SHE: A Generic Framework for Data Stream Mining over Sliding Windows (Technical Report, for ICPP '22 review only)

Anonymous Author(s)

ABSTRACT

Data stream mining over a sliding window is a fundamental problem in many applications, such as financial data trackers, intrusion detection and QoS. To meet the demand for high throughput of high speed data streams, hardware platforms (FPGA/ASIC) have been designed. These hardware platforms have three constraints for algorithms running on, which are 1) small memory usage 2) single stage memory access and 3) limited concurrent memory access. Algorithms perfectly fit in with these constraints will enable a highest utilization of these hardware platforms. However, no existing sliding window algorithm is specifically designed for hardware platforms. In this paper, we propose the Sliding Hardware Estimator (SHE), which is a generic framework that extends existing fixed window algorithms to sliding windows for hardware platforms. The key idea of SHE is that, during insertions we approximately delete out-dated information with little time and space overhead, while during queries we design sophisticated techniques to minimize error. We have fully implemented our SHE on FPGA, achieving a throughput of 544 Mips. We apply SHE to four typical data stream mining tasks. Experimental results show that, when compared with the state-of-the-art which cannot be implemented in hardware, SHE reduces the error by up to 100 times in membership queries. All related source codes are anonymously released at Github.

1 INTRODUCTION

1.1 Background and Motivations

Data stream measurement over sliding windows provides fundamental information for data management and analyses [1]. It collects information of recent data arrivals (*e.g.*, data updates over the last second or the most recent 1,000,000 items) while expiring obsolete data, thus can more accurately describe the current status of the data stream than fixed window measurements. Fundamental measurement tasks include cardinality[1] [2, 3], membership [4, 5], frequency [6], similarity [7], *etc.* These tasks play important roles in various real world applications of sliding window measurements, such as in financial data trackers [8, 9], intrusion or anomaly detector [10, 11], improving Quality of Service (QoS) [1], *etc.*

At present, as the amount of data grows tremendously, high-speed data transmission has become a new challenge. For example, the bandwidth of the servers in Facebook's datacenter achieves at least 10Gbps [12]. In order to meet the demand for high-speed data streams, a variety of dedicated hardware platforms have been designed, including FPGA [13] and high-speed commodity ASIC (Application Specific Integrated Circuit). For example, ASIC is a microchip designed for a special application, such as a specific transmission protocol or a hand-held computer. In order to achieve high performance for specific tasks, algorithms are required to meet the following constraints of these dedicated hardware platforms: **1) small memory usage which can be implemented in SRAM** (*e.g.*, a Virtex FPGA has less than 30MB of memory available on-chip [13]); **2) single stage memory access** (two stages are not allowed to visit the same memory region simultaneously [14]); **3) limited concurrent memory access** (Each stage can access one memory address with limited size [14–16]). Because algorithms violating any of the three constrains will either degrade performance or be incompatible to hardware platforms, the *design goal* of this paper is to propose a generic algorithm over sliding window measurement while supporting the three constraints.

[1]Cardinality refers to the number of different items in a data stream

1.2 Our Solution

In this paper, we propose Sliding Hardware Estimator (SHE). It is a generic framework which can adapt common fixed window algorithms (*e.g.*, Bloom filter [17], Bitmap [18], HyperLogLog [19], Count-Min Sketch [20], MinHash [21]) to sliding window scenarios. SHE not only meets the above three constraints of special hardware platforms, but also can achieve higher measurement accuracy compared with the state-of-the-art works.

In aspect of memory constraint, we propose the key idea of approximate cleaning. For sliding window measurements, the key challenge lies in handling out-dated information. One typical strategy is to accurately clean out-dated cells (*i.e.*counters/bits) by using additional data structures for precise timestamp records (usually 64-bits). However, the large memory overhead hinders implementation on dedicated hardware platforms. We, on the other hand, use approximate cleaning, which abandons timestamps and allows tolerable error in kicking out out-dated information. Specifically, we use an additional process to circularly clean the cells in the data structure, and thus information in any cell is sure to be cleaned within a cycle. The circular cleaning method also gives each cell an age, denoting the age of information stored in this cell. We set the cleaning cycle larger than the size of sliding window, so that there will be both younger cells (age smaller than a window) and aged cells (age larger than a window). With the help of this, we can pick cells with proper age for specific measurement tasks. For example, for data structures like Bloom filter [17] which only tolerates false positives, we pick aged cells. Contrarily, for data structures like Bitmap [18], we pick both younger and aged cells with age at about the size of the window, so as to avoid too biased information. Although approximate cleaning introduces some error in kicking out out-dated information, it can achieve higher accuracy than the first strategy of accurate cleaning. The reason is that memory of timestamps can be saved, thus increasing the number of cells and lowering hash collision. For example, for a Bloom filter with 8 hash functions, if we save the memory of all 64-bit timestamps attached to each bit, we can enlarge the size of the Bloom filter by 64 times. In fact, the error of membership query can decrease by 2.8×10^{15} times.

In aspect of single stage memory access, we choose to extend five fixed window algorithms (*i.e.* Bloom filter [17], Bitmap [18], HyperLogLog [19], Count-Min Sketch [20], MinHash [21]). The reason is that, in these fixed window algorithms, the processing of certain memory region can be finished in one stage. Our SHE framework preserved this fine characteristic and can be implemented on dedicated hardware platforms such as FPGA.

In aspect of limited concurrent memory access, we implement two techniques in SHE: group cleaning and on-demand cleaning. Group cleaning means updating several continuous cells (*e.g.*, 128 bits) in the data structure, rather than updating 1 bit at a time. Because hardware devices fetch a continuous memory fragment at a time (*e.g.*, FPGA reads 1024 bits at one memory access), the cost of updating a single bit is about the same for updating a group. Therefore, group cleaning technique can profoundly reduce the frequency of memory accesses, thus accelerating the hardware procession. However, both circular bit cleaning and group cleaning require another process to perform circular memory access. Therefore, we propose on-demand cleaning: we clean a group only when a new item is mapped into this group and this group needs to be cleaned. Otherwise, if a group does not undergo any insertion after its original cleaning time, it remains unchanged.

Main Experimental Results: We implemented our SHE framework on FPGA, a representative of dedicated hardware platforms. Results of SHE-Bloom filter and SHE-Bitmap show that it requires only limited resources while achieving a throughput of 544 Mips (million items per second). Additionally, SHE framework also achieves higher accuracy than the state-of-the-art: 1) For membership task, our algorithm can achieve about 100 times smaller false positive error rate compared to SWAMP. 2) For cardinality task, to achieve the same relative error of 1%, our algorithm requires only 1% of memory used by SWAMP. 3) For frequency task, our algorithm is 10 times more accurate than its competitors when memory resources are scarce. 4) For similarity task, our algorithm is 10 times more accurate than a straw-man solution.

2 RELATED WORK

2.1 Fundamental Measurement Tasks and Algorithms

Here we demonstrate the four basic measurement tasks of the data stream (*i.e.*, membership query, frequency, cardinality and similarity estimation) and their representative algorithms. These algorithms are widely used when dealing with data steam measurement tasks, because these tasks value more on processing speed and can accept relatively low errors in query. These representative algorithms do not give an accurate report of data stream queries, but gives an estimated result instead.

Membership query asks whether an item is a member of the past data stream. Bloom filter (BF) [17] is a representative algorithm for membership query. It is an $n-bit$ array. All n bits are initially set as 0. In the insertion process, for each coming item x, it computes k hash values of x, which are $h_1(x), h_2(x), \ldots, h_k(x)$. It sets the bits in these k locations to 1, no matter whether they were 0. Figure 1 gives an example of insertion with $k = 4$. As for query, it refers to the same k locations. The BF reports true if all k bits are 1 and reports false if there is at least one bit being 0.

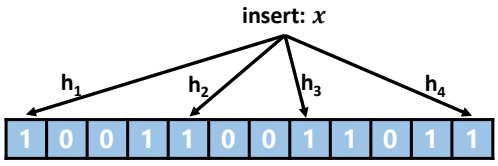


Figure 1: Insertion of Bloom filter with $k = 4$ hash functions.

Cardinality estimation asks the number of different items in the past data stream. Bitmap and HyperLogLog are two representative algorithms for cardinality estimation. **Bitmap** [18] is a n-bit vector. All bits are initially set to 0. To insert an item x, it computes a hash value $h(x)$, then sets the $h(x)^{th}$ bit to 1. As for query, it counts the number of 0s in the vector, denoted as u. The bitmap gives the maximum likelihood estimation of data stream cardinality as: $-n \ln \frac{u}{n}$. **HyperLogLog** [19] is an m-counter array. The counters are denoted as $C[0], C[1], \ldots, C[m-1]$. There are two hash functions, H_c and H_z. To insert an item x, we first use $H_c(x)\%m$ to select a counter, denoted as $C[i]$, and count the number of the leading 0 bits in $H_z(x)$, which is denoted as ℓ_{zero}. When the insertion is over, we compute the largest ℓ_{zero} value for each counter $C[i]$ and denote it as ℓ_i As for query, HyperLogLog gives the estimation of cardinality as $\widehat{C} = ck(\sum_{j=1}^{k} 2^{-\ell_j})^{-1}m$, where c is a constant.

Frequency estimation asks the number of items with the same item ID in the past data stream. Count-Min Sketch (CM Sketch) [6] is a representative algorithm for frequency estimation. It is an n-counter array. All counters are initially set to 0. Just like the BF, for each incoming item x, it computes k hash values of x, which are $h_1(x), h_2(x), \ldots, h_k(x)$. It adds 1 to these k counters. As for query, it refers to the same k locations and reports the smallest value among all k counters as the estimation of the frequency of item x.

Similarity estimation asks the similarity of items between two data streams, regardless of their time order. We use Jaccard index to measure similarity. It is defined as $J(A, B) = \frac{|A \cap B|}{|A \cup B|}$, where A and B are two multisets. The Jaccard index equals to 0 when two sets are disjoint and equals to 1 when they are identical. MinHash [21] is a representative algorithm for similarity estimation. It uses n hash functions. For each hash function, MinHash checks whether the minimal hash values of all items in the two sets are equal. If there are m minimal hash values that are equal, the Jaccard similarity of the two sets is estimated at $\frac{m}{n}$.

2.2 Prior Work for Measurements over Sliding Windows

We divide the probabilistic statistical algorithms for sliding window measurements into two categories: 1) specialized algorithms for single task 2) generic algorithms supporting multiple tasks.

1) Specialized algorithms:

The Sliding HyperLogLog (SHLL) [22] is used for cardinality estimation based on HyperLogLog. For each counter, it adopts a monotone priority queue to maintain the possible extreme values within the time range. The query step of SHLL is exactly the same as HyperLogLog. The advantage is that the queues can perfectly delete out-dated information. In certain cases, however, the queues may be undesirably long, thus breaking memory limits.

The Counter Vector Sketch (CVS) [23] is used for cardinality estimation based on Bitmap. It is an array of counters with maximal value of c and minimal value of 0. When inserting an item, it updates all hashed counters to c. After that, it randomly choose several counters and decrease them by 1. For query, CVS counts the number of non-zero counters and estimate the cardinality by maximum likelihood estimation same as Bitmap 2.1. CVS falls short in the error induced by the randomness in picking counters to decrease.
The Timestamp-Vector algorithm (TSV) [24], is used for cardinality estimation based on Bitmap. It uses an array of timestamps. For insertion, it sets the hashed counters to the arriving time of the item. For query, it counts the number of active timestamps (*i.e.*, timestamps within the latest sliding window) in the array and estimate the cardinality by maximum likelihood estimation same as Bitmap 2.1.
The Time-Out Bloom Filter (TOBF) [25] is used for membership estimation based on Bloom filter. It uses an array of timestamps. For insertion, it sets the hashed counters to the arriving time of the item. For query, if there are any out-dated counters among several hash positions, it reports the queried item does not appear in the latest sliding window. Otherwise, it returns true.
The Timing Bloom filter (TBF) [26] is also used for membership estimation based on Bloom filter. TBF is similar to TOBF but uses a wraparound counter array to record arrival time instead of recording timestamps directly. Every time an item is inserted, TBF scans a piece of the array to remove the out-dated time records. The disadvantage of the above three algorithms based on timestamp array (TSV, TOBF, and TBF) is memory inefficiency. The timestamp is large and could be stored for many times.
2) Generic algorithms:
SWAMP [27] is currently the best generic algorithm for sliding window measurements. In SWAMP, there is a cyclic queue, whose size w equals to the size of the sliding window, to record the fingerprints of the latest w items. In addition, there is a Tiny Table [28] used to record the frequency of distinct items in the latest w items. Upon an item arriving, the oldest fingerprint in the queue is replaced by the fingerprint of the item, and the frequencies of the oldest fingerprint and the newly arrived fingerprint are updated in the Tiny Table. Using SWAMP, we can easily get the statistics of cardinality, membership and frequency in a sliding window. In practice, SWAMP is versatile and accurate when the memory is sufficient. In order to maintain the diversity of algorithm functions, SWAMP is not memory efficient enough because the space complexity of SWAMP is O(W) where W is the number of items in a sliding window. Besides, SWAMP can not be implemented on hardware devices like P4 switches or FPGA. We explain the reason in the next part. Unlike SWAMP which is an algorithm supporting multiple sliding window measurement-tasks, we propose a generic framework to enhance the existing algorithms designed for fixed windows so that they can be applied to measurement tasks over sliding windows. In that case, the algorithms enhanced by our framework can play their own expertise in different tasks. Therefore, we can achieve significantly better performance in each task.

2.3 Constraints for Hardware Implementations

Although the circuits on some advanced programmable ASICs can be designed as complex as the microprocessor, the pipeline architecture is usually preferred in order to achieve high processing speed with limited hardware resources. Therefore, when processing data streams, a hardware-friendly algorithm is supposed to be implemented as a series of pipeline stages with the following constraints:

(1) **Limited size of SRAM memory.** Memory access is usually the limitation of the processing speed on most hardware platforms. SRAM provides faster memory access but it is more expensive and therefore the memory size of SRAM is typically small.
(2) **Single stage memory access.** A read-write hazard may be caused when two stages access the same memory region simultaneously. Therefore, each memory region is supposed to be accessed once when an item is going through the pipeline.
(3) **Limited concurrent memory access.** Each stage can access one memory address with limited size. In the other word, it is not suitable to access the entire memory block or a large amount of memory in a single stage.

The constraints mentioned above are not met when using SWAMP to process data streams. First, the Tiny Table used by SWAMP records the fingerprints of all the arriving items, and thus the memory usage is not affordable when dealing with a large sliding window. Meanwhile, buckets, of which the Tiny Table consists, are somehow tied together. When an item is inserted into a filled bucket, the bucket will expand to its adjacent buckets, and the domino effect may occur, which leads to an unlimited concurrent memory access. Even if the domino effect is constrained, the operation of the Tiny Table is still too complex to be finished within one memory access. There are three fields in a bucket, each of which is probably modified during one single insertion. However, an extra deletion occurs when the space of the bucket is fully used and it is not allowed to expand to other buckets. Therefore, the changes of the three distinct fields may affect each other and therefore they can not be modified sequentially.

3 THE SHE FRAMEWORK

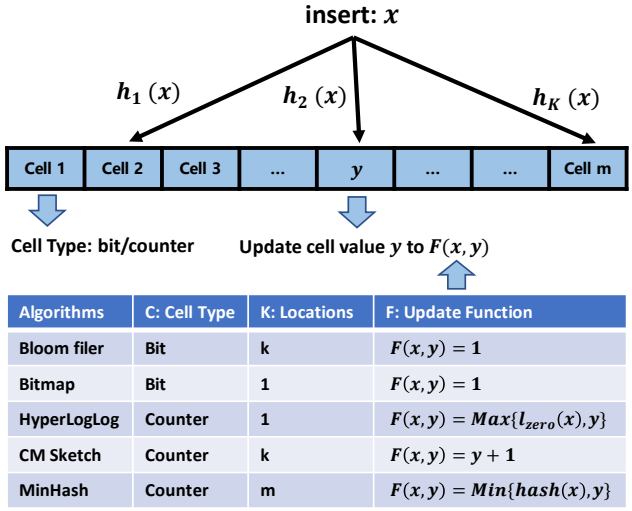
We propose our framework, Sliding Hardware Estimator (SHE) by two versions, a basic software version for CPU platform (Sec. 3.2) and the hardware version for FPGA and ASIC (Sec. 3.3). Our SHE is a framework that can be combined with different algorithms. For easier understanding, we will introduce SHE combined with the Bloom filter (SHE-BF). Important notations used in the rest of the paper are demonstrated in Table 1.

3.1 Preliminaries and Problem Statement

Sliding-window model: A data stream is an infinite sequence of items. The sliding window includes the most recent items appearing in a data stream. A sliding window with size N can be count-based (contains the last N items) or time-based (contains the items in the last N time units). The items who leave the sliding window are outdated. In this paper, we only concern the fundamental measurement tasks for the items within the sliding window.

Table 1: Notations used in the rest of the paper.

Notation	Meaning
t_{cur}	current time
N	size of a sliding window
M	number of cells in a data structure
T_{cycle}	time of a cleaning cycle
α	a constant parameter, $\alpha = \frac{T_{cycle}-N}{N}$
G	number of groups
w	number of cells in a group, $w = \frac{M}{G}$
d_{gid}	the time offset of the gid-th group, $d_{gid} = -\lfloor \frac{T_{cycle} \cdot gid}{G} \rfloor$
$m[gid]$	the time mark of the gid-th group



Algorithms	C: Cell Type	K: Locations	F: Update Function
Bloom filer	Bit	k	$F(x,y) = 1$
Bitmap	Bit	1	$F(x,y) = 1$
HyperLogLog	Counter	1	$F(x,y) = Max\{l_{zero}(x), y\}$
CM Sketch	Counter	k	$F(x,y) = y + 1$
MinHash	Counter	m	$F(x,y) = Min\{hash(x), y\}$

Figure 2: Common Sketch Model for data stream.

Common Sketch Model (CSM) for data stream (Fig. 2): We propose *Common Sketch Model (CSM)* by summarizing the five algorithms for data stream in Sec 2.1. An algorithm of CSM can be denoted by a triple $< C, K, F >$. Each algorithm of CSM has an array of m *cells*. The cell can be a bit or a counter, recorded by C. When inserting an item x, the algorithm selects K cell(s), called *hashed/mapped cell(s)*, by hash and independently update them with the update function F. The function F sets the value in hashed cell, y, to $F(x, y)$. Our SHE can enhance any algorithm of CSM to make them apply to the sliding window. For convenience, we call the algorithms before enhancing *original algorithms*.

3.2 SHE Framework on Software

In this part, we introduce SHE framework for CPU platform (SHE_{soft}). The SHE should complete two important tasks. One is cleaning out-dated items with minimal additional structure (memory). The other is extracting all useful information based on the limitation of the data structure.

1) Structure and Operations

Structure: To avoid a larger additional structure than the original algorithms. We use an additional cleaning process to delete the out-dated items periodically, which does not cause additional memory consumption.

Cleaning: There is a cleaning process that cleans the cells in the array from the left to the right one by one periodically. Specifically, the cleaning process resets a cell to zero when scanning it. The process starts from the leftmost cell in the array and ends at the right most cell. The process spends T_{cycle} time from the left to the right and it moves at a constant speed. T_{cycle} is larger than the sliding window size N. When the process gets the rightmost cell, it goes back to the leftmost cell in an instant and repeat the cleaning process.

Insert: We insert an item by the same method as the fixed-window algorithms. To insert an item x, we update all mapped cells. The insertion is independent from the cleaning.

Query: Before introducing the detailed operations, we classify all cells into three types, 1) perfect cells, 2) young cells, and 3) aged cells. When querying at the current time t_{cur}, a cell, whose latest cleaning was at exactly N time units before t_{cur}, is a perfect cell for query, because it records the items within the sliding window, no more and no less. The young cells are the cells who were cleaned later than perfect cells. Many of them are cleaned recently and furthermore they record the items in a smaller window. The aged cells are the bits (cells) who were cleaned earlier than the perfect bit and have not been cleaned again. They record more items and furthermore they record the items in a larger window.

To achieve a better accuracy, we use two techniques to select as many proper cells as possible for estimation. The first technique is called **age sensitive selecting**. When dealing with one-sided error algorithms, we only choose the perfect cells and aged cells because the young cells loose part of information within the sliding window and is potential to violate the one-sided error feature. However, when the original algorithm is two-sided error, selecting the young cells whose age is close to N lets the result less biased and therefore increases the accuracy. The second technique is called **on-demand cleaning**. Since most of young cells are ignored due to age sensitive selecting, we should guarantee the number of aged cells for measurement. To achieve this, the time of a cleaning cycle T_{cycle}, is set to larger than the size of sliding windows N. Even though the preserved out-dated information caused by on-demand cleaning occasionally leads to an estimation error, the enlarged number of cells for measurement improves the average performance.

Although we use different query strategies for different original algorithms, the common process to determine the cells not to be ignored can be concluded as follows. For a cell, we first compute its age (*i.e.*, the length of time since its latest cleaning) according to the distance from this cell to the current position of the cleaning process, and the scanning speed of the cleaning process. Then we classify the cell into the above three types by its age. Finally, we determine whether it should be included/ignored by both its age and its type.

2) SHE Works on the Bloom filter

Structure and common operations: For the Bloom filter, each bit in the bit array is a cell. We combine it with our SHE and the new algorithm is denoted by SHE-BF. When inserting, we set all hashed bits to 1. There is an additional cleaning process.

Query: When querying, all young cells (bits) should be ignored and the following operations only include the perfect cells and the aged cells. The method that we check the cells is the same as the original Bloom filter: We check whether there is any 0 bit in the included cells. If so, we answer the item is not in the sliding window. Otherwise, we answer the item is in the sliding window.

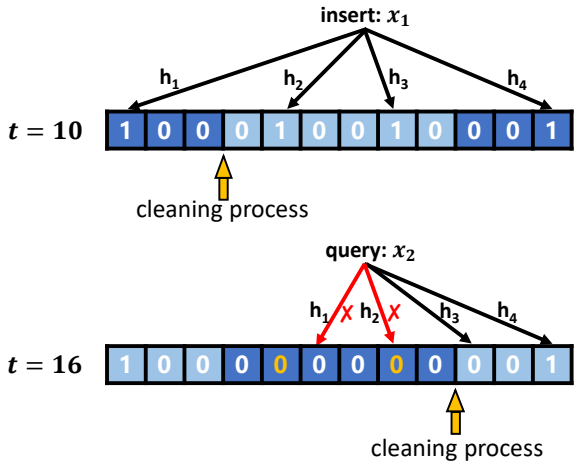


Figure 3: An example of SHE_{soft} applied on the Bloom filter.

As shown in Fig. 3, we give an example to show how the software version of SHE works on the Bloom filter. Suppose we insert an item x_1 at time $t = 10$ and we query another item x_2 at time $t = 16$. Let the sliding window size N be 6. The clean process cleans one bit (cell) per time unit (*i.e.*, $T_{cycle} = M$). Let the number of hash functions in the Bloom filter be 4. Let $h_i()$ denotes the i-th hash function. To insert x_1 at $t = 10$, we set the 4 bits (cells), $h_i(x_1)$, to 1. The clean process cleans one bit (cell) in each time unit. After 6 time units, there are 6 bits (marked by dark blue) that are cleaned in recent $N = 6$ window. These bits (cells) are young cells. To query another item x_2 at current time ($t = 16$), we ignore those young cells and we only check whether the remaining mapped cells are all 1. We answer x_2 did not appear because the cell mapped by $h_3(.)$ replies false.

3) Discussion

Error classification: There are three sources of errors, 1) hash collision, 2) error incurred by aged cells and 3) error incurred by younger cells. The hash collision is that different items were hashed to a same cell and some errors occur. The error incurred by aged is the error incurred by some items out of the sliding window. The error incurred by aged will introduce false positives or over estimation. The error incurred by young is the error incurred by the missing items, which are cleaned too early, in the sliding window. The error incurred by young will introduce false negatives or lower estimation. In the example of the Bloom filter, the hash collision is that a bit (cell) was hashed by multiple items so that each item cannot distinguish whether itself has been inserted through the bit. For Bloom filter, we ignore young cells to eliminate the false negative. The key idea of the query of a Bloom filter is to find a zero bit. The aged cells can be used, because a zero aged bit indicates that the item has not been inserted in a larger window, which can conclude that it has not been inserted in the sliding window.

Error control: There are two factors that influence the error, the speed of cleaning and the strategy of ignoring cells when querying. The speed of cleaning influences the ratio of young cells in total. The total number of young cells and aged cells is approximately a constant M. We use α to denote the ratio of $T_{cycle} - N$ to window size N, $\alpha = \frac{T_{cycle}-N}{N}$. The number of aged cells is α times of the number of young cells. When α becomes larger, the speed of cleaning becomes slower. As the result, there are less young cells. For SHE-BF, less young cells mean lower possibility of ignoring cells. Simultaneously, the slower clean will lead to more hash collisions because many out-dated items cannot be cleaned in time.

The strategy of ignoring cells depends on different original algorithms. For SHE-BF, it has no false negative because it ignored all young cells. The hash collision will not result in false negative. Because the original Bloom filter has one-sided error (*i.e.*, only reply false positive answers), our SHE preserves this property. However, ignoring all young cells is not the best choice for some original algorithms with two-sided errors. If some false negative is accepted, some young cells which are going to become perfect cells can also be referred.

3.3 SHE Framework on Hardware

As the performance of additional cleaning process is limited on hardware, we propose a new reset strategy to replace the process completely. Firstly, we divide the cell array into groups and we reset groups instead of cells. Next, we give each group a time offset to simulate the operation of scanning and cleaning. Finally, we attach our time mark to every group so that we can update each group by lazy update strategy.

Structure: Based on the software version, we divide the cell array into G groups equally, each of which has continuously $w = \frac{M}{G}$ ($w \geqslant 1$) cells. We attach one bit, called *time mark*, to every group. Let $m[gid]$ denotes the time markattached to the gid-th group. Let d_{gid} be the time offset of the gid-th group, which is evenly spaced in range $[0, T_{cycle})$, *i.e.*, $d_{gid} = -\lfloor \frac{T_{cycle} \cdot gid}{G} \rfloor$.

Algorithm 1: New operations in hardware version

Input: the index of a group gid, current time t_{cur}

Procedure `CheckGroup`(gid, t_{cur}):
  $CurMark \leftarrow \lfloor \frac{t_{cur}+d_{gid}}{T_{cycle}} \rfloor \mod 2$;
  if $m[gid] \neq CurMark$ **then**
    $m[gid] \leftarrow CurMark$;
    Reset all cells in group gid to 0.

Function `CheckMature`(gid, t_{cur}):
  CheckGroup(gid, t_{cur});
  if $(t_{cur} + d_{gid} \mod T_{cycle}) \geqslant N$ **then**
    return $True$;
  else
    return $False$;

Cleaning and insertion: As the cleaning/insertion for each cells is the same, we only describe the operation for one cell. Before inserting into a cell, we additionally check whether the group that

the cell is in need to be cleaned. The pseudo-code(1) shows how we check the group. We calculate the current time mark by $m_{cur} = \lfloor \frac{t_{cur}+d_{gid}}{T_{cycle}} \rfloor \mod 2$, where t_{cur} is the current time, d_{gid} is the time offset, and T_{cycle} is the time of a cleaning cycle for every group. T_{cycle} corresponds to the time of the cleaning cycle in the software version of SHE. For each group, the current time mark (*i.e.*,$m_{cur} = \lfloor \frac{t_{cur}+d_{gid}}{T_{cycle}} \rfloor \mod 2$) flips every T_{cycle}. Then we check whether the current time mark is equal to the last time mark $m[gid]$. if not, we record the current time mark and clean all cells in the group.

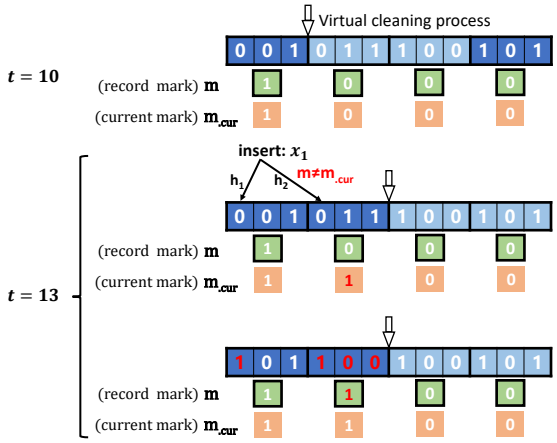


Figure 4: Cleaning and insertion of SHE in Bloom filter.

As shown in Fig. 4, we give an example to explain how the hardware version of SHE works on the Bloom filter. We show the details of the structure at time $t = 10$. At time $t = 13$, we insert an item x_1 to the structure. For the bit hashed by h_1, the current time mark is the same as the recorded mark. We just set the bit to 1. For h_2, we find that the current time mark $m_{cur} = 1 \neq m$. So we update m to 1 and clean all cells in the group. Then we set the bit to 1.

Query: We query each hashed cell one by one. When querying a cell in the gid-th group whose time offset is d_{gid} at time t_{cur}, we check the group and compute its age (*i.e.*, the length of time since latest cleaning). If its age, $t_{cur} + d_{gid} \mod T_{cycle}$, is less than N, it is a young cell. If $t_{cur} + d_{gid} \mod T_{cycle} = N$, it is a perfect cell. If $t_{cur} + d_{gid} \mod T_{cycle} > N$, it is an aged cell. The querying strategy for a cell is the same as that can be used for query.

4 APPLICATIONS OF SHE

In this section, we apply SHE to five well-known data structures, which are Bitmap, Count-Min Sketch, Bloom filter, HyperLogLog, and MinHash. We show the insertion and the query procedure of the five data structures, respectively. As mentioned above, N denotes the sliding window size, T_{cycle} denotes the time of the cleaning cycle, and $m[i]$ denotes the one-bit time mark of the i^{th} group.

4.1 Bitmap Using SHE (SHE-BM)

Data structure: In the bitmap, there is an M-bit array, $a[0]$, $a[1]$, . . ., $a[m-1]$, and a hash function H. The M bits are divided into G groups, each of which has $w = \frac{M}{G}$ bits. The i^{th} group has an time offset d_i.

Insert: To insert the p^{th} item with key x, we compute $i = \lfloor \frac{H(x)}{M} \rfloor$ along with $j = H(x)\%M$, and then update the j^{th} bit which belongs to the i^{th} group. We first compute the time mark $m_x = \lfloor \frac{p+d_i}{T_{cycle}} \rfloor \%2$. If $m_x = m[i]$, we set the j^{th} bit of the $i^t h$ group to 1. Otherwise, we clear the i^{th} group, set the j^{th} bit to 1, and let $m[i] = m_x$.

Query: To query the cardinality, we first compute the age of each group. Then we collect those groups whose age is within the legal range, *i.e.*, $[\beta N, T_{cycle}]$ where β is small than but close to 1, and count the number of 0 bit in these groups, which is denoted as u. Suppose there are ℓ legal groups. Then, the estimated cardinality is $-m \ln \frac{u}{w\ell}$.

4.2 Bloom filter Using SHE (SHE-BF)

Data structure: The Bloom filter has a similar data structure to that of Bitmap, except that there are k hash functions, H_1, H_2, . . ., H_k, instead of just one hash function.

Insert: The insert operation of the Bloom filter is very similar to that of the Bitmap, except that we have to update k bits with the k hash functions.

Query: To query whether an item x appears in the latest time window, we first find out the bits hashed by this item. Then, we ignore those hashed bits whose age is shorter than the sliding time window size N. For the rest hashed bits, if there is at least one 0 bit, we return that the item x does not appear in the last time window. Otherwise, we return that x appears in the latest time window.

4.3 HyperLogLog Using SHE (SHE-HLL)

Data structure: In HyperLogLog, there are M counters, $C[0]$, $C[1]$, . . ., $C[m-1]$, and two hash functions, H_c and H_z. In HyperLogLog, each group has only one counter, *i.e.*, $w = 1$.

Insert: To insert an item x at time t(x is the t^{th} item), we first use $i = H_c(x)\%M$ to select a counter, denoted as $C[i]$, and count the number of the leading 0 bits in $H_z(x)$, which is denoted as ℓ_{zero}. Then, we compute the item time mark m_x. If $m_x = m[i]$ and $\ell_{zero} \geqslant C[i]$ where $m[i]$ is the time mark of $C[i]$, we set $C[i]$ to $\ell_{zero} + 1$. If $m_x \neq m[i]$, we set $C[i]$ to $\ell_{zero} + 1$ and set $m[i]$ to m_x.

Query: To query the cardinality, we first find out those legal groups. Suppose that there are k legal groups whose values are ℓ_j ($1 \leqslant j \leqslant k$), respectively. The estimated cardinality is $\widehat{C} = ck(\sum_{j=1}^{k} 2^{-\ell_j})^{-1}M$, where c is a special constant [19].

4.4 Count-Min Sketch Using SHE (SHE-CM)

Data structure: The Count-Min Sketch has a similar data structure to that of the Bloom filter except that the cell of Count-Min Sketch is not the bits but the counters. The Count-Min Sketch also needs k hash functions, H_1, H_2, . . ., H_k

Insert: To insert an item x, we collect k counters mapped by the k hash functions. First, we compute the time marks of the groups containing these counters. If there are groups whose age is longer than T_cycle, they will be cleaned to zero. Then each of the collected counters is added one.

Query: The query operation of CM sketch is similar to the Bloom filter. We first find out the counters hashed by this item and ignore those hashed counters whose age is shorter than N. Then we

choose the minimum value among these counters as the estimated frequency of item x. According to our experiments, the counters whose age is shorter than but close to N is occasionally a good estimation. However, one important feature of the original CM sketch is that it never underestimates the frequency. To use the counters whose age is not long enough obviously goes against this feature.

4.5 MinHash Using SHE (SHE-MH)

Data structure: In MinHash, there are two counter arrays C_1 and C_2, each of which contains M counters, and there are two item streams S_1 and S_2 inserted into C_1 and C_2, respectively. There are also M hash functions $H_0, H_1, \ldots, H_{M-1}$. And each group has only one counter, *i.e.*, $w = 1$.

Insert: To insert a item x of S_x at time t, we compute M hash values, $H_0(x), \ldots, H_{M-1}(x)$. Then, we compute the item time mark m_x. For the i^{th} $(0 \leqslant i < m)$ hash value, if $m_x = m[i]$ and $H_i(x) < C_x[i]$ where $m[i]$ is the time mark of $C_x[i]$, we set $C_k[i]$ to $H_i(x)$. If $m_x \neq m[i]$, we set $C_x[i]$ to $H_i(x)$ and set $m[i]$ to m_x.

Query: To compare S_1 and S_2, we first ignore those illegal counters in C_1 and C_2, and suppose that there are k legal counters left. Then, we count the number of indexes i where $C_1[i] = C_2[i]$, which is denoted as u. Thus, the similarity between S_1 and S_2 is $\frac{u}{k}$.

5 MATHEMATICAL ANALYSIS

In this section, we analysis 1) the error caused by the on-demand cleaning and 2) the error bounds in different tasks. For time-based sliding window, we assume that the items arrive at a uniform speed. So we only need to analyse SHE based on counter-based sliding window.

5.1 Error Bound of On-demand Cleaning

In this part, we analyze the accuracy of on-demand cleaning. If every group can be updated in one time cycle T_{cycle}, the on-demand update of the group is zero error. We compute the probability that a group fails to be updated, *i.e.*, fails to be mapped by any item in a sliding window. Let n be the number of groups that fail to be updated and let $\overline{C}$ be the cardinality of a sliding window. We assume that the cardinality of the stream in a cleaning cycle, whose size is $T_{cycle} = (1+\alpha)N$, is $(1+\alpha)\overline{C}$. Thus, the number of cells that are updated in the cleaning cycle is $(1+\alpha)\overline{C}H$, where H is the number of inserted cell(s) in each insertion. The expectation of the number of groups that fail to be updated in one sliding window is $E = G \cdot (1-\frac{1}{G})^{(1+\alpha)\overline{C}H} = G \cdot e^{-\frac{(1+\alpha)\overline{C}H}{G}}$. To make $E \leqslant \varepsilon$ where ε is a small constant, we get the following inequality :

$$G \ln G * \frac{1}{(1+\alpha)\overline{C}H} \leqslant \varepsilon \tag{1}$$

In practice, when we determine an ε and compute a G, we can get the size of each group $w = \frac{M}{G}$.

5.2 False Positive Rate of the Membership Task

In this section, we estimate the false positive rate of the Bloom filter using our SHE (SHE-BF), and provide a equation to determine the value of α. Similar to the Bloom filter, the SHE-BF has a one-side error (*i.e.*, only false positive error but no false negative error). Let $R = \alpha + 1$. The estimated cardinality of data streams in a cleaning cycle of size $T_{cycle} = rN$ $(r \in [0, R))$ is rC. For a group whose age is rT, the expectation of the proportion of 0 bits in the group is $P_0(r) = (1-\frac{1}{w})^{CHr/G}$. For fixed w,G,C,H, let $Q = (1-\frac{1}{w})^{CH/G}$. Then we get the false positive rate:

$$FPR_{(R)} = \left[1 - \frac{\int_1^R P_0(r)\mathrm{d}r}{R}\right]^H = \left[1 - \frac{(Q^R - Q)}{\ln(Q)R}\right]^H$$

Let $g(R) = \frac{(Q^R - Q)}{R}$. As H and Q are fixed and $\frac{1}{\ln Q} \leqslant 0$, we can minimize $FPR_{(R)}$ by minimizing $g(R)$. To minimize $g(R)$, we take the derivative of $g(R)$ with respect to R:

$$\frac{\mathrm{d}g}{\mathrm{d}R} = Q^R \cdot [R\ln(Q) - 1] + Q$$

From the above equation, we can see that $\frac{\mathrm{d}g}{\mathrm{d}R}$ is monotonically increasing when $R \in [0, +\infty)$. Next, we solve the equation $\frac{\mathrm{d}g}{\mathrm{d}R} = 0$, and let R_0 denote the solution of the equation. Then, the optimal α is:

$$\alpha = R_0 - 1 \tag{2}$$

5.3 Error Bounds of cardinality estimation

In this section, we analyze the accuracy of SHE-BM (Bitmap using SHE), SHE-HLL (HyperLogLog using SHE), and SHE-MH (MinHash using SHE), and offer two error bounds with respect to α for the three algorithms, respectively.

SHE-BM: For SHE-BM, we analyze its error bound. Then, we discuss the value of α for stable performance. Let $F(x)$ denotes the cardinality of data streams in the sliding window of size x (*i.e.*, the recent x items). Our goal is to estimate the cardinality when $x = T$, *i.e.*, $C = F(T)$. Suppose there are m_ℓ bits whose ages are legal in the SHE-BM, and u bits of value 0 among the m_ℓ bits.

We first analyze the over-estimation case. When the age of a group, *i.e.*, x, is within $[(1-\alpha)T, T)$, the largest possible valve of $F(x)$ is C. When the age of a group is within $[T, (1+\alpha)T)$, the largest estimation of the cardinality is $C + (x - T)$. Therefore, we get the upper bound:

$$F_{upper}(x) = \begin{cases} C, & \textbf{if } (1-\alpha)T \leqslant x < T \\ C + (x - T), & \textbf{if } T \leqslant x < (1+\alpha)T \end{cases}$$

Then, we can get a lower bound of u, *i.e.*, the number of 0 bits among the m_ℓ legal bits.

$$E_{lower}\left[\frac{\widehat{u}}{m_\ell}\right] = \frac{1}{2}\left[\frac{\int_0^{\alpha T}(1-\frac{1}{m_\ell})^{C+x}\mathrm{d}x}{\alpha T} + \left(1-\frac{1}{m_\ell}\right)^C\right] \geqslant e^{-\frac{C}{m_\ell}} \cdot \left(1 - \frac{\alpha T}{4m_\ell}\right)$$

Therefore, the upper bound of the estimated cardinality $\widehat{C}$ is:

$$E[\widehat{C}] = -m_\ell \cdot \ln\left(E_{lower}\left[\frac{\widehat{u}}{m_\ell}\right]\right) \leqslant C + \frac{\alpha T}{4}$$

In the same way, we can get the lower bound of the estimated cardinality $\widehat{C}$:

$$E[\widehat{C}] \geqslant C - \frac{\alpha T}{4}$$

Finally, the error bound of SHE-BM is:

$$\left|\frac{E[\widehat{C}]-C}{C}\right| \leqslant \varepsilon = \frac{\alpha T}{4C} \tag{3}$$

Therefore, we can adjust the error bound ε by setting α.

From Equations (3), we can see that the error bound of $E[\widehat{C}]$ is tighter when α is smaller. However, α cannot be too small, because a small α can lead to a large variance for $E[\frac{\widehat{u}}{m_\ell}]$, and thus lead to a large variance for $E[\widehat{C}]$. Specifically, suppose p is the true proportion of 0 bits in legal bits (*i.e.*, the bits in legal groups), then the variance of $E[\frac{\widehat{u}}{m_\ell}]$ is:

$$Var\left(E\left[\frac{\widehat{u}}{m_\ell}\right]\right) = \frac{p}{m_\ell}$$

Therefore, $m_\ell = \frac{2\alpha}{1+\alpha}m = (2-\frac{2}{1+\alpha})m$ cannot be too small, and thus α cannot be too small.

SHE-HLL: The analyzing procedure is similar to that of SHE-BM. For a group, let ρ_{max} be the expectation of the position in which the leftmost "1" is. Then we can get an upper bound of $\widehat{\rho}_{max}$:

$$E[\widehat{\rho}_{max}] \leqslant \frac{1}{2}\cdot\left[log_2\left(\frac{C}{G}+\frac{\alpha T}{2G}\right)+\log_2\left(\frac{C}{G}\right)\right]$$

Then, the upper bound of the estimated cardinality $\widehat{C}$ is:

$$E[\widehat{C}] \leqslant C + \frac{\alpha T}{4}$$

In the same way, we can get the lower bound, and finally we get the following error bound:

$$\left|\frac{E[\widehat{C}]-C}{C}\right| \leqslant \varepsilon = \frac{\alpha T}{4C}\cdot\left[1+O\left(\frac{\alpha T}{C}\right)\right] \tag{4}$$

SHE-MH: Let $F(x)$ be the similarity between two streams in the sliding window of size x, and $S = F(T) = \frac{S_\cap}{S_\cup}$ be the similarity of the two streams when the sliding window size is T, where $S_\cap$ and $S_\cup$ are the size of the intersection set and the union set of the items in the two streams, respectively. Then, we compute the error bound of $E[\widehat{S}]$. The worst case of over-estimation is that $F(x)$ sharply decrease when $x \in [(1-\alpha)T, T]$ and sharply increased when $x \in [T, (1+\alpha)T]$. In this situation, we overestimate the similarity and therefore the upper bound of the estimated similarity is:

$$\begin{aligned} E[\widehat{S}] =& \frac{1}{2}\left(\frac{\int_0^{\alpha T}\frac{S_\cap}{S_\cup-2x}\mathrm{d}x}{\alpha T}+\frac{\int_0^{\alpha T}\frac{S_\cap+x}{S_\cup+x}\mathrm{d}x}{\alpha T}\right) \\ \leqslant& \frac{S_\cap}{S_\cup}+\left[\frac{1}{4}\varepsilon+\frac{1}{6}\varepsilon^2+O(\varepsilon^3)\right], \text{ where } \varepsilon = \frac{2\alpha T}{S_\cup}. \end{aligned}$$

We can get the lower bound of $E[\widehat{S}]$ in the same way, and the final error bound of $E[\widehat{S}]$ is:

$$\left|E[\widehat{S}]-S\right| \leqslant \frac{1}{4}\varepsilon+\frac{1}{6}\varepsilon^2 \tag{5}$$

According to Equation (5), the bias $|E[\widehat{S}]-S|$ can be bounded by ε, where ε is a small constant related to α. Therefore, we can have a tight error bound by adjusting α.

6 IMPLEMENTATION ON FPGA

In this section, we show that the hardware version of SHE is able to be implemented on the pipeline architecture and meets with the constraints mentioned in Section 2.3.

We prove that the memory usage of SHE is small theoretically and practically, and it is feasible to implement SHE in SRAM. In Section 5, the error bound of the cardinality estimation is proportional to the size of sliding window. Fig.9 shows that when memory size is under 2MB, SHE-CM achieves a acceptable performance for frequency estimation while the average relative error of SWAMP is more than 10. To achieve a extremely high accuracy, the memory usage of other SHE algorithms is no more than 128KB, which can undoubtedly fit in SRAM.

SHE can be implemented on the pipeline architecture with constraints of single stage memory access and limited concurrent memory access. As shown in Section 3.3, the cleaning and insertion process of SHE-BM on the counter-based sliding window can be concluded as four stages:

(1) On receiving a fixed-length key of an input item, we first get the time from the item counter and update the item counter.
(2) A hash function is used to calculate the index of the mapped bit.
(3) The new time marks are calculated for each of the groups in parallel, and then we update the stored time mark of the mapped group after comparing it with the new time mark.
(4) The last stage is to update the mapped bit and group according to the comparison of the new time mark and the stored time mark.

Each memory region, including the time marks and the mapped bit, is accessed in one stage, and therefore meets with the constraints of single stage memory access. Furthermore, the third constraints, *i.e.*, limited concurrent memory access, is sufficient since there is only one memory address to be accessed with at most the size of a group for each stage. The insertion process of SHE-BF and other SHE algorithms is barely the same as SHE-BM.

We implement the SHE-BM and SHE-BF on the Virtex-7 family of Xilinx FPGA (xc7vx690t) [13] and achieve the processing speed of 544 Mips(million items per second). In our FPGA implementation of SHE-BM, we set the size of group to 64 bits and the size of bit array to 1024 bits. Therefore, we use one 1024-bit register for the bit array. The item counter is implemented as a 32-bit register. The settings of SHE-BF are the same as SHE-BM but there are 8 identical processes in the implementation of SHE-BF.

Table 2 shows the evaluation of the resource utilization of SHE-BM and SHE-BF and Table 3 shows the performance. The clock frequency of our implementation of SHE-BM achieves 544MHz, which means that the processing speed is 544 Mips(million items per second). Since the 8 identical insertion processes in SHE-BF can work in parallel on FPGAs, the frequency with SHE-BF is barely the same as with SHE-BM. The FPGA clock frequency with both SHE-BM and SHE-BF achieves more than 200MHZ, which is a typical FPGA clock frequency [29].

Table 2: Resource utilization of FPGA implementation.

	LUT	Register	Block Memory
SHE-BM	1653(0.38%)	1509(0.17%)	0
SHE-BF	12875(2.97%)	11790(1.36%)	0

Table 3: The clock frequency of FPGA implementation.

	Clock Frequency(MHz)
SHE-BM	544.07
SHE-BF	468.82

7 EXPERIMENTAL RESULTS

We test the **hardware version** of SHE on both CPU platform and FPGA platform. The result on FPGA has been shown above. On CPU platform, each of the five SHE-algorithms are compared to prior works and the **ideal goal**, which is the performance that the original algorithms without SHE achieve by treating each window as static data. All related source codes are anonymously released at Github [30].

7.1 Experimental Setup

1. Datasets: The following datasets is used in our experiments.

- **CAIDA:** We use the public traffic dataset released by CAIDA[31] to test all algorithms mentioned above except SHE-MH. Each trace collected from the dataset contains approximately 30M items and 600K distinct items (srcIP).
- **Distinct Stream:** We test SHE-BF with additional synthetic dataset where the frequency of each distinct item is 1. This dataset is used to simulate the worst case when applying SHE-BF.
- **Relevant Stream:** For SHE-MH, we conduct experiments on synthetic datasets generated from the traces collected from IMC10 [32]. Each synthetic dataset has two traces, each of which contains 100K distinct items (2.5M items).
- **Other Datasets:** We test the processing speed of the SHE on the CAIDA along with the two other datasets, Campus and Webpage. Campus contains IP traces captured by the main gateway of our campus. Webpage is obtained from the public dataset repository *Frequent Itemset Mining Dataset Repository* [33] collected from a number of web pages.

2. Evaluation metrics:

- **FPR (False Positive Rate):** $\frac{n}{m}$, where m denotes the number of queried items which does not appear in the latest sliding window. By default, We query items which do not present in recent $(1+\alpha)T$ items to calculate the FPR. We use FPR to evaluate the accuracy of the membership task.
- **RE (Relative Error):** $\frac{|f-\widehat{f}|}{f}$, where f denotes the true value of the measurement results and $\widehat{f}$ denotes the estimated measurement result of f. We use RE to evaluate the accuracy of SHE-BM, SHE-HLL and SHE-MH. For SHE-BM and SHE-HLL, f is the number of distinct items within the sliding window, which is called cardinality. For SHE-MH, f is the similarity[21, 34] of the two data stream within the sliding window.
- **ARE (Average Relative Error):** $\frac{1}{N}\sum_{i=1}^{N}\frac{|f_i-\widehat{f}_i|}{f_i}$, where N denotes the number of distinct items within the sliding window, f_i denotes the true frequency of the item i and $\widehat{f}_i$ denotes the estimated frequency of item i. We use ARE to evaluate the accuracy of SHE-CM.
- **Throughput:** We use Mips (million insertions per second) to evaluate the throughput of insertion for each algorithm.

3. Default Settings: We implement the **hardware version** of SHE and the other algorithms in C++, and use BOBHash [35] as the hash function. Before evaluating the performance, we feed enough items until the performance is stable.

The default parameters are described as follows. The window size N is set to 2^{16} items. The number of cells in a group, denoted as w, is 64 SHE-BF, SHE-BM and SHE-CM, and is 1 for SHE-HLL and SHE-MH. α, defined as $\frac{T_{cycle}-N}{N}$, varies among different applications, which depends on the distinguishing features of these algorithms. For SHE-BM, SHE-HLLand SHE-MH, it is set to 0.2 by default. For SHE-CM, it is set to 1 and for SHE-BF, it is set to 3. More detailed settings are listed below.

- **SHE-BM:** We compare SHE-BM with three algorithms: TSV [24], CVS [23], and SWAMP [27]. For TSV, we use the 64-bit timestamp. For CVS, we set the maximum value of its counter to 10. For SWAMP, we use its DISTINCTMLE estimator. For SHE-BM, all the parameters are using the default settings.
- **SHE-HLL:** We compare two algorithms: SHLL [22] and SHE-HLL. For the SHLL, we store the 64-bit timestamp. For both of algorithms, we calculate 32-bit hash values and store the numbers of leading 0 of these hash values in 5-bit cells. We set the window size to 2^{21} because HyperLogLog is usually used to estimate massive cardinality.
- **SHE-CM:** We compare SHE-CM with two algorithm: ECM [36] and SWAMP. For ECM, we set the number of hash function to 4. For SHE-CM, we set the number of the hash function to 8.
- **SHE-BF:** We compare SHE-BF with three algorithms: TBF [26], TOBF [25], and SWAMP. For TBF, we set the size of each counter to 18 bits and the number of hash functions to 8. For TOBF, we use the 64-bit timestamp. For SWAMP, we use its ISMEMBER estimator. For SHE-BF, we fix the number of hash functions to 8. α is determined according to Equation (2), which is roughly 3.
- **SHE-MH:** We compare two algorithms: the straw-man MinHash and SHE-MH. The straw-man MinHash is the modified MinHash by adding a 64-bit timestamp for each pair of counters to indicate if the counters need to be cleaned. The outputs of hash functions used in both algorithms are 24-bit integers.

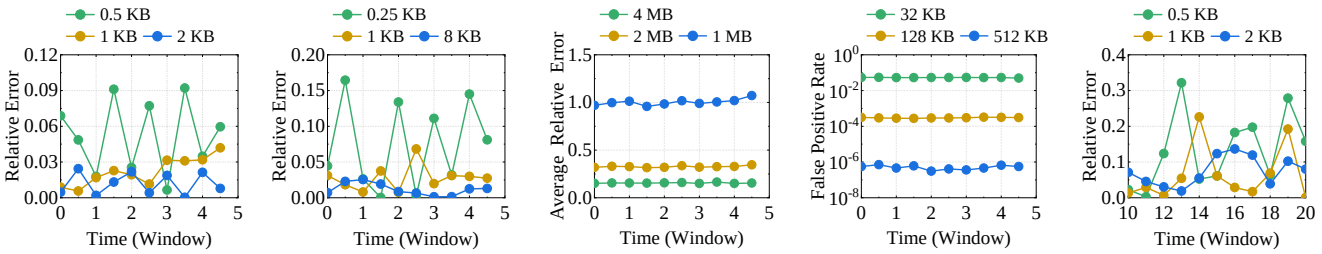
7.2 Impact of Parameters

1) Common Parameters:

Performance *vs.* time (Fig. 5): We find that the SHE is stable as time goes by and the window slides. We test our algorithms every half window on the same stream and we test the algorithms using three different sizes of memory. When given enough memory, the performance is stable especially for SHE-BF and SHE-CM.

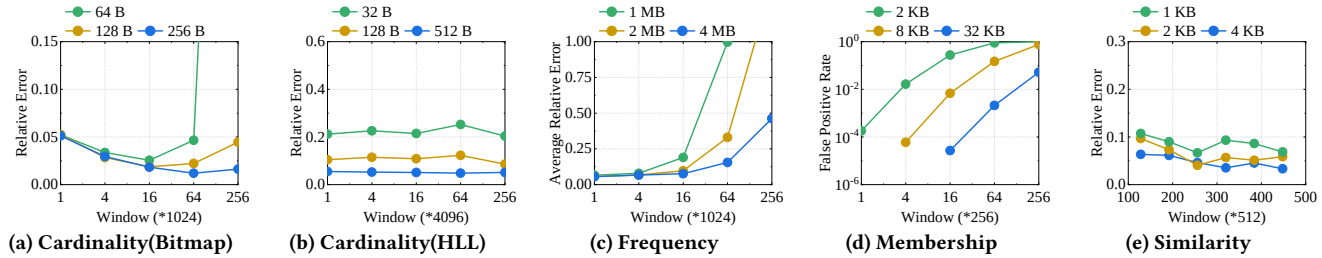
Performance *vs.* window size (Fig. 6): We find that the SHE is stable to the window size when other parameters are fixed. We test our algorithms for three different sizes of memory. The performance of the SHE is actually similar to the ideal goal. For example, SHE-HLL and SHE-MH keep almost constant ARE because the original algorithms are not sensitive to the size of data.

Performance *vs.* α (Fig. 7): The experimental results of the other three algorithms (SHE-CM, SHE-HLL, and SHE-MH) are similar to SHE-BM. Therefore, we only show the performance of the SHE-BF and SHE-BM. The optimal α is computed according to Equation (2). As shown in Fig. 7a, the SHE-BF using the optimal α performs



(a) Cardinality (Bitmap) (b) Cardinality (HLL) (c) Frequency (d) Membership (e) Similarity

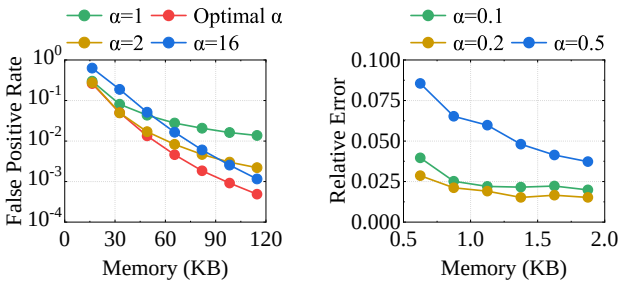
Figure 5: The stability of SHE as the window slides with time.



(a) Cardinality(Bitmap) (b) Cardinality(HLL) (c) Frequency (d) Membership (e) Similarity

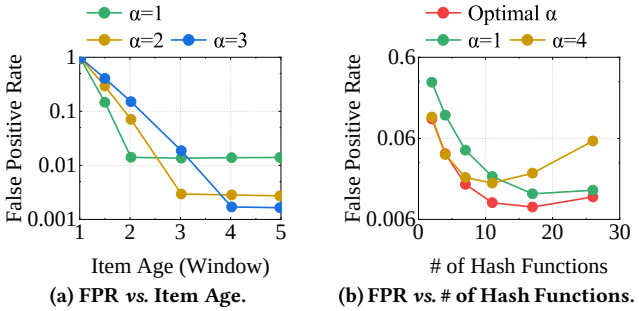
Figure 6: The adaptation for different window size.
When the 'Window(*k)' is x, it means that the window size is x*k.

well on the real dataset. For SHE-BM and other SHE-algorithms, the SHE performs well when α is from 0.2 to 0.4 as an empirical setting.



(a) SHE-BF (b) SHE-BM

Figure 7: Performance *vs.* α.



(a) FPR *vs.* Item Age. (b) FPR *vs.* # of Hash Functions.

Figure 8: Parameters of SHE-BF.

2) Parameters of SHE-BF:

FPR *vs.* time age (Fig. 8a): The figure shows that the FPR becomes stable when the item age is increasing. The item age is the time spans from the item's arrival to the current time. We test the SHE-BF on Distinct Stream and repeated our experiment for 10, 000 times. As the item age increases, the FPR decreases at a nearly exponential speed until the item age is larger than the size of relaxed window.

FPR *vs.* number of hash functions (Fig. 8b): The optimal α depends on the number of hash functions according to Equation 2. We find that the optimal α for different number of hash functions estimated by Equation 2 works well on Distinct Stream, which is the worst case for Bloom filter as mentioned in Section 7.1.

7.3 Accuracy Performance

We compare the accuracy of SHE with both the state-of-the-art and the ideal goal in the five tasks. **The ideal goal** for each measurement task is the accuracy achieved if we treat the sliding window task as a fixed window task. For example, we insert all items in the sliding window to an empty Bloom filter [17], and calculate the membership accuracy by it.

SHE-BM *vs.* Others (Fig. 9a): We find that the SHE-BM achieves a much more stable and precise performance in a wide range of memory than the state-of-the-art. To achieve 0.01 ARE, the SHE-BM uses 1KB memory while SWAMP needs more than 100KB memory. Furthermore, when the memory size is limited under 3KB, only the SHE-BM make a good estimation while others are not.

SHE-HLL *vs.* Others (Fig. 9b): We find that the SHE-HLL is about 10 times more accurate than the SHLL when the memory size is less than 16KB. The SHE-HLL achieves 0.02 ARE when the memory size grows to more than 4KB, and it is quite close to the ideal goal.

SHE-CM *vs.* Others (Fig. 9c): We find that the SHE-CM is often 10 times more accurate than the competitors. SWAMP only works when the memory is sufficient while its accuracy is barely acceptable when the memory is scarce.

SHE-BF *vs.* Others (Fig. 9d): We find that the SHE-BF is much more accurate than other algorithms. Specifically, the FPR of the

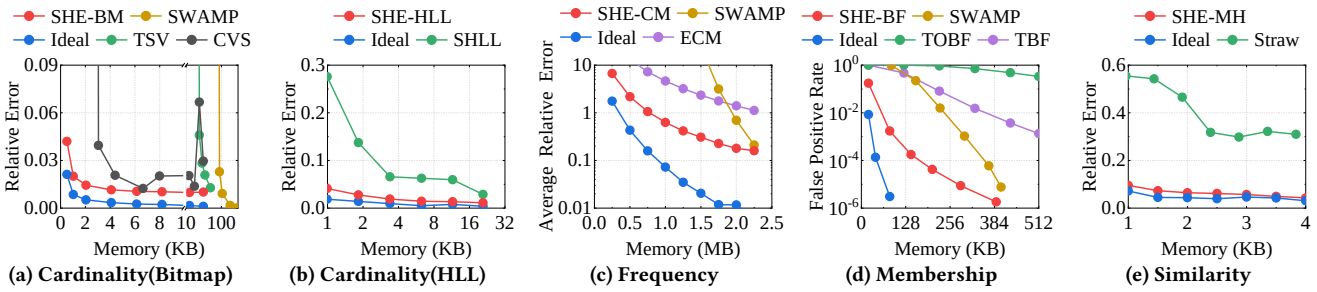
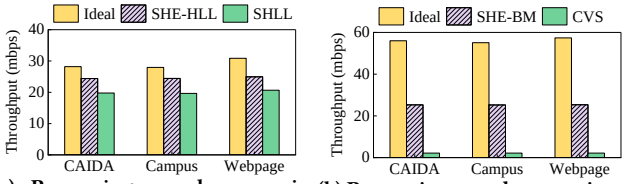


Figure 9: Accuracy comparison for five tasks.



(a) Processing speed comparison for SHE-HLL on different dataset.

(b) Processing speed comparison for SHE-BM on different dataset.

Figure 10: Processing speed comparison for two specific tasks.

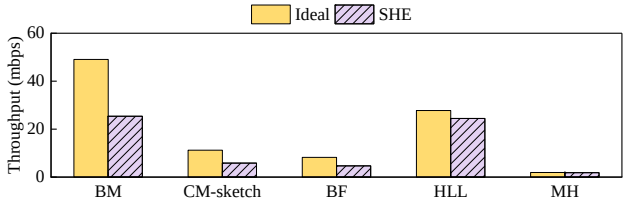


Figure 11: Processing speed comparison with the ideal goal.

SHE-BF is roughly 100 times lower than all of the other algorithms when the memory size is under 256KB. Even when the memory is more than 256KB, the SHE-BF is still better than SWAMP.

SHE-MH *vs.* Others (Fig. 9e): The SHE-MH is about 10 times more accurate than the straw-man MinHash with same memory footprints. The performance of SHE-MH is almost the same with the ideal goal when the memory size grows.

7.4 Throughput

As shown in Fig. 10, SHE is much faster than other sliding-window algorithms on the three different datasets. We find that the processing speeds of *SHE*-algorithms are comparable to those of the original algorithms as shown in Fig. 11.

8 CONCLUSION

This paper proposes the SHE for high-speed data stream mining over sliding windows. It is a generic framework which can extend common fixed window algorithms to sliding windows. SHE is a hardware friendly algorithm because it meets the three major constraints for dedicated hardware platforms: 1) limited SRAM memory, 2) single stage memory access and 3) limited concurrent memory access. SHE uