threshold.

3.3.5 GetItem

In Algorithm 5 we present the GetOldestItem and the GetNewestItem operations simultaneously. The basic algorithm is the same; only the retrieved end of a queue is different. In the algorithm, the left side of the ↔ symbol corresponds to the GetOldestItem operation while the right side to the GetNewestItem operation.

The algorithm receives the itemset of which the oldest or newest item in the queue is to be found. Initially, the count node corresponding to the itemset is retrieved from the *ItemSets* hash table. Subsequently, the *QItems* linked list in the count node is accessed and depending on whether the oldest or newest item in the queue is requested, the front or back item in the queue is returned.

Since no iterations are performed and since only the first or last item of the linked list *QItems* is accessed, these operations are performed in $O(1)$.

Algorithm 5: `Get Oldest ↔ Newest Item` operation

```
1:function GetOldestItem(itemset: ITEMSET)
2:   cn ← ItemSets.Get(key:itemset)
3:   item ← cn.QItems.Front() ↔ cn.QItems.Back()
4:   return item
5:end function
```

3.4 Space Complexity

The space complexity of the HL-HITTERS data structure can be fully derived and is exclusively dependent on the maximum window size Q. The *ItemSets* hash table contains a maximum of Q entries, the *Ranges* array has a constant size of Q entries and the *Counts* doubly linked list contains a maximum of Q count nodes. Furthermore, each node in the doubly linked list *Counts*, contains *QItems*, a linked list of pointers to items in the queue. This linked list uses a pool of preallocated nodes which is shared between all the *Counts* nodes. Since there can only at most Q items in queue, the preallocated pool of *QItems* nodes also has a size of Q. It follows that the space complexity of the whole HL-HITTERS data structure is $O(Q)$.

4 Results

It is clear from the previous analysis that the computational complexity of the HL-HITTERS algorithms presented is overall constant time whp. However, this does not guarantee an acceptable level of performance if in practice the constant time required is too high. We have created a router-like scenario, and have performed experiments to gauge the actual performance of the proposed algorithms. We have to note that, to our knowledge, there exists no other algorithm for calculating the heaviest-k hitters exactly, which also provides close to constant time performance. Therefore, we have implemented a naive but efficient as far as possible algorithm to find the heaviest-k hitter. This algorithm, each time the heaviest hitter is requested, creates a hash-table, and records within it the counts for each itemset. As it does this, it keeps track of the running heaviest hitter. However, this algorithm has an $O(Qlogk)$ time complexity, due to the partial (k-largest) sort needed to find the heaviest-k hitters. Furthermore, in the experiments performed, we restricted ourselves to finding the top heaviest hitter only, i.e., $k = 1$, in order not to significantly disadvantage the direct counting algorithm. For reference, the computational complexity of the operations implemented by the direct counting and the HL-HITTERS algorithm is presented in Table 2.

Table 2 Computational Complexity

Operation	Direct counting	HL-Hitters
Initialize	$O(Q)$	$O(Q)$
Append	$O(1)$	$O(1)$
Expire	$O(1)$	$O(1)$
QueryHeaviest	$O(Qlogk)$	$O(1)$
QueryLightest	$O(Qlogk)$	$O(1)$
GetOldestItem	$O(1)$	$O(1)$
GetNewestItem	$O(1)$	$O(1)$

4.1 Experimental Scenarios

The experimental evaluation of our implementation is performed in two distinct scenarios. The first scenario is geared towards evaluating the performance of HL-HITTERS when the queue is full but experiences no dropped packets, i.e., the rate of serving packets from the end of the queue is the same as the rate of arriving packets at the beginning of the queue. Furthermore, this scenario seeks to evaluate how much impact querying to find the heaviest hitter has when it is performed every time a new packet arrives at the queue, since this is what would happen in a real application. Finally, it seeks to measure the impact of tracking the packets which belong to each flow within the queue. This ability will permit the implementation of the Prince policy in the second scenario.

The second scenario aims to measure both the performance and the efficiency of the Prince policy in contrast to a simple FIFO (DropTail) policy when the queue is full and experiences dropped packets, i.e., the rate of serving packets from the end of the queue is higher than the rate of arriving packets at the beginning of the queue. In this scenario, we use two groups of flows, normal and aggressive. The normal flows, which constitute 90% of the total number of flows never send packets at a rate higher than their fair share while the aggressive flows (10% of total flows) always exceed their fair share (within a range of different amounts). As a result, the queue is overflown and needs to drop packets. To compare performance, the Prince policy is implemented by both the naive direct counting algorithm and HL-HITTERS. We measure the time taken to service packets as well as how fairly the policies manage to limit the aggressive flows while not disadvantaging the normal flows.

4.2 Experiment Setup

The implementation has been performed using C++, with standard C++ versions of the building blocks, as described in Section 3.1. We used the G++ compiler with all the optimizations enabled (*-Ofast*) for our specific architecture. The experiments were executed on an Intel Quad Core Q9300 processor with 4 GB of main memory, using one dedicated core for the execution of the experiments. The operating system used was Arch Linux, with the 3.0.1 version kernel. For each result point 10 identical sequential executions of the experiment were performed to remove any bias.

5 Discussion

A selected but representative and indicative of the worst case performance subset of the experimental results are presented here. The source code used to perform the experiments will be available on-line.

5.1 Scenario 1

The results obtained for the first scenario are summarized in Figure 2 where the performance of the direct counting algorithm is compared to the HL-HITTERS algorithm. When counting only, i.e., just keeping track of the count of packets of each flow in the queue the two algorithms perform similarly, whether they also track the positions of the packets in the queue or not. This performance is consistent with the theoretical $O(1)$ complexity given in Table 2 for the Append and Expire operations. However, when querying to find the heaviest hitter ($k = 1$) is introduced (counting needs to be performed as well since without it querying is not possible), the results reflect the $O(Q)$ complexity of direct counting and the $O(1)$ complexity of HL-HITTERS. It is noteworthy to examine the absolute numbers as well. The HL-HITTERS algorithm has a maximum processing time per packet of 0.25 μs. This means that despite using general purpose building blocks and no hardware-based content addressable memory or specialized CPUs, we can process at least 4 million packets per second using our implementation. According to [20] IP packet sizes vary between 40 bytes and 1,500 bytes, with strong polarization tendencies. Given those values, we can achieve a throughput between 1.2 Gbit/sec and 48 Gbit/sec. We stress the fact that this performance is achievable without any specialized hardware as would typically exist in an Internet router. Furthermore, performance profiling has shown that approximately 50% of the processing time is spent on the hash-table operations. Since these would heavily benefit from optimizations on a hardware router, we are confident that significantly higher performance is attainable under such conditions.

5.2 Scenario 2

The results generated from the experiments in the second scenario are displayed in Figures 3 and 4. Figure 3 shows the results of the comparison between a simple FIFO DropTail policy (with no packet tracking) and the HL-HITTERS and direct counting algorithms implementing the Prince policy with packet tracking. The simple FIFO policy is the most performant and is not significantly affected by the increase in total sending rate. The direct counting algorithm slows down linearly with the increase in sending rate and scales badly as the queue size used increases. The loss of performance due to sending rate increase is expected since the QueryHeaviest operation is executed analogously more as well. However, the bad scaling in relation to the queue size leads to unusable performance for a router. Finally, the HL-HITTERS algorithm also slows down as the sending rate increases, at a much lower rate, and scales very well even when the size of the queue is increased. The absolute numbers show that the HL-HITTERS algorithm has a maximum processing time per packet of 0.45 μs when implementing Prince, which as described in the previous paragraph, would accordingly lead to a throughput between 0.7 Gbit/sec and 26 Gbit/sec.

Figure 4 shows the results of the comparison between a simple FIFO DropTail policy (with no packet tracking) and the HL-HITTERS algorithm implementing the Prince policy with packet tracking. These results show that although the FIFO policy is very fast, as seen in Figure 3, it is not able to limit the aggressive players effectively. As the sending rate of the aggressive players increases and the total sending rate as a result increases (since the sending rate of the normal flows is constant) the aggressive players manage to obtain a much higher portion of throughput in

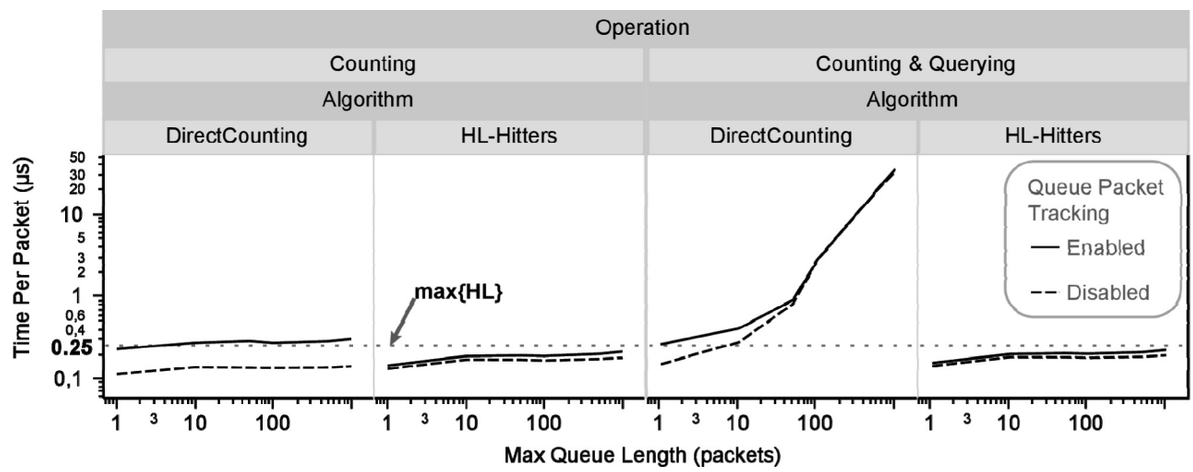

Figure 2 Scenario 1

Note. Performance of HL-HITTERS vs. direct counting for different Q queue lengths and grouped based on operation performed (counting or counting + querying) and on whether the packet positions in the queue are tracked. Measured in mean processing time per packet (shown in μs). The maximum time taken by HL-HITTERS is 0.25 μs.

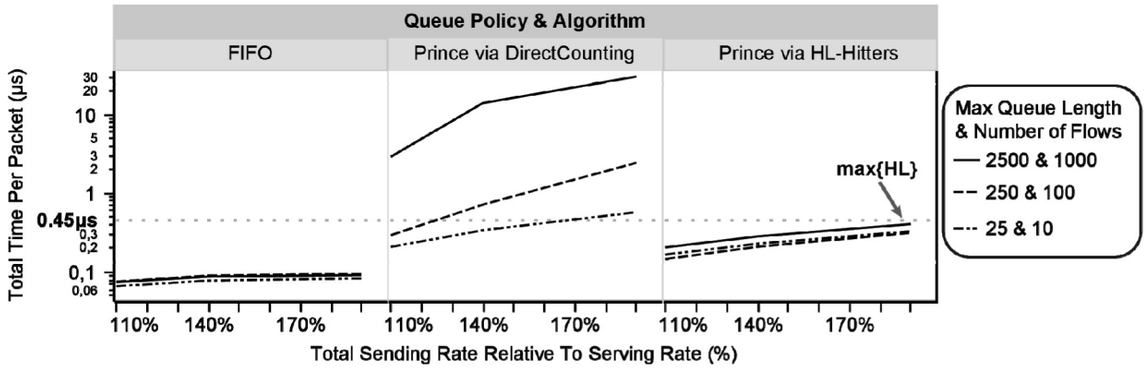

Figure 3 Scenario 2

Note. Performance of simple FIFO (no packet tracking) vs. HL-HITTERS and direct counting implementing the Prince policy. Results shown for different Q queue lengths and number of flows as a function of the total sending rate of the flows vs. the serving rate of the queue. Measured in mean processing time per packet (shown in µs). The maximum time taken by HL-HITTERS is 0.45 µs.

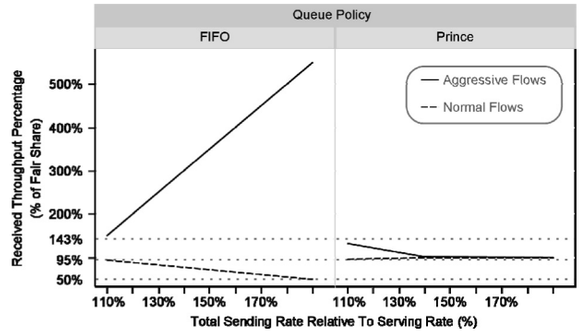

Figure 4 Scenario 2

Note. Measure of policy fairness for the simple FIFO and the Prince policy. The ideal received throughput for both aggressive and normal flows is 100% of their fair share. Here the actual achieved throughput of the aggressive and normal flows is displayed as a function of the total sending rate of the flows vs. the serving rate of the queue. Measured in percent of fair share achieved. For the Prince policy the aggressive flows achieve a maximum of 143% of the fair share and the normal flows a minimum of 95% of the fair share.

respect to the fair share that they should get. For example, when the aggressive players send 10 times faster than the normal players the total sending rate becomes 190% of the service rate and the aggressive players get more than 500% of the fair share while the rest of the 90% of the flows, the normal flows, all receive 50% of the fair share. In contrast, using the Prince policy, the aggressive flows only manage to get 143% of the fair share and as they increase their sending rate they make themselves clearer targets for limiting and are limited even more effectively. At the same time, the lowest share of throughput the normal flows receive is 95% of the fair share.

6 Conclusion

Our work on the problem of the heaviest-k and lightest-k hitters in a sliding-window data stream has resulted in a data structure and an efficient set of algorithms for its operations. These in tandem allow us to achieve constant time updates and queries. Building on this feature, we implement the Prince policy, an effective rate-limiting mechanism, on a simulated router queue and show that it is possible to achieve both a highly performance and extremely fair rate-limiter on a router queue. We have also shown that the performance achieved is high enough in absolute numbers to be used in practical applications. We have attempted to maximize performance on a standard PC while at the same time have found that using a fairly standard component in hardware routers can potentially double performance.

An interesting idea would be to extend this mechanism to incorporate the size of the packets as well, not only their number. This would allow us to make decisions based on the quantity of data that an itemset is responsible for, rather than how many items it is generating. Another direction would be to use multiple HL-HITTERS structures in a queue in parallel, each monitoring a different length of history. This would allow monitoring not only the highest hitters currently in the queue but also in longer periods of time.

Acknowledgements

This research has received funding from the E.U. 7th Framework Programme (FP7 2007-2013) under grant agreement no 264226: SPace Internetworking CEnter -- SPICE. This paper reflects only the views of the authors -- The Union is not liable for any use that may be made of the information contained. We would like to thank Dimitrios Fotakis for our insightful discussions on efficient hashing.

References

[1] Pavlos S. Efraimidis, Lazaros Tsavlidis and George B. Mertzios, *Window-Games between TCP Flows*,

Theoretical Computer Science, Vol.411, No.31-33, 2010, pp.2798-2817.

[2] Lazaros Tsavlidis, Pavlos S. Efraimidis and Remous-Aris Koutsiamanis, *Prince: An Effective Router Mechanism for Networks with Selfish Flows*, *Journal of Internet Engineering*, in press.

[3] Yulia Ponomarchuk and Dae-Wha Seo, *Intrusion Detection Based on Traffic Analysis and Fuzzy Inference System in Wireless Sensor Networks*, *Journal of Convergence*, Vol.1, No.1, 2010, pp.35-42.

[4] David Dominguez-Sal, Marta Perez-Casany and Josep Lluis Larriba-Pey, *Cooperative Cache Analysis for Distributed Search Engines*, *International Journal of Information Technology*, *Communications and Convergence*, Vol.1, No.1, 2010, pp.41-65.

[5] John Nagle, *On Packet Switches with Infinite Storage*, *IEEE Transactions on Communications*, Vol.35, No.4, 1987, pp.435-438.

[6] Scott J. Shenker, *Making Greed Work in Networks: A Game-Theoretic Analysis of Switch Service Disciplines*, *IEEE/ACM Transactions on Networking*, Vol.3, No.6, 1995, pp.819-831.

[7] Aditya Akella, Srinivasan Seshan, Richard M. Karp, Scott Shenker and Christos H. Papadimitriou, *Selfish Behavior and Stability of the Internet: A Game-Theoretic Analysis of TCP*, *ACM SIGCOMM Computer Communication Review*, Vol.32, No.4, 2002, pp.117-130.

[8] Christos Papadimitriou, *Algorithms, Games, and the Internet*, *Proc. of STOC '01*, Heraklion, Greece, July, 2001, pp.749-753.

[9] Robert S. Boyer and J. Strother Moore, *MJRTY -- A fast majority vote algorithm*, February, 1981. Technical Report 32.

[10] Erik D. Demaine, Alejandro Lopez-Ortiz and J. Ian Munro, *Frequency Estimation of Internet Packet Streams with Limited Space*, *Proc. of Algorithms-ESA 2002*, Rome, Italy, September, 2002, pp.348-360.

[11] Richard M. Karp, Scott Shenker and Christos H. Papadimitriou, *A Simple Algorithm for Finding Frequent Elements in Streams and Bags*, *ACM Transactions on Database Systems*, Vol.28, No.1, 2003, pp.51-55.

[12] Hongyan Liu, Yuan Lin and Jiawei Han, *Methods for Mining Frequent Items in Data Streams: An Overview*, *Knowledge and Information Systems*, Vol.26, No.1, 2011, pp.1-30.

[13] S. Muthukrishnan, *Data Streams: Algorithms and Applications*, *Foundations and Trends in Theoretical Computer Science*, Vol.1, No.2, 2005, pp.117-236.

[14] Remous-Aris Koutsiamanis and Pavlos S. Efraimidis, *An Exact and O(1) Time Heaviest and Lightest Hitters Algorithm for Sliding-Window Data Streams*, *Proc. of MUE*, Loutraki, Greece, June, 2011, pp.89-94.

[15] Olaf Krzikalla and Ion Gaztanaga, *Chapter 13. Boost. Intrusive*, 2011, http://www.boost.org/doc/html/intrusive.html

[16] Martin Dietzfelbinger and Friedhelm Meyer auf der Heide, *A New Universal Class of Hash Functions and Dynamic Hashing in Real Time*, *Proc. of ICALP*, Warwick University, UK, July, 1990, pp.6-19.

[17] Yuriy Arbitman, Moni Naor and Gil Segev, *De-amortized Cuckoo Hashing: Provable Worst-Case Performance and Experimental Results*, *Proc. of ICALP*, Rhodes, Greece, July, 2009, pp.107-118.

[18] Kostas Pagiamtzis and Ali Sheikholeslami, *Content-Addressable Memory (CAM) Circuits and Architectures: A Tutorial And Survey*, *IEEE JSSC*, Vol.41, No.3, 2006, pp.712-727.

[19] Daniel James, *Chapter 33. Boost.Unordered*, 2011, http://www.boost.org/doc/html/unordered.html

[20] Rishi Sinha, Christos Papadopoulos and John Heidemann, *Internet packet size distributions: Some observations*, May, 2007. Technical Report ISI-TR-2007-643. http://www.isi.edu/~johnh/PAPERS/Sinha07a/index.html

Biographies

Remous-Aris Koutsiamanis is a PhD candidate at the Department of Electrical and Computer Engineering of the Democritus University of Thrace, Greece. His main interests reside in the fields of algorithmic game theory and streaming algorithms applied to modelling and solving network congestion problems. He has received his MSc in Artificial Intelligence from the University of Edinburgh and his BSc with distinction from the Department of Computer Science of the University of Piraeus, Greece.

Pavlos Efraimidis is an assistant professor in Algorithms at the Department of Electrical and Computer Engineering of the Democritus University of Thrace, Greece. He received his PhD in Informatics in 2000 from the University of Patras under the supervision of Paul Spirakis. His main work is on algorithms and his current research interests are in the fields of algorithmic game theory and algorithmic aspects of privacy.