Received August 19, 2017, accepted September 21, 2017, date of publication October 31, 2017, date of current version November 28, 2017.

Digital Object Identifier 10.1109/ACCESS.2017.2757238

On Frequency Estimation and Detection of Frequent Items in Time Faded Streams

MASSIMO CAFARO[1,2], (Senior Member, IEEE), ITALO EPICOCO[1,2], MARCO PULIMENO[3], AND GIOVANNI ALOISIO[1,2]

[1]Department of Engineering for Innovation, University of Salento, 73100 Lecce, Italy
[2]Euro-Mediterranean Center on Climate Change, 73100 Lecce, Italy
[3]Department of Mathematics and Physics, University of Salento, 73100 Lecce, Italy

Corresponding author: Massimo Cafaro (massimo.cafaro@unisalento.it)

ABSTRACT We deal with the problem of detecting frequent items in a stream under the constraint that items are weighted, and recent items must be weighted more than older ones. This kind of problem naturally arises in a wide class of applications in which recent data is considered more useful and valuable with regard to older, stale data. The weight assigned to an item is, therefore, a function of its arrival timestamp. As a consequence, whilst in traditional frequent item mining applications we need to estimate frequency counts, we are instead required to estimate *decayed counts*. These applications are said to work in the *time fading* model. Two sketch-based algorithms for processing time-decayed streams have been recently published independently near the end of 2016. The Filtered Space Saving with Quasi-Heap (FSSQ) algorithm, besides a sketch, also uses an additional data structure called quasi-heap to maintain frequent items. Forward Decay Count-Min Space Saving (FDCMSS), our algorithm, cleverly combines key ideas borrowed from forward decay, the Count-Min sketch and the Space Saving algorithm. Therefore, it makes sense to compare and contrast the two algorithms in order to fully understand their strengths and weaknesses. We show, through extensive experimental results, that FSSQ is better for detecting frequent items than for frequency estimation. The use of the quasi-heap data structure slows down the algorithm owing to the huge number of maintenance operations. Therefore, FSSQ may not be able to cope with high-speed data streams. FDCMSS is better suitable for frequency estimation; moreover, it is extremely fast and can be used in the context of high-speed data streams and for the detection of frequent items as well, since its recall is always greater than 99%, even when using an extremely tiny amount of space. Therefore, FDCMSS proves to be an overall good choice when considering jointly the recall, precision, average relative error and the speed.

INDEX TERMS Data stream mining, time fading model, frequency estimation, frequent items.

I. INTRODUCTION

In streaming applications we deal with a data stream σ which consists of a sequence of n items drawn from a universe set $\mathcal{U}$. We assume, without loss of generality, that the number of distinct items in σ is D (i.e., $\mathcal{U} = \{1, 2, \ldots, D\}$). The nature of items is strictly dependent on the particular application: indeed, items can be IP addresses, graph edges, points, numbers etc. When the stream size n is reasonably small, an application can store all of the items. However, in typical streaming applications it is generally assumed that n is so large that the items can not be stored and must instead be processed upon their arrival: just one pass over the data stream is allowed. The interested reader may refer to [1], a comprehensive survey of streaming algorithms.

We are interested in the problem of mining frequent items in a data stream. Frequent items, also called *heavy hitters*, have been extensively studied and investigated as witnessed by the huge number of papers published on this interesting topic. Informally, the problem requires determining those items in a stream occurring most frequently. From a practical perspective, given a user's support threshold ϕ, this entails finding all of the items in the input stream whose frequency exceeds ϕn. In the literature, the problem is also referred to as market basket analysis [2], *hot list analysis* [3] and *iceberg query* [4], [5].

Mining frequent items is important not only from a theoretical perspective, but also for its many practical applications, e.g. analysis of web logs [6], computational and theoretical linguistics [7] and the analysis of network traffic [8]–[10].

In order to formally state the problem, let f_i be the frequency of the item $i \in \mathcal{U}$ (i.e., its number of occurrences in the stream σ), $f = (f_1, \ldots, f_m)$ the frequency vector, $0 < \phi < 1$ a support threshold and $||f||_1$ the 1-norm of f (i.e., the total number of occurrences of all of the stream items). Then, an approximate solution to the problem requires determining all of the items i such that $f_i > \phi||f||_1$. Moreover, letting $0 < \epsilon < 1$ (with $\epsilon < \phi$) denote the tolerance on the error committed, an algorithm must not return any item i such that $f_i \leq (\phi - \epsilon)||f||_1$.

Different algorithms for detecting frequent items have been designed, and can be broadly classified as being either *counter* or *sketch* based. In practice, the main difference lies in the data structure used. Counter–based algorithms are deterministic and exploit a fixed number of counters, used to keep track of the items. In this context, a counter is an object storing at least the identity of an item and its associate estimated frequency. The minimum number of counters required to track all of the possible frequent items, expressed as a function of the support threshold ϕ is $1/\phi$. However, taking into account the error bound tied to the ϵ tolerance, the minimum number of counters required is $1/\epsilon > 1/\phi$. Sketch–based algorithms are randomized and provide a probabilistic guarantee. The sketch data structure is usually a bi-dimensional array of cells. Each cell contains a counter variable and stream items are mapped, through pairwise independent hash functions, to their corresponding cells in the sketch.

We deal with the specific streaming model we have described so far. It is commonly referred to in the literature as either *cash register* or *strict turnstile* model [1]; in particular, only *insertions* are allowed (i.e., the weight associated to an item can only be positive). The more general *turnstile* model, also allows *deletions* (i.e. items with a negative weight).

Misra and Gries [11] designed the first sequential, counter–based algorithm. Interestingly, this algorithm was independently rediscovered more than twenty years later by both Demaine *et al.* [8] (this algorithm is now known as the *Frequent* algorithm) and Karp *et al.* [12]. Among the other counters–based algorithms it is worth recalling here *Sticky Sampling*, *Lossy Counting* [13] and *Space Saving* [14], which is widely regarded as the best algorithm in this class. Regarding sketch–based algorithms, notable examples are *CountSketch* [6], *Count-Min* [15], *Group Test* [16], and *hCount* [17].

The need to accelerate the processing of large datasets led to the design and development of many parallel algorithms. The message-passing based algorithms described in [18], [19], and [20] provide parallel versions of Frequent and Space Saving. Shared-memory algorithms have also been designed, including parallel versions of Frequent [21], Lossy Counting [22], and Space Saving [23]–[25]. Novel shared-memory algorithms are described in [26]. GPU (Graphics Processing Unit) based parallel algorithms include [27], [28].

In this paper, we deal with the problem of detecting frequent items in a stream under the constraint that items are weighted, and recent items must be weighted more than former items. This kind of problem naturally arises in a wide class of applications in which recent data is considered more useful and valuable with regard to older, stale data. The weight assigned to an item in the stream is therefore a function of the item's arrival timestamp. As a consequence, whilst in traditional frequent item mining applications we need to estimate frequency counts (i.e., the weight of an item is unitary and the corresponding model is the so-called strict turnstile), we are instead required to estimate *decayed counts*. These applications are said to work in the *time fading* model. A different model, called *sliding window* model [1], [29] has been proposed and investigated as well.

In the time fading model [30]–[32], freshness of more recent items is achieved by *fading* the estimated frequency of older items. A decaying factor $0 < \lambda < 1$ is used to compute an item's *decayed count* (or *decayed frequency*) by means of a decay function assigning greater weight to more recent elements. By using an appropriate decay function, the older an item is, the lower its decayed count shall be. A commonly used decay function provides exponential decay, i.e., the weight of an item occurred n time units in the past is set to λ^n, which is an exponentially decreasing quantity.

Two sketch-based algorithms for mining frequent items in time-decayed streams have been recently proposed and published independently near the end of 2016. The Filtered Space Saving with Quasi-Heap (FSSQ) algorithm [33], besides a sketch, also uses an additional data structure called Quasi-Heap to maintain frequent items. Forward Decay Count-Min Space Saving (FDCMSS) [34], our algorithm, cleverly combines key ideas borrowed from forward decay [35], the Count-Min sketch and the Space Saving algorithm. Therefore, it makes sense to compare and contrast the two algorithms in order to fully understand their strengths and weaknesses with regard to frequency estimation, detection of frequent items and speed.

In order to formally state the problem we are dealing with, we need the following definitions. We assume a stream with discrete time steps labeled as $0, 1, 2, 3, \ldots$ and only one item i arrives at time step $t = 1, 2, 3, \ldots$.

Definition 1 (Exponentially Decayed Count of an Item): Given a time decaying factor $0 < \lambda < 1$, the exponentially decayed count of an item i at time t is $C_t(i) = C_{ut}(i) \times \lambda^{(t-i.ut)} + \alpha$, where $i.ut$ is the last update time for the item i and $C_{ut}(i)$ is the exponentially decayed count of the item i at time $i.ut$ and α is equal to 1 when the item i arrives at time t and is equal to zero otherwise.

Definition 2 (Total Decayed Count of the Stream σ): The Total Decayed Count of a stream σ of size n evaluated at *time $t = n$ is defined as the sum of the exponentially decayed count of all of the items of the stream: $C = \sum_{i \in \mathcal{U}} C_t(i)$.*

We note here that the Total Decayed Count of the stream σ is such that $C = (1 - \lambda^n)/(1 - \lambda)$, a quantity approaching $1/(1 - \lambda)$ for $n \to \infty$.

Definition 3 (Frequent Item Under Decayed Count): Given a stream σ of size n and a support threshold $0 < \phi < 1$, an item i is frequent under decayed count if $C_n(i) > \phi C$.

We are now ready to formally state the problem of ϵ-Approximate Frequent Items under decayed count.

Problem 1 (ϵ-Approximate Frequent Items Under Decayed Count): Given an error tolerance ϵ and a threshold ϕ, determine all of the items i satisfying $C_n(i) > \phi C$, and report no items with $C_n(i) \leq (\phi - \epsilon)C$.

The rest of this paper is organized as follows. We describe the FSSQ and FDCMSS algorithms respectively in Section II and III. We compare and contrast the two algorithms from a theoretical perspective in Section IV, and present experimental results in Section V. We draw our conclusions in Section VI.

II. THE FSSQ ALGORITHM

As its name suggests, the FSSQ algorithm is heavily inspired by the Filtered Space Saving algorithm [36], which uses a bitmap data structure to filter unfrequent items and a list of monitored items to keep track of frequent items. Instead, FSSQ uses a traditional Count-Min based sketch data structure with width w and depth d to filter unfrequent items, and a data structure called Quasi-Heap containing m nodes to monitor frequent items. The Quasi-Heap is still based on the notion of *heap property* for a min-heap, i.e., the key value of a node must be less than or equal to the keys of its children nodes. However, in a Quasi-Heap the heap property is relaxed and does not need to always hold. For instance, if two nodes are swapped the data structure is not maintained as usual by means of the standard *heapify* function. The aim is to avoid running an heapify operation whose worst-case computational complexity when run on m nodes is $O(\log m)$. The Quasi-Heap therefore postpones sorting operations when the decayed count of an existing item is increased; of course, this may result in an inconsistent heap, i.e., the heap property may be violated by some nodes. The *delayedSorting* function, whose pseudocode is shown as Algorithm 1, executes heapify on the Quasi-Heap with the aim of identifying the node whose key is minimum.

A counter object c related to an item stores the item's identity $c.item$, its decayed count $c.cnt$, error $c.error$ and update time $c.ut$. Additionally, assuming that c is not a leaf node, the boolean flag $c.delay$ determines whether or not after updating the node the rest of the operations in *delayedSorting* must be executed: if the delayed flag of an item is not marked, then its count must be the minimum count of its subtree (cfr. [33, Lemma 1]); otherwise, the heap property may be violated. In this case, the function is recursively called on both the children nodes of c and, if the decayed count of the root node is larger than the children, the root node is swapped with its child node whose decayed count is smaller. In the case of identical decayed counts, the root node is swapped with its child only when the child has larger estimated error.

In practice, instead of running heapify for maintenance of the heap structure, items are simply marked as delayed since they are the old items in the Quasi-Heap. Upon arrival of a new item, delayedSorting runs heapify on the delayed nodes, starting from the root.

Algorithm 1 FSSQ Delayedsorting

```
Require: $Qh$, a Quasi-Heap; $c$, a counter (node) in $Qh$; $t$, the current time
Ensure: a Quasi-heap $Qh$
 1: procedure FSSQ-delayedSorting($c$, $t$)
 2:     $c.error \leftarrow c.error \times \lambda^{t-c.ut}$; $c.cnt \leftarrow c.cnt \times \lambda^{t-c.ut}$; $c.ut \leftarrow t$
 3:     if $c.delay = 0$ then
 4:         return
 5:     if $c$ is a leaf node then
 6:         return
 7:     FSSQ-delayedSorting(the left node of $c$, $t$)
 8:     FSSQ-delayedSorting(the right node of $c$, $t$)
 9:     let $sml$ be the smaller of the two child nodes
10:     if ($c.cnt > sml.cnt$) OR ($c.cnt = sml.cnt$ AND $sml.error > c.error$) then
11:         swap $c$ and $sml$; $sml.delay \leftarrow 1$
12:     $c.delay \leftarrow 0$
```

Each cell in the sketch data structure stores a decayed count and the latest update time. The sketch dimensions d and w are initialized as follows: letting $0 < \delta \leq 1$ be a probability of failure, $d = \lceil \ln \delta \rceil$ is the number of rows in the sketch and $w = \lceil e/\epsilon \rceil$ the number of columns.

Assuming that the number of distinct items in the stream σ is $D \leq n$, the sketch cells are updated using d pairwise independent hash functions $h_1, \ldots, h_d$, where $h_i : [D] \to [w], i = 1, \ldots, d$ maps D distinct items into w cells.

FSSQ works as shown in the pseudocode of Algorithm 2. For each incoming item i with timestamp t, if the item is in the Quasi-Heap its error, decayed count and update time are updated as needed. The delay flag is also set. Otherwise (the item is not in the Quasi-Heap), the algorithm determines whether the Quasi-Heap is full or not.

If the Quasi-Heap is not full, a new node c is created to store the item i and the item error, decayed count and update time are set respectively to zero, one and t. The delay flag is set to false. Then, the node is inserted into the Quasi-Heap, with the insertion maintaining the heap property. Otherwise, the *delayedSorting* operation is invoked on the root node, and the sketch cells corresponding to the item i are updated. Then, the algorithm determines est, which is the minimum decayed count among the d cells in the sketch that have been just updated. If est is greater than the decayed count of the

Algorithm 2 FSSQ Update

```
Require: s, a sketch; Qh, a Quasi-Heap; i, an item; t, the current time
Ensure: a Quasi-Heap Qh containing frequent items
 1: procedure FSSQ-update(s, Qh, i, t)
 2:     if i is tracked by Qh then
 3:         let c be the node monitoring i
 4:         $c.error \leftarrow c.error \times \lambda^{t-c.ut}$; $c.cnt \leftarrow c.cnt \times \lambda^{t-c.ut} + 1$; $c.delay \leftarrow 1$; $c.ut \leftarrow t$
 5:     else                                    ▷ i is not tracked by Qh
 6:         if Qh is full and r is the root node of Qh then
 7:             FSSQ-delayedSorting(r, t)
 8:             for $j = 1$ to d do
 9:                 $s[j, h_j(i)] \leftarrow s[j, h_j(i)].cnt \times \lambda^{t-s[j,h_j(i)].ut} + 1$
10:                 $s[j, h_j(i)].ut \leftarrow t$
11:             $est \leftarrow min_{1\le j\le d} s[j, h_j(i)].cnt$
12:             if $est > r.cnt$ then
13:                 for $j = 1$ to d do
14:                     $s[j, h_j(r.item)] \leftarrow r.cnt \times \lambda^{t-r.ut}$
15:                     $s[j, h_j(r.item)].ut \leftarrow t$
16:                 $r.item \leftarrow i$
17:                 $r.error \leftarrow r.cnt \times \lambda^{t-r.ut}$; $r.cnt \leftarrow r.cnt \times \lambda^{t-r.ut} + 1$; $r.delay \leftarrow 1$; $r.ut \leftarrow t$
18:         else                                ▷ Qh is not full
19:             create a new counter c
20:             $c.item \leftarrow i$; $c.error \leftarrow 0$; $c.cnt \leftarrow 1$; $c.delay \leftarrow 0$; $c.ut \leftarrow t$
21:             insert and maintain c in Qh
```

root node r, the sketch is updated again, considering the item monitored by the root node. Then, the root node is updated as well, and the identity of its monitored item is changed to reflect the fact that the node is now tracking the item i. Moreover, the root node delay flag is set to true and its update time is set to t.

Frequent items are retrieved by posing a query to the Quasi-Heap. For frequency estimation, if an item is not in the Quasi-Heap, then the sketch is queried as in Count-Min.

III. THE FDCMSS ALGORITHM

FDCMSS cleverly combines key ideas borrowed from forward decay, the Count-Min sketch and the Space Saving algorithm. We begin by noting that, even though FDCMSS is based on the concept of forward decay instead of backward decay, which is used in FSSQ, we can nonetheless compare the algorithms owing to the fact that backward and forward exponential decay coincide [35]. In the following, we provide a description of FDCMSS based on backward exponential decay; details related to forward decay can be found in [34]. FDCMSS uses an augmented sketch data structure, in which each cell contains a Space Saving stream summary with two counters. The key idea is to work as in the Count-Min sketch algorithm, but to rely on the Space

Algorithm 3 Space Saving Update

```
Require: $\mathcal{S}$, a stream summary; j, an item; w, the weight of item j
Ensure: a stream summary $\mathcal{S}$ containing frequent items
 1: procedure SpaceSavingUpdate($\mathcal{S}$, j, w)
 2:     if j is monitored then
 3:         let $c_l$ be the counter monitoring j
 4:         $c_l.f \leftarrow c_l.f + w$
 5:     else
 6:         if there is a counter $c_r$ which is not monitoring any item then
 7:             $c_r.i \leftarrow j$
 8:             $c_r.f \leftarrow w$
 9:         else
10:             let $c_s$ be the counter monitoring the item with least hits
11:             $c_s.i \leftarrow j$
12:             $c_s.f \leftarrow c_s.f + w$
```

Algorithm 4 FDCMSS Update

```
Require: i, an item; $t_i$, timestamp of item i;
Ensure: update of sketch related to item i
 1: procedure FDCMSS-update(i, $t_i$)
 2:     $x \leftarrow \lambda^{-t_i}$     ▷ compute the decayed weight of item i and update the sketch
 3:     $count \leftarrow count + x$     ▷ update the total decayed count of the stream
 4:     for $j = 1$ to d do
 5:         $\mathcal{S} \leftarrow s[j][h_j(i)]$
 6:         SpaceSavingUpdate($\mathcal{S}$, i, x)
```

Saving stream summary to allow for simultaneous better frequency estimation and tracking of frequent items. Therefore, FDCMSS does not need any additional data structure to keep track of frequent items. We now describe how FDCMSS works.

FDCMSS initializes its sketch data structure $s[x, y]$ by setting the dimensions d and w as follows: $d = \lceil \ln 1/\delta \rceil$ is the number of rows in the sketch and $w = \lceil \frac{e}{2\epsilon} \rceil$ the number of columns.

Each of the sketch cells available stores a Space Saving stream summary, i.e., a data structure $\mathcal{S}$ with two counters c_1 and c_2. Given a counter $c_j, j = 1, 2$, let $c_j.i$ and $c_j.f$ be respectively the counter's item and its estimated decayed count. We use d pairwise independent hash functions $h_1, \ldots, h_d$, where $h_i : [D] \rightarrow [w], i = 1, \ldots, d$ maps D distinct items into w cells, and initialize the *count* variable, representing the total decayed count of all of the items in the stream (see Definition 2) to zero.

The sketch is updated upon arrival of an item i with timestamp t_i; the corresponding pseudo-code is shown as Algorithm 4. We compute x, the forward decayed weight of the item, and increment *count* by x. Then, the d cells in which the item is mapped to by the corresponding hash functions are

Algorithm 5 Query

```
Require: $t$, query time; $count$, total decayed count;
Ensure: set of frequent items
1:  procedure query($t$)
2:      $R = \emptyset$
3:      for $i = 1$ to $d$ do
4:          for $j = 1$ to $w$ do
5:              $\mathcal{S} \leftarrow s[i][j]$
6:              let $c_1$ and $c_2$ be the counters in $\mathcal{S}$, and $c_m$ the counter with maximum decayed count
7:              $c_m \leftarrow \text{argmax}(c_1, c_2)$
8:              if $c_m.f > \phi \times count$ then
9:                  $p \leftarrow$ PointEstimate($c_m.i, t$)
10:                 if $p > \phi \times count \times \lambda^t$ then
11:                     $R \leftarrow R \cup \{(c_m.i, p)\}$
12:     return $R$
```

Algorithm 6 PointEstimate

```
Require: $j$, an item; $t$, query time
Ensure: estimation of item $j$ decayed count;
1:  procedure pointestimate($j, t$)
2:      $answer \leftarrow \infty$
3:      for $i = 1$ to $d$ do
4:          $\mathcal{S} \leftarrow s[i][h_i(j)]$
5:          let $c_1$ and $c_2$ be the counters in $\mathcal{S}$
6:          if $j == c_1.i$ then
7:              $answer \leftarrow \min(answer, c_1.f)$
8:          else
9:              if $j == c_2.i$ then
10:                 $answer \leftarrow \min(answer, c_2.f)$
11:             else
12:                 $m \leftarrow \min(c_1.f, c_2.f)$
13:                 $answer \leftarrow \min(answer, m)$
14:     return $answer \times \lambda^t$
```

updated by using the well-known Space Saving item update procedure (shown as Algorithm 3).

To retrieve the frequent items, a query can be posed to the sketch. Let t be the query time. The query, shown in pseudo-code as Algorithm 5, initializes R, an empty set, and then it inspects each of the cells in the sketch. For a given cell, we determine c_m, the counter in the data structure $\mathcal{S}$ with maximum decayed count. We compare the decayed count with $\phi \times count$. If the decayed count is greater, we pose a point query for the item $c_m.i$, shown in pseudo-code as Algorithm 6. If p, the returned value, is greater than $\phi \times count$, then we insert in R the pair $(c_m.i, p)$.

The point query for an item j returns its estimated decayed count. We inspect each of the d cells in which the item is mapped to by the corresponding hash functions, to determine the minimum decayed count of the item. In each cell, if the item is stored by one of the Space Saving counters, we set *answer* to the minimum between *answer* and the corresponding counter's decayed count. Otherwise (none of the two counters monitors the item j), we set *answer* to the minimum between *answer* and the minimum decayed count stored in the counters. Finally, we return *answer*.

From the previous discussion it is clear that our algorithm also solves the *decayed count estimation* problem for arbitrary items. Indeed, given an item, it suffices to pose a point query for that item.

IV. THEORETICAL COMPARISON

In this Section, we compare FSSQ and FDCMSS from a theoretical perspective. We begin by analyzing the worst case computational complexity. For FSSQ with m nodes in the Quasi-Heap and a sketch of dimensions d and w, the *delayed-Sorting* function requires in the worst case $O(m)$, because it may traverse the entire tree. Assuming that searching for an item in the Quasi-Heap is done by using an auxiliary hash table (incurring the expense of additional space), then the search requires $O(1)$ constant time; otherwise, it requires $O(m)$ time since the heap property does not provide any useful information and one must check both subtrees of every node.

Regarding the FSSQ *update*, there are three cases to consider. If the incoming item is in the Quasi-Heap, updating the corresponding node requires $O(1)$ time. If the item is not in the Quasi-Heap and the Quasi-Heap is not full, inserting a new node corresponding to the item requires $O(\log m)$. Finally, if the item is not in the Quasi-Heap and the Quasi-Heap is full, the following operations are done. The *delayed-Sorting* function is invoked, requiring $O(m)$, and the sketch is updated in $O(d)$ time. Computing the minimum value in the sketch associated to the incoming item i requires $O(d)$. The conditional update of the sketch requires again $O(d)$, and replacing the root node in the Quasi-Heap with the new node corresponding to i can be done in $O(1)$ time. Overall, the worst case computational complexity of FSSQ update is therefore $O(m + d)$. Since $d = \lceil \ln \delta \rceil$, the worst case computational complexity can be rewritten as $O(m + \ln 1/\delta)$.

For FDCMSS with a sketch of dimensions d and w, the worst case computational complexity is simply $O(d)$. Taking into account that $d = \lceil \ln \delta \rceil$, we can rewrite it as $O(\ln 1/\delta)$. It is clear that FDCMSS has a better worst case computational complexity with regard to FSSQ, and we shall see in the next Section reporting the experimental results that, in the majority of the cases, FDCMSS outperforms FSSQ with regard to speed, measured as the number of updates/ms.

We now analyze the space required. FSSQ uses a sketch of dimensions $d = \lceil \ln \delta \rceil$ and $w = \lceil e/\epsilon \rceil$. Each cell of the sketch stores a decayed count and the cell update time. Assuming 8 bytes for each field, the sketch requires a total of $16 \left\lceil \frac{e}{\epsilon} \right\rceil \left\lceil \ln \frac{1}{\delta} \right\rceil$ bytes. The Quasi-Heap data structure stores in the worst case m nodes, and each node stores the item identity, its decayed count, error, update time and a delayed flag. We assume 4 bytes for the item identity (in our C++ implementation the item is stored as an unsigned int), 8 bytes for each of the other fields (decayed count is a double, error is a double, update time is a long) and 4 bytes for the boolean delayed flag.

Therefore, we assume a total of 32 bytes for each node. It follows that the Quasi-Heap requires 32 m bytes. Finally, the hash table data structure needed to search for an item in the Quasi-Heap requires $3m \times 8 + 32m = 56\ m$ bytes. Indeed, given m counters, $3m$ buckets are allocated, each storing a 8 bytes pointer to a hash table data structure which holds a key, its value and two pointers (required to navigate the bucket list in case of collisions). Given an item as key, we store as its value the corresponding pointer to a Quasi-Heap node (8 bytes). The whole structure requires 32 bytes (8 for the key, 8 for the value and 8 for each pointer). In the worst case we insert all of the m items into the hash table, each one requiring 32 bytes (the size of the hash table data structure for an element). The choice of allocating $3m$ buckets is arbitrary, but is commonly used in practice. There is a tradeoff between the buckets allocated and the number of collisions: the more the buckets, the less are the collisions and vice-versa. In total, FSSQ needs in the worst case $16 \left\lceil \frac{e}{\epsilon} \right\rceil \left\lceil \ln \frac{1}{\delta} \right\rceil + 88m$ bytes.

On the other hand, FDCMSS uses a sketch of dimensions $d = \lceil \ln 1/\delta \rceil$ and $w = \lceil e/(2\epsilon) \rceil$, and each cell stores 2 Space Saving counters. Each counter requires 4 bytes to track the identity of an item and 8 bytes for its decayed count (in our implementation the item is an unsigned int and the decayed count a double). Therefore, a counter requires 12 bytes and a sketch cell therefore requires 24 bytes. It follows that the space used by FDCMSS in the worst case is $12 \left\lceil \frac{e}{\epsilon} \right\rceil \left\lceil \ln \frac{1}{\delta} \right\rceil$ bytes. Comparing the space required by the two algorithms, again FDCMSS proves to be better than FSSQ.

We now compare the error bounds provided by the two algorithms. For FSSQ, given the sketch dimensions d and w, the real error guaranteed by the algorithm is $\hat{\epsilon} < e/w$ with probability $1 - e^{-d}$ (cfr. [33, Lemma 5]). The error bound guarantee for FDCMSS is the following: $\hat{\epsilon} < e/(2w)$ with probability $1 - e^{-d}$ (cfr. [34, Th. 1]), and therefore, our guaranteed bound is obviously better. It follows immediately, that FDCMSS provides overall better *frequency estimation* than FSSQ. Indeed, we shall see in the experimental results reported in the next Section that, given the same amount of space, FDCMSS outperforms FSSQ with regard to overall accuracy, measured in terms of the mean relative error committed.

Finally, we discuss the *recall* of the two algorithms. Recall is the total number of true frequent items reported over the number of true frequent items given by an exact algorithm. Therefore, an algorithm is correct iff its recall is equal to 1 (or 100%). For FSSQ, recall is guaranteed iff m, the number of counters in the Quasi-Heap data structure is such that $m \geq 1/\phi$. For FDCMSS, we guarantee the recall with probability greater than or equal to $1 - (\frac{1}{2\phi w})^d$ subject to the constraint $2\phi w \geq 1$ (cfr. [34, Th. 2]).

V. EXPERIMENTAL RESULTS

We present and discuss experimental results on both synthetic and real datasets, thoroughly comparing FDCMSS against FSSQ with regard to several metrics.

FDCMSS and FSSQ have been implemented in C++. Since in [33] there is no mention of the hash functions used for the sketch data structure, for fairness we use the *xxhash* hash function used by FDCMSS, and reuse as much as possible the same source code related to the sketch data structure. Moreover, we also implemented in FSSQ the search for an item in the Quasi-Heap by using a hash table in order to make it faster.

The source code has been compiled using the latest version of the Intel c++ compiler v17.0.4 on linux CentOS 7 with the following flags: -O3 -std=c++11. The tests have been carried out on a workstation equipped wth 64 GB of RAM and two 2.0 GHz exa-core Intel Xeon CPU E5-2620 with 15 MB of level 3 cache. The source code is freely available for inspection and for reproducibility of results contacting the authors by email.

A. SYNTHETIC DATASETS

Regarding synthetic datasets, the input distribution used in our experiments is the Zipf distribution. For each different value of n (number of items), ϕ (support threshold), ρ (skew of distribution), λ and budgeted memory, the algorithms have been run 20 times using a different seed for the pseudo-random number generator associated to the distribution (using the same seeds in the corresponding executions of different algorithms). For each input distribution generated, the results have been averaged over all of the runs. The input elements are 32 bits unsigned integers.

In order to provide a fair comparison of the algorithms, we make sure that the decayed frequencies computed by both algorithms are equal. To this end, we use in FDCMSS the same exponential decay function and the same λ parameter. This way, for a given input stream, the decayed counts of the input items and the set of frequent items computed by an exact algorithm are the same for both algorithms.

We compare our algorithm against FSSQ taking into account the following standard metrics: *recall*, *precision*, *mean absolute error* and *updates per millisecond*. For each metric, we plot the values (mean and confidence intervals) obtained varying n, ϕ, ρ, λ and the budgeted memory.

In our implementation, as already discussed in Section IV, FSSQ requires a total amount of bytes given by $S_1 = 16\ d_1\ w_1 + 32m + 56m$ where d_1 and w_1 are the sketch dimensions and m is the number of nodes in the Quasi-Heap, so that $32m$ is the space required by the Quasi-Heap and $56m$ the space required by the hash table. On the other hand, FDCMSS requires $S_2 = 24\ d_2\ w_2$ bytes, where d_2 and w_2 are the sketch dimensions. We set $d_1 = d_2 = 4$ since this value is already enough to amplify the probability of success as needed. Therefore, FSSQ requires a number of bytes equal to $S_1 = 64\ w_1 + 32m + 56m$, whilst FDCMSS requires $S_2 = 96\ w_2$ bytes. Therefore, we need to equate S_1 to S_2 and to determine the correct values to be assigned to w_1 and m for FSSQ and to w_2 for FDCMSS.

Since FSSQ is based on FSS, this is achieved as explained in [36], i.e., the space assigned to the sketch and the

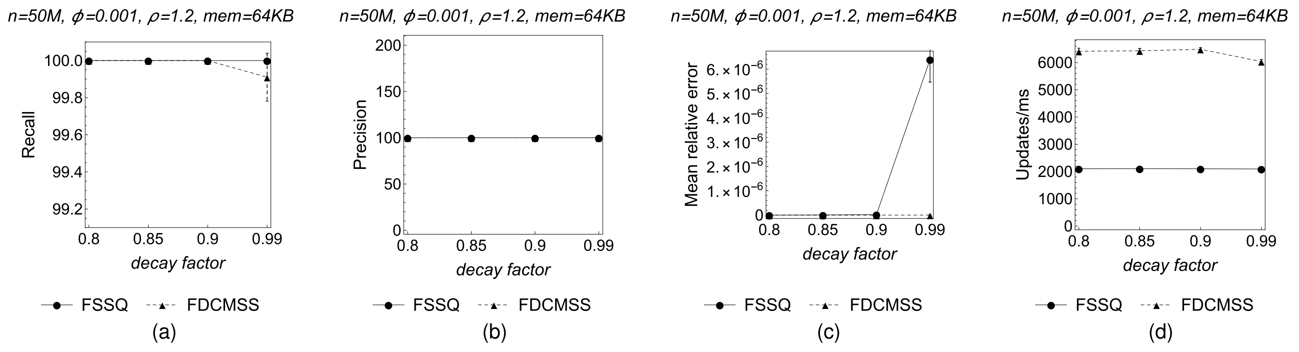

FIGURE 1. Results varying λ decaying factor (mean and confidence interval). (a) Recall. (b) Precision. (c) Average Relative Error. (d) Updates/ms.

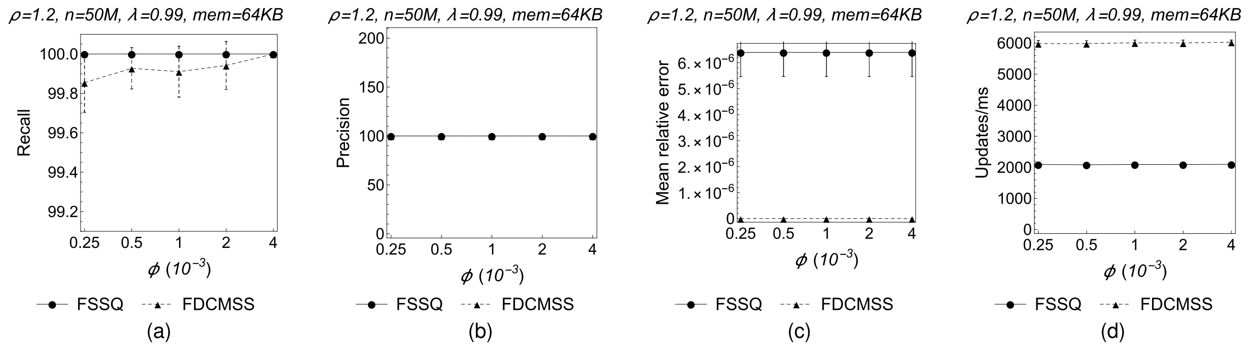

FIGURE 2. Results varying ϕ support threshold (mean and confidence interval). (a) Recall. (b) Precision. (c) Average Relative Error. (d) Updates/ms.

Quasi-Heap is the same: 64 $w_1 = 32\ m$. It follows that $m = 2\ w_1$, therefore we need to compare $S_1 = 64\ w_1 + 32m + 56m = 240\ w_1$ versus $S_2 = 96\ w_2$. From $S_1 = S_2$ we derive immediately the final relation $w_2 = 240/96\ w_1 = 5/2\ w_1$. Therefore, fixing the dimension w_1 in FSSQ we obtain immediately the corresponding values for m and w_2 that provide the same amount of budgeted memory for both algorithms.

Without the hash table, FSSQ can use more space for both its sketch and Quasi-Heap data structures; however, the speed of FSSQ, measured in terms of updates/ms, is always at least an order of magnitude worse than FDCMSS (in all of the experiments carried out, for both synthetic and real datasets), reaching at most a few hundreds of updates/ms. In practice, without the hash table FSSQ can only process relatively slow data streams; for this reason, we do not report the experimental results related to the implementation of FSSQ without the hash table.

Finally, from a practical perspective, taking into account that $w_2 = 5/2\ w_1$, the theoretical bound on the error related to our experiments is therefore for FSSQ $\hat{\epsilon} < e/w_1$ with probability $1 - e^{-d}$ and for FDCMSS is the following: $\hat{\epsilon} < e/(2w_2) < e/(5w_1)$ with probability $1 - e^{-d}$, i.e., the theoretical error bound of FDCMSS is 1/5 of FSSQ.

Table 1 reports the parameters' values and their default in the experiments carried out on synthetic zipf datasets.

Recall is the total number of true frequent items reported over the number of true frequent items given by an exact algorithm. Therefore, an algorithm is correct iff its recall is equal to 1 (or 100%).

TABLE 1. Synthetic data: experiments carried out.

Parameter	Values	Default
n	$10^6, 25 \times 10^6, 50 \times 10^6, 75 \times 10^6, 100 \times 10^6$	50×10^6
ρ	0.7, 1.2, 1.7, 2.2	1.2
ϕ	0.00025, 0.0005, 0.001, 0.002, 0.004	0.001
λ	0.8, 0.85, 0.9, 0.99	0.99
space (KB)	32, 48, 64, 80, 96	64

Precision is defined as the total number of true frequent items reported over the total number of items reported. As such, this metric quantifies the number of false positives outputted by an algorithm. It follows that, from this point of view, an algorithm's precision should ideally be 1 (or 100%).

Denoting with f the true decayed frequency of an item and with $\hat{f}$ the corresponding decayed frequency reported by an algorithm, then the absolute error is, by definition, the difference $\left|f - \hat{f}\right|$. The (absolute) total error is then defined as the sum of the absolute errors. Similarly, the absolute relative error is defined as $\Delta f = \frac{\left|f - \hat{f}\right|}{C}$, where C is the total decayed count of the stream, and the average relative error is derived by averaging the absolute relative errors over all of the measured decayed frequencies.

Figures 1, 2, 3, 4 and 5 depict the experimental results obtained on synthetic datasets when varying respectively λ, ϕ, the budgeted memory space, the skew ρ

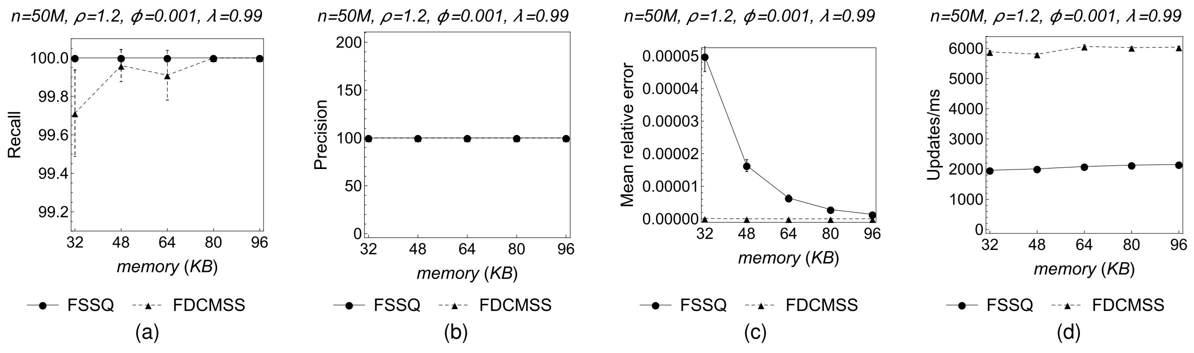

FIGURE 3. Results varying the budgeted space (mean and confidence interval). (a) Recall. (b) Precision. (c) Average Relative Error. (d) Updates/ms.

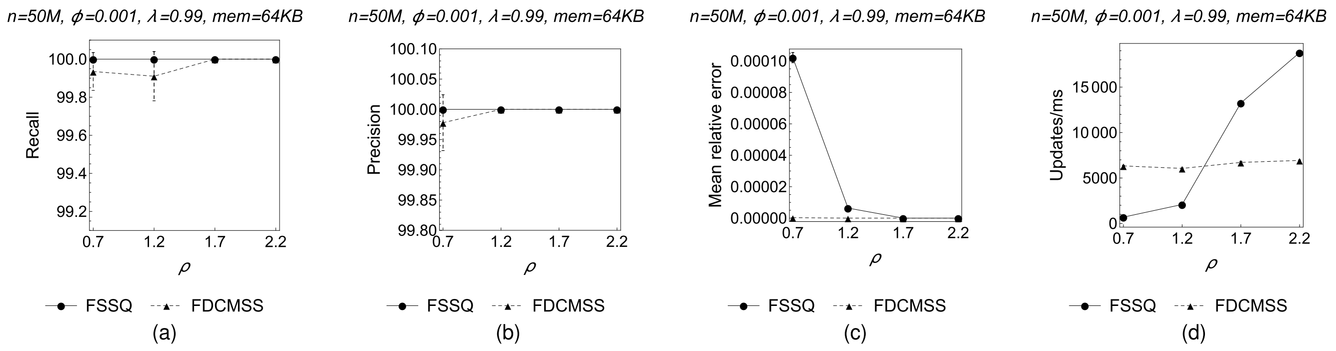

FIGURE 4. Results varying the skew ρ (mean and confidence interval). (a) Recall. (b) Precision. (c) Average Relative Error. (d) Updates/ms.

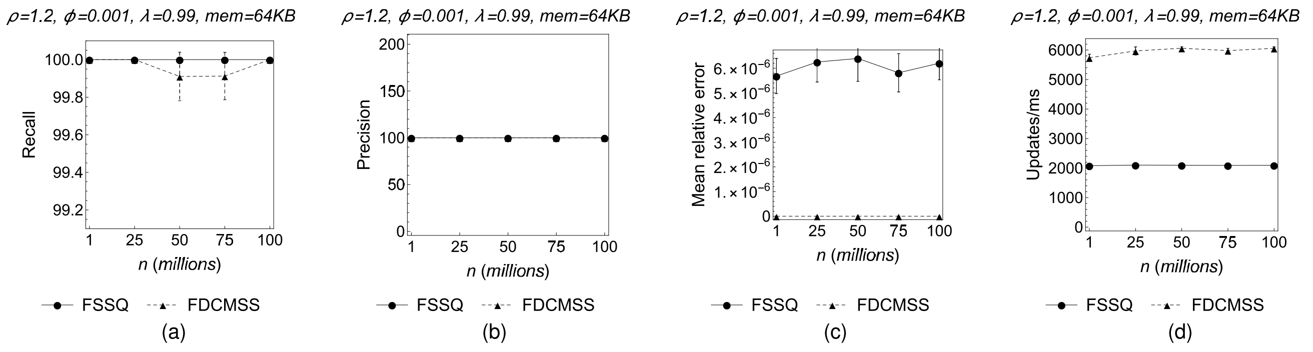

FIGURE 5. Results varying the stream size n (mean and confidence interval). (a) Recall. (b) Precision. (c) Average Relative Error. (d) Updates/ms.

and the stream size n. In our experiments related to synthetic datasets, FSSQ always reached 100% recall and precision. As shown by Figure 3a, FDCMSS requires slightly more space than FSSQ to reach 100% recall. Nonetheless, by using the default space of 64 KB, FDCMSS reaches more than 99.8% of recall, and in all of the cases in which the recall is slightly below 100%, FDCMSS loses at most one frequent item. Regarding precision, FDCMSS always reaches 100% except when the skew is low ($\rho = 0.7$), as shown in Figure 4b.

Regarding the average relative error (henceforth called ARE), FDCMSS clearly outperforms FSSQ, exhibiting smaller values of ARE throughout all of the experiments carried out.

Finally, FDCMSS outperforms FSSQ with regard to speed, measured in terms of updates/ms in all of the experiments in which we vary λ, ϕ, n and the budgeted memory space used. Regarding the experiment in which we vary the skew parameter ρ, FSSQ is faster starting from $\rho = 1.7$. The reason for this behaviour is strictly related to the maintenance of the Quasi-Heap data structure, i.e., the higher the skew, the lesser the number of heapify operations required. In particular, FSSQ speed depends heavily on the fraction of items which are not monitored by the Quasi-Heap and that are processed when the Quasi-Heap is full. In this case, a delayed-Sorting operation is required, slowing down the algorithm's execution.

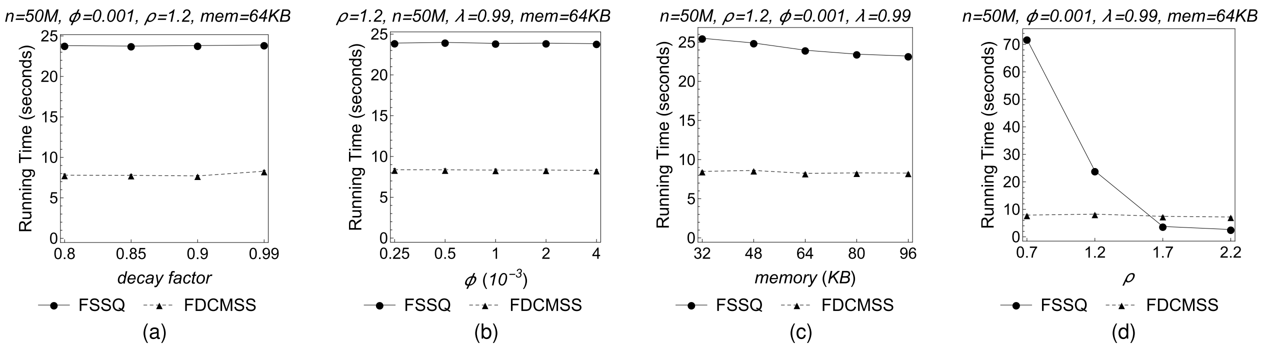

FIGURE 6. Running time varying several parameters (mean and confidence interval). (a) λ. (b) ϕ. (c) Budgeted space. (d) ρ.

TABLE 2. Synthetic datasets: number of delayedSorting operations over the total number of item's updates. Default values have been used for those parameters which are not reported in the table.

Zipfian Distribution	Ratio	Updates/ms
ρ=0.7	94.72%	503
ρ=1.2	19.91%	1850
ρ=1.7	0.56%	8004
ρ=2.2	0.01%	10175

We have experimentally measured the number of times that the delayedSorting is invoked over the total number of updates. Table 2 reports the percentage of delayedSorting invocations with regard to the total number of updates varying the skewness of the Zipfian distribution in the case of synthetic datasets; default values have been used for all of the parameters, except when explicitly specified. The third column of the table refers to the algorithm speed. As expected, the results unequivocally point out that the speed of the FSSQ algorithm is heavily influenced by the number of invocations of *delayedSorting*.

Even though the updates/ms metric is the standard way to measure the performance of this kind of algorithms, for completeness we report in Figure 6 the running times of both FSSQ and FDCMSS when varying respectively λ, ϕ, the budgeted memory and ρ. We note here that the running time of an algorithm (in ms) is easily derived from the corresponding updates/ms value, since n, the length of the stream, is also known. Indeed, letting v be the updates/ms value, the corresponding running time is given by n/v. We report the running time in seconds, except when the running time is below one second, in which case we report it in milliseconds.

Finally, Figure 7 depicts the running time of both FSSQ and FDCMSS when varying n, the length of the stream. Therefore, this plot clearly shows the scalability of the algorithms under test. As can be seen, both algorithms scale linearly, with FDCMSS clearly outperforming FSSQ.

B. REAL DATASETS

The real datasets we used come from different domains. All of the datasets are publicly available, and two of them (Kosarak and Retail) have been widely used and reported in the data mining literature. Overall, the four datasets are characterized by a diversity of statistical characteristics, which we report in Table 3.

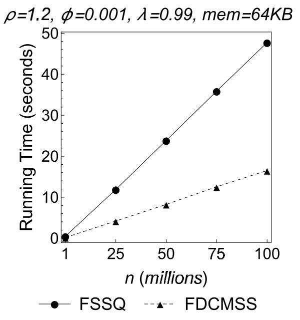

FIGURE 7. Scalability of algorithms: running time varying n (mean and confidence interval).

TABLE 3. Statistical characteristics of the real datasets.

	Kosarak	Retail	Q148	Nasa
Count	8019015	908576	234954	284170
Distinct items	41270	16470	11824	2116
Min	1	0	0	0
Max	41270	16469	149464496	28474
Mean	2387.2	3264.7	3392.9	353.9
Median	640	1564	63	120
Std. deviation	4308.5	4093.2	309782.5	778.1
Skewness	3.5	1.5	478.1	6.5

1) KOSARAK

This is a click-stream dataset of a Hungarian online news portal. It has been anonymized, and consists of transactions, each of which is comprised of several integer items. In the experiments, we have considered every single item in serial order.

2) RETAIL

This dataset contains retail market basket data coming from an anonymous Belgian store. Again, we consider all of the items belonging to the dataset in serial order.

3) Q148

Derived from the KDD Cup 2000 data, compliments of Blue Martini, this dataset contains several data. The ones we use for our experiments are the values of the attribute Request

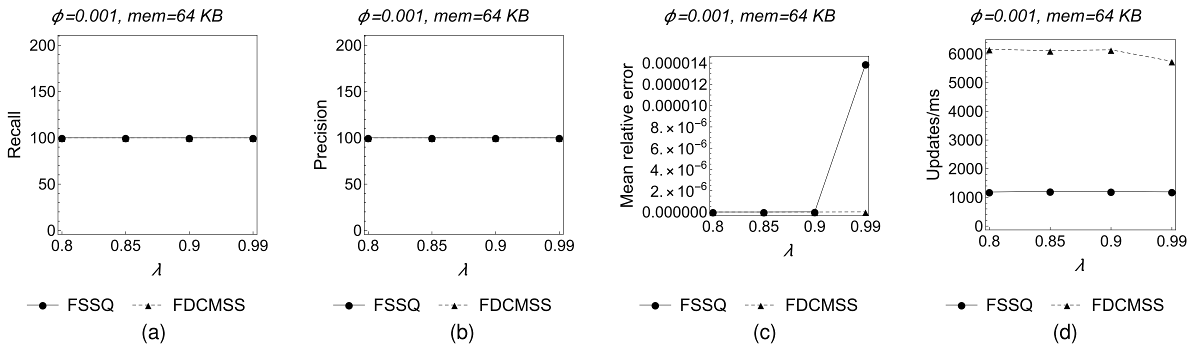

FIGURE 8. Kosarak results varying λ decaying factor. (a) Recall. (b) Precision. (c) Average Relative Error. (d) Updates/ms.

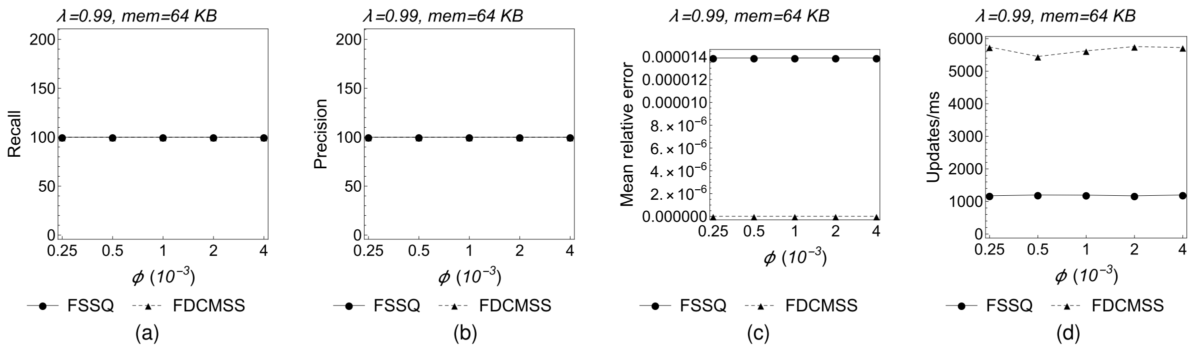

FIGURE 9. Kosarak results varying ϕ support threshold. (a) Recall. (b) Precision. (c) Average Relative Error. (d) Updates/ms.

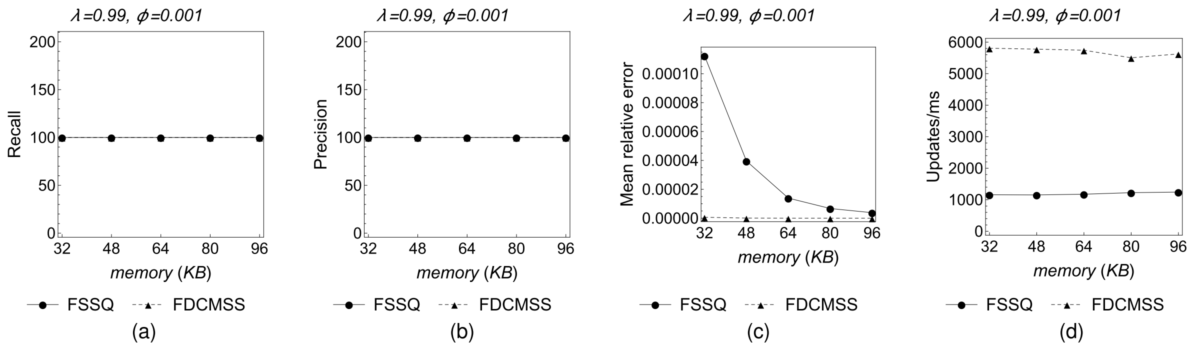

FIGURE 10. Kosarak results varying the budgeted memory. (a) Recall. (b) Precision. (c) Average Relative Error. (d) Updates/ms.

Processing Time Sum (attribute number 148), coming from the "clicks" dataset. A pre-processing step was required, in order to obtain the final dataset. We had to replace all of the missing values (appearing as question marks) with the value of 0.

4) NASA

Compliments of NASA and the Voyager 2 Triaxial Fluxgate Magnetometer principal investigator, Dr. Norman F. Ness, this dataset contains several data. We selected the Field Magnitude (F1) and Field Modulus (F2) attributes from the Voyager 2 spacecraft Hourly Average Interplanetary Magnetic Field Data. A pre-processing step was required for this dataset: having selected the data for the years 1977-2004, we removed the unknown values (marked as 999), and multiplied all of the values by 1000 to convert them to integers (since the original values were real numbers with precision of 3 decimal points). The values of the two attributes were finally concatenated. In our experiments, we read all of the values of the attribute F1, followed by all of the values of the attribute F2.

TABLE 4. Real datasets: experiments carried out.

Parameter	Values	Default
ϕ	0.00025, 0.0005, 0.001, 0.002	0.001
λ	0.8, 0.85, 0.9, 0.99	0.99
space (KB)	32, 48, 64, 80, 96	64

Table 4 reports the parameters value and their default in the experiments carried out on the real datasets.

Regarding Kosarak, as shown by Figures 8, 9 and 10, both FSSQ and FDCMSS reach 100% precision and recall. However, FDCMSS outperforms FSSQ with regard to both

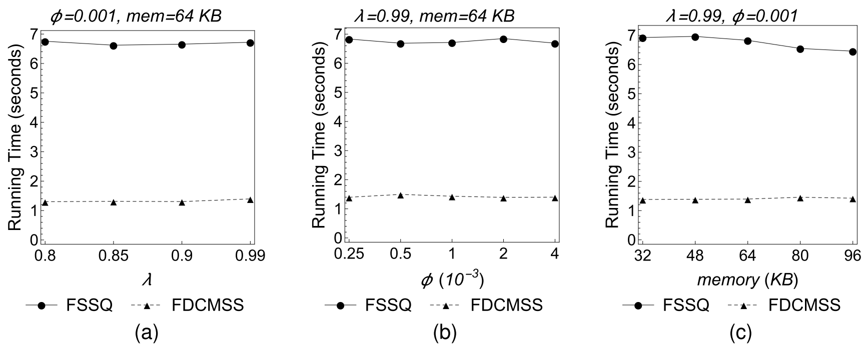

FIGURE 11. Kosarak running time varying several parameters. (a) λ. (b) ϕ. (c) Budgeted space.

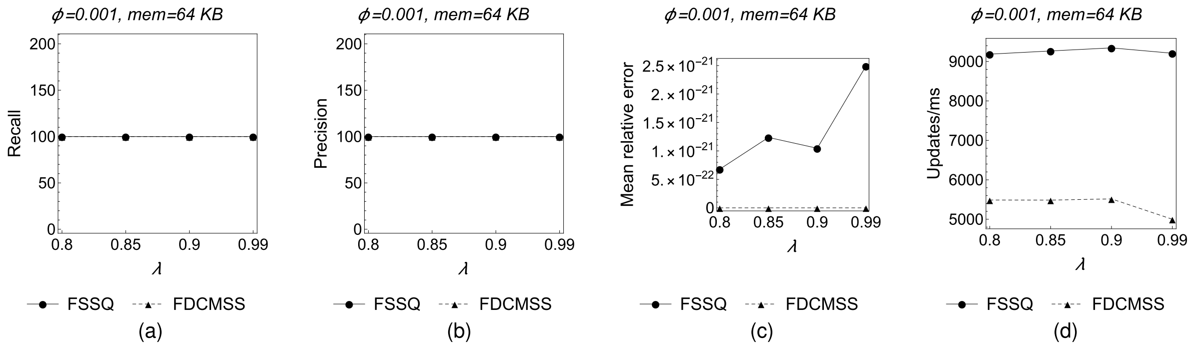

FIGURE 12. Nasa results varying λ decaying factor. (a) Recall. (b) Precision. (c) Average Relative Error. (d) Updates/ms.

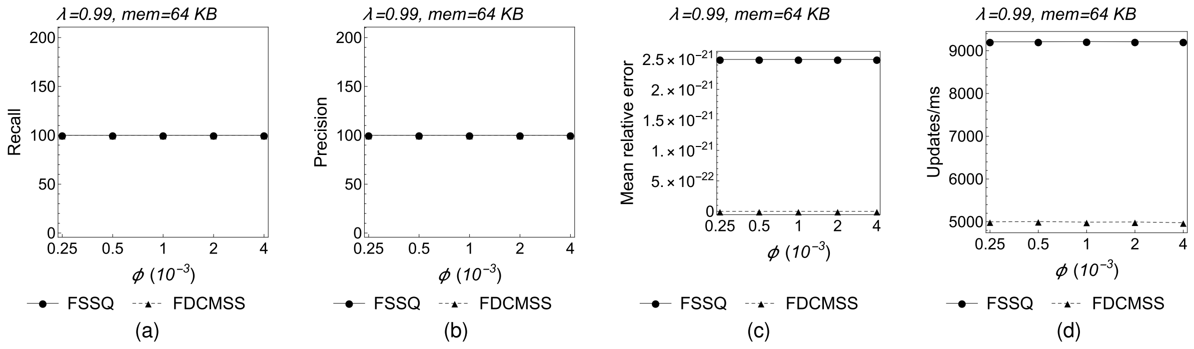

FIGURE 13. Nasa results varying ϕ support threshold. (a) Recall. (b) Precision. (c) Average Relative Error. (d) Updates/ms.

ARE and speed in all of the experiments carried out. Figure 11 depicts the running time varying λ, ϕ and the budgeted memory.

For the Nasa dataset, as shown by Figures 12, 13 and 14, again both FSSQ and FDCMSS reach 100% precision and recall. FDCMSS outperforms FSSQ with regard to ARE, whilst FSSQ outperforms FDCMSS with regard to speed in all of the experiments carried out. Figure 15 depicts the running time varying λ, ϕ and the budgeted memory. As already pointed out in Section V-A, this behaviour is strictly related to the number of times that the *delayedSorting* is invoked over the total number of updates. In particular, Table 5 reports the percentage of *delayedSorting* invocations with regard to the total number of updates for all of the real datasets under test.

TABLE 5. Real datasets: number of delayedSorting operations over the total number of item's updates. Default values have been used for those parameters which are not reported in the table.

Real Dataset	Ratio	Updates/ms
Kosarak	30.67%	1249
Nasa	0.03%	7880
Q148	16.25%	2811
Retail	93.44%	1053

We now analyze the results obtained for the q148 dataset, depicted by Figures 16, 17 and 18. FSSQ always provide 100% recall, whilst FDCMSS reaches 100% recall using slightly more space than the default 64 KB as shown in Figure 18a; its recall is 100% or slightly below 100% in all of

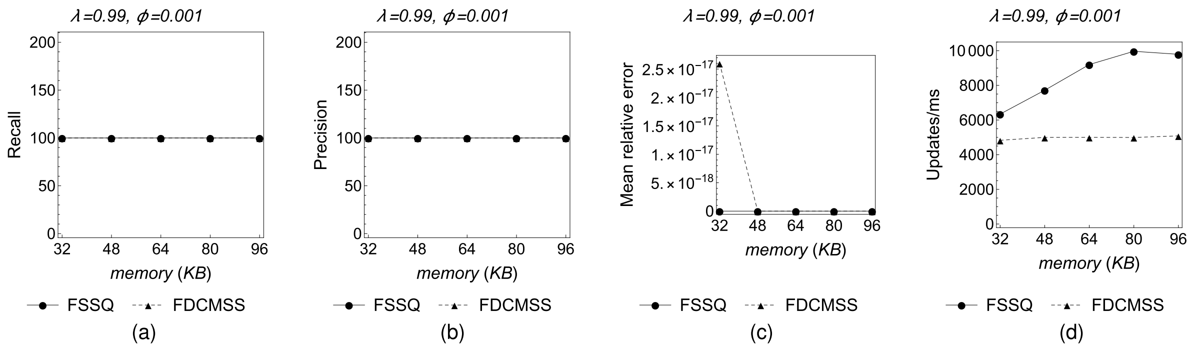

FIGURE 14. Nasa results varying the budgeted memory. (a) Recall. (b) Precision. (c) Average Relative Error. (d) Updates/ms.

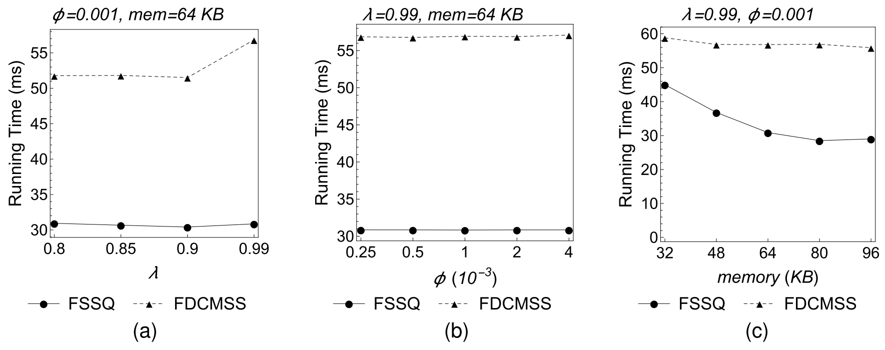

FIGURE 15. Nasa running time varying several parameters. (a) λ. (b) ϕ. (c) Budgeted space.

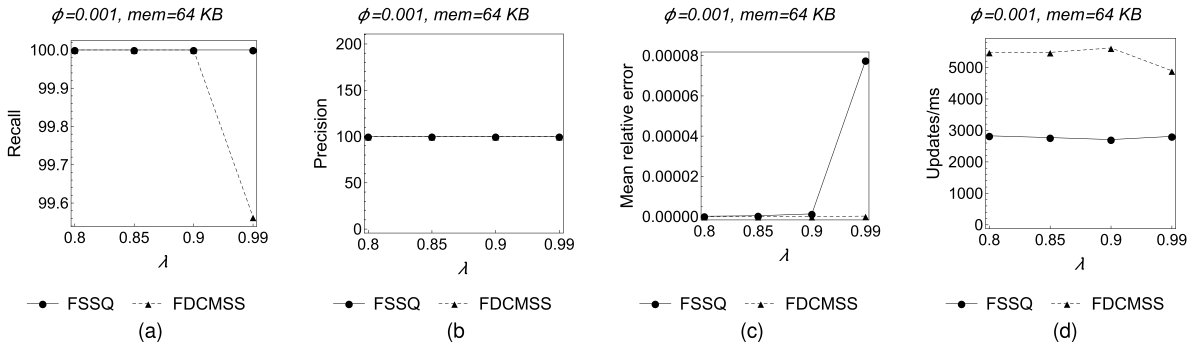

FIGURE 16. q148 results varying λ decaying factor. (a) Recall. (b) Precision. (c) Average Relative Error. (d) Updates/ms.

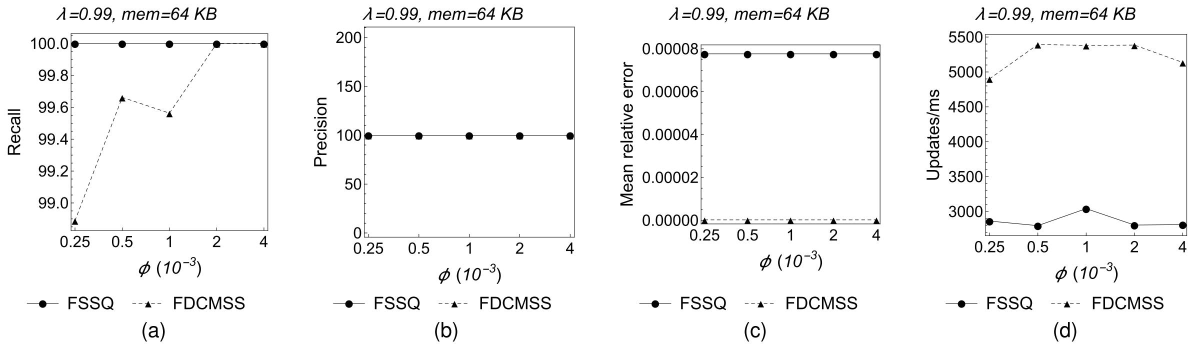

FIGURE 17. q148 results varying ϕ support threshold. (a) Recall. (b) Precision. (c) Average Relative Error. (d) Updates/ms.

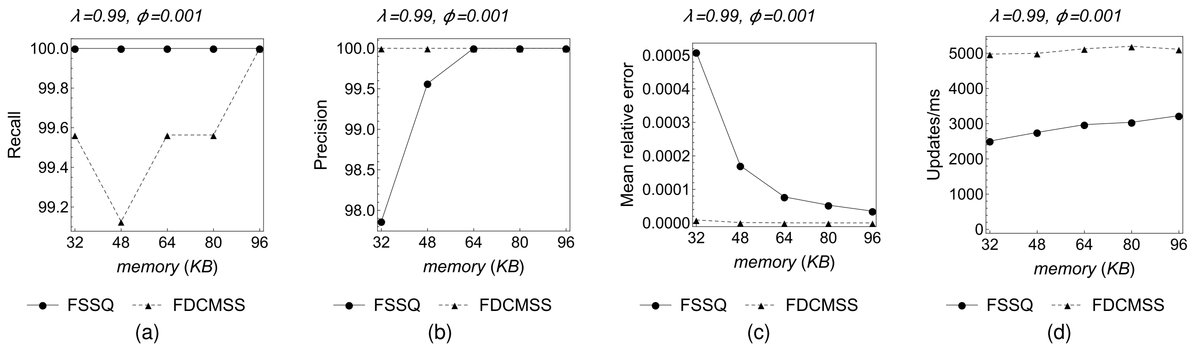

FIGURE 18. q148 results varying the budgeted memory. (a) Recall. (b) Precision. (c) Average Relative Error. (d) Updates/ms.

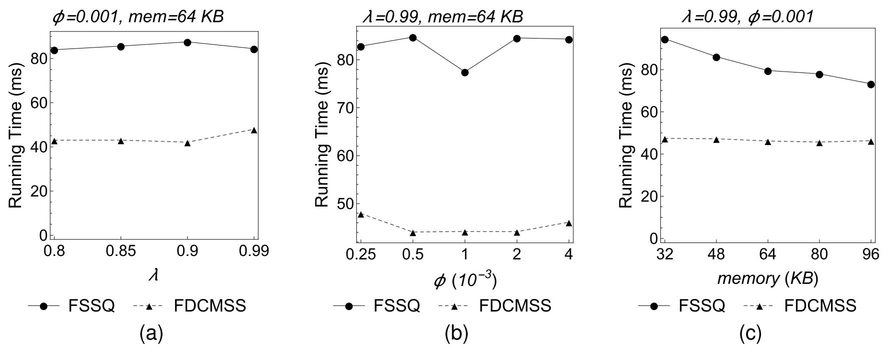

FIGURE 19. q148 running time varying several parameters. (a) λ. (b) ϕ. (c) Budgeted space.

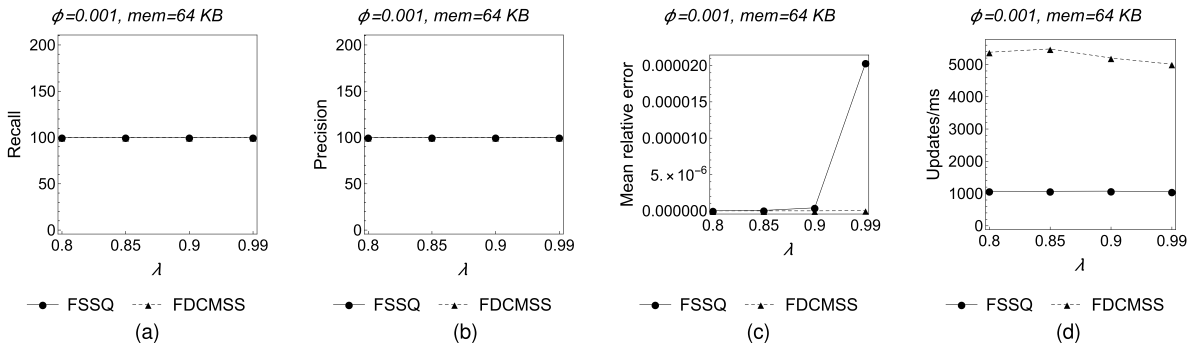

FIGURE 20. Retail results varying λ decaying factor. (a) Recall. (b) Precision. (c) Average Relative Error. (d) Updates/ms.

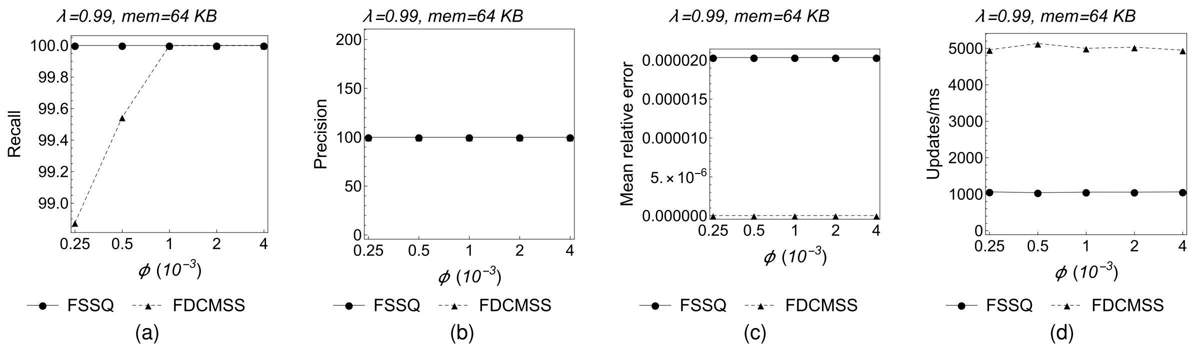

FIGURE 21. Retail results varying ϕ support threshold. (a) Recall. (b) Precision. (c) Average Relative Error. (d) Updates/ms.

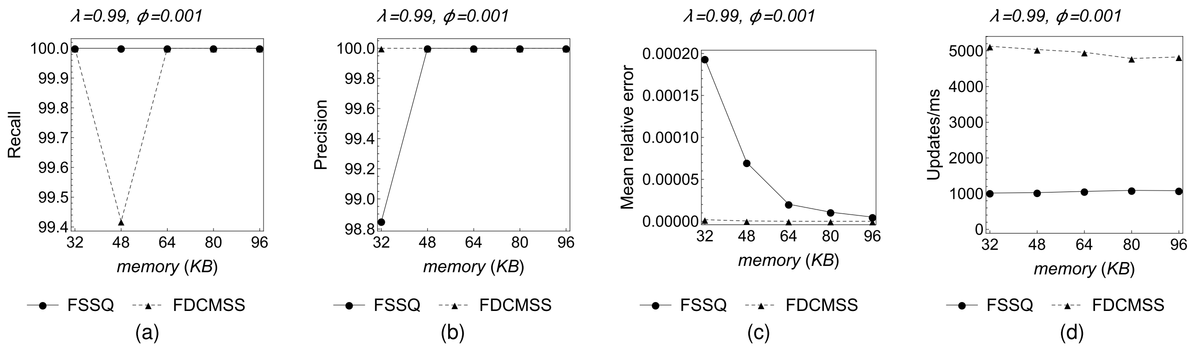

FIGURE 22. Retail results varying the budgeted memory. (a) Recall. (b) Precision. (c) Average Relative Error. (d) Updates/ms.

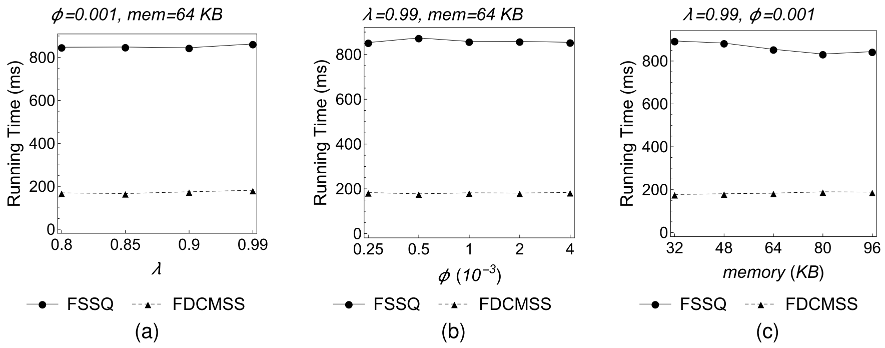

FIGURE 23. Retail running time varying several parameters. (a) λ. (b) ϕ. (c) Budgeted space.

the remaining cases. Indeed, since using the default values of λ, ϕ and budgeted memory FDCMSS does not provide 100% recall, it is easy to see why it does not reaches 100% recall in Figure 16a for $\lambda = 0.99$: when λ decreases there are less frequent items, and the space allowed is enough to track them. The same reasoning can be applied to the ϕ parameter, as shown in Figure 17a: when ϕ decreases, there are more frequent items and the budgeted memory is *a fortiori* not enough to track them. Regarding the precision, FDCMSS always reaches 100%, whilst FSSQ is below 100% in two cases, when the budgeted memory is below the default value. Regarding both ARE and speed, FDCMSS outperforms FSSQ in all of the cases. Figure 19 depicts the running time varying λ, ϕ and the budgeted memory.

For the retail dataset, FSSQ provides 100% recall and precision in all cases except that the precision is below 100% when the budgeted memory is 32 KB; for FDCMSS, using a default of 64 KB of space is just enough to report all of the frequent items. Therefore, FDCMSS in some cases does not reach 100% of recall when the number of frequent items increases, for instance when ϕ decreases. Nonetheless, FDCMSS in all of the cases loses at most one frequent item. Regarding the precision, FDCMSS always reaches 100%. As shown by Figures 20, 21 and 22, FDCMSS outperforms FSSQ with regard to both ARE and speed in all of the cases. Figure 23 depicts the running time varying λ, ϕ and the budgeted memory.

VI. CONCLUSIONS

In this paper, we have compared and contrasted two recently proposed and independently published sketch-based algorithms for mining frequent items in time-decayed streams. The FSSQ algorithm, besides a sketch, also uses an additional data structure called Quasi-Heap to maintain frequent items. FDCMSS, our algorithm, cleverly combines key ideas borrowed from forward decay, the Count-Min sketch and the Space Saving algorithm. The aim was to fully understand the strengths and weaknesses of both algorithms, with regard to frequency estimation, detection of frequent items and speed.

On the basis of the experimental results, we can infer the following conclusions. FSSQ is better suitable to the detection of frequent items than to frequency estimation. The algorithm exploits the space available very well for this purpose. However, the use of the Quasi-Heap data structure appears to be more a disadvantage than an advantage with regard to the speed of the algorithm. Indeed, the number of invocations of the *delayedSorting* function heavily influences the speed. In particular, FSSQ may not be able to cope with high-speed data streams.

FDCMSS is better suitable for frequency estimation; moreover, it is extremely fast and can be used in the context of high-speed data streams. Even though FDCMSS does not exploit the space available as well as FSSQ, nevertheless, it can be used anyway for the detection of frequent items in the time fading model, since its recall is always greater than

99%, even when using an extremely tiny amount of space. Moreover, by using slightly more space (a few additional tens of KB are enough), FDCMSS always reaches 100% recall and precision as FSSQ does, but is much faster in processing time faded streams and much less sensitive than FSSQ with regard to the input distribution (for both synthetic and real datasets). Therefore, FDCMSS proves to be an overall good choice when considering jointly the recall, precision, average relative error and the speed.

REFERENCES

[1] S. Muthukrishnan, "Data streams: Algorithms and applications," *Found. Trends Theor. Comput. Sci.*, vol. 1, no. 2, pp. 117–236, 2005. [Online]. Available: http://dx.doi.org/10.1561/0400000002

[2] S. Brin, R. Motwani, J. D. Ullman, and S. Tsur, "Dynamic itemset counting and implication rules for market basket data," in *Proc. Proc. ACM SIGMOD Int. Conf. Manage. Data (SIGMOD)*, 1997, pp. 255–264.

[3] P. B. Gibbons and Y. Matias, "Synopsis data structures for massive data sets," *External Memory Algorithms*, vol. 50, pp. 39–70, Jan. 1999.

[4] K. Beyer and R. Ramakrishnan, "Bottom-up computation of sparse and iceberg cubes," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 1999, pp. 359–370.

[5] M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani, and J. D. Ullman, "Computing iceberg queries efficiently," in *Proc. 24th Int. Conf. Very Large Data Bases (VLDB)*, 1998, pp. 299–310.

[6] M. Charikar, K. Chen, and M. Farach-Colton, "Finding frequent items in data streams," in *Proc. 29th Int. Colloquium Autom., Lang. Program. (ICALP)*, 2002, pp. 693–703.

[7] A. Gelbukhl, Ed., "Computational linguistics and intelligent text processing," in *Proc. 7th Int. Conf.*, Feb. 2006, pp. 1–2.

[8] E. D. Demaine, A. López-Ortiz, and J. I. Munro, "Frequency estimation of Internet packet streams with limited space," in *Proc. ESA*, 2002, pp. 348–360.

[9] C. Estan and G. Varghese, "New directions in traffic measurement and accounting," in *Proc. 1st ACM SIGCOMM Workshop Internet Meas. (IMW)*, 2001, pp. 75–80.

[10] R. Pan, L. Breslau, B. Prabhakar, and S. Shenker, "Approximate fairness through differential dropping," *SIGCOMM Comput. Commun. Rev.*, vol. 33, no. 2, pp. 23–39, 2003.

[11] J. Misra and D. Gries, "Finding repeated elements," *Sci. Comput. Program.*, vol. 2, no. 2, pp. 143–152, 1982.

[12] R. M. Karp, S. Shenker, and C. H. Papadimitriou, "A simple algorithm for finding frequent elements in streams and bags," *ACM Trans. Database Syst.*, vol. 28, no. 1, pp. 51–55, 2003.

[13] G. S. Manku and R. Motwani, "Approximate frequency counts over data streams," in *Proc. VLDB*, 2002, pp. 346–357.

[14] A. Metwally, D. Agrawal, and A. E. Abbadi, "An integrated efficient solution for computing frequent and top-k elements in data streams," *ACM Trans. Database Syst.*, vol. 31, no. 3, pp. 1095–1133, Sep. 2006. [Online]. Available: http://doi.acm.org/10.1145/1166074.1166084

[15] G. Cormode and S. Muthukrishnan, "An improved data stream summary: The count-min sketch and its applications," *J. Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.

[16] G. Cormode and S. Muthukrishnan, "What's hot and what's not: Tracking most frequent items dynamically," *ACM Trans. Database Syst.*, vol. 30, no. 1, pp. 249–278, Mar. 2005. [Online]. Available: http://doi.acm.org/10.1145/1061318.1061325

[17] C. Jin, W. Qian, C. Sha, J. X. Yu, and A. Zhou, "Dynamically maintaining frequent items over a data stream," in *Proc. CIKM*, 2003, pp. 287–294.

[18] M. Cafaro and P. Tempesta, "Finding frequent items in parallel," *Currency Comput., Pract. Exper.*, vol. 23, no. 15, pp. 1774–1788, Oct. 2011. [Online]. Available: http://dx.doi.org/10.1002/cpe.1761

[19] M. Cafaro and M. Pulimeno, "Merging frequent summaries," in *Proc. 17th Italian Conf. Theor. Comput. Sci. (ICTCS)*, 2016, pp. 280–285.

[20] M. Cafaro, M. Pulimeno, and P. Tempesta, "A parallel space saving algorithm for frequent items and the Hurwitz zeta distribution," *Inf. Sci.*, vol. 329, pp. 1–19, Feb. 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S002002551500657X

[21] Y. Zhang, Y. Sun, J. Zhang, J. Xu, and Y. Wu, "An efficient framework for parallel and continuous frequent item monitoring," *Concurrency Comput., Pract. Exper.*, vol. 26, no. 18, pp. 2856–2879, 2014. [Online]. Available: http://dx.doi.org/10.1002/cpe.3182

[22] Y. Zhang, "Parallelizing the weighted lossy counting algorithm in high-speed network monitoring," in *Proc. 2nd Int. Conf. Instrum., Meas., Comput., Commun. Control (IMCCC)*, 2012, pp. 757–761.

[23] M. Cafaro, M. Pulimeno, I. Epicoco, and G. Aloisio, "Parallel space saving on multi- and many-core processors," *Concurrency Comput., Pract. Exper.*, p. e4160, 2017. [Online]. Available: http://dx.doi.org/10.1002/cpe.4160

[24] S. Das, S. Antony, D. Agrawal, and A. El Abbadi, "Thread cooperation in multicore architectures for frequency counting over multiple data streams," *Proc. VLDB Endowment*, vol. 2, no. 1, pp. 217–228, Aug. 2009. [Online]. Available: http://dx.doi.org/10.14778/1687627.1687653

[25] P. Roy, J. Teubner, and G. Alonso, "Efficient frequent item counting in multi-core hardware," in *Proc. 18th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2012, pp. 1451–1459. [Online]. Available: http://doi.acm.org/10.1145/2339530.2339757

[26] K. Tangwongsan, S. Tirthapura, and K.-L. Wu, "Parallel streaming frequency-based aggregates," in *Proc. 26th ACM Symp. Parallelism Algorithms Archit.*, 2014, pp. 236–245. [Online]. Available: http://doi.acm.org/10.1145/2612669.2612695

[27] U. Erra and B. Frola, "Frequent items mining acceleration exploiting fast parallel sorting on the GPU," *Proc. Comput. Sci.*, vol. 9, pp. 86–95, Mar. 2012. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1877050912001317

[28] N. K. Govindaraju, N. Raghuvanshi, and D. Manocha, "Fast and approximate stream mining of quantiles and frequencies using graphics processors," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2005, pp. 611–622. [Online]. Available: http://doi.acm.org/10.1145/1066157.1066227

[29] M. Datar, A. Gionis, P. Indyk, and R. Motwani, "Maintaining stream statistics over sliding windows: (Extended abstract)," in *Proc. 13th Annu. ACM-SIAM Symp. Discrete Algorithms*, 2002, pp. 635–644.

[30] L. Chen and Q. Mei, "Mining frequent items in data stream using time fading model," *Inf. Sci.*, vol. 257, pp. 54–69, 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0020025513006403

[31] G. Cormode, F. Korn, and S. Tirthapura, "Exponentially decayed aggregates on data streams," in *Proc. IEEE 24th Int. Conf. Data Eng. (ICDE)*, Apr. 2008, pp. 1379–1381.

[32] A. Manjhi, V. Shkapenyuk, K. Dhamdhere, and C. Olston, "Finding (recently) frequent items in distributed data streams," in *Proc. 21st Int. Conf. Data Eng. (ICDE)*, Apr. 2005, pp. 767–778.

[33] S. Wu, H. Lin, Y. Gao, and D. Lu, "Novel structures for counting frequent items in time decayed streams," *World Wide Web*, vol. 20, no. 5, pp. 1111–1133, 2017. [Online]. Available: http://dx.doi.org/10.1007/s11280-017-0433-5

[34] M. Cafaro, M. Pulimeno, I. Epicoco, and G. Aloisio, "Mining frequent items in the time fading model," *Inf. Sci.*, vols. 370–371, pp. 221–238, Nov. 2016.

[35] G. Cormode, V. Shkapenyuk, D. Srivastava, and B. Xu, "Forward decay: A practical time decay model for streaming systems," in *Proc. IEEE 25th Int. Conf. Data Eng. (ICDE)*, Mar. 2009, pp. 138–149.

[36] N. Homem and J. P. Carvalho, "Finding top-k elements in data streams," *Inf. Sci.*, vol. 180, no. 24, pp. 4958–4974, 2010. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S002002551000397X

MASSIMO CAFARO (SM'11) received the Ph.D. degree in computer science from the University of Bari. He is currently an Associate Professor with the Department of Engineering for Innovation, University of Salento, where he is also the Director of the CINI Research Unit. He has authored over 100 refereed papers. He holds a patent on distributed database technologies. His research covers parallel and distributed computing, cloud and grid computing, data mining, and big data. He is a Senior Member of the IEEE Computer Society and the ACM, the vice chair of regional centers and a Coordinator of the Technical Area on Data Intensive Computing for the IEEE Technical Committee on Scalable Computing. He serves as an Associate Editor of the IEEE Access.

ITALO EPICOCO received the Ph.D. degree in computational engineering from the University of Lecce, Italy. He is currently an Assistant Professor with the University of Salento, Lecce, Italy. He is also an Affiliate Researcher with the Euro-Mediterranean Center on Climate Change (CMCC). He has authored over 40 papers in refereed books, journal, and conference proceedings. His research interests include high performance, distributed, grid, and cloud computing, with particular emphasis on parallel data mining. During his past research activities, he addressed issues related to the optimization of numerical methods for solving PDEs applied to Earth system models and to fluid dynamics models on high-end parallel architectures, including heterogeneous architectures made of accelerators (NVIDIA GPU and Intel MIC). Relevant activities also included optimized management of a huge amount of data produced by the climate models.

MARCO PULIMENO received the Laurea (M.Sc.) degree in computer engineering and the Ph.D. degree in mathematics and computer science from the University of Salento, Italy. He has authored on the topic of frequent items in several refereed journals and conference proceedings. His research interests include high performance computing, distributed computing, and in particular, parallel data mining.

GIOVANNI ALOISIO is currently a Full Professor of information processing systems with the Department of Engineering for Innovation, University of Salento, Lecce, Italy, where he is leading the HPC Laboratory. He is a member of the Governance bodies, the Director of the Supercomputing Center, and a member of the Strategic Council and the Executive Committee, Euro-Mediterranean Center on Climate Change (CMCC). His expertise concerns high performance computing, grid and cloud computing, and distributed data management. He has been a Co-Founder of the European Grid Forum, which then merged into the Global Grid Forum, now Open Grid Forum. He was one of the key experts of the International Exascale Software Project, whose main goal was the definition of the software roadmap for scientific computing at exascale. He was the responsible for European Network for Earth System Modeling within the European Exascale Software Initiative. He chairs the WCES, European working group of the EESI2 Project. He is also the responsible for CMCC of the EUBrazil CC and Ofidia EU-FP7 projects. He is a member of the ENES HPC Task Force. He has authored over 100 papers in referred journals on parallel, grid computing, distributed data management, and exascale computing.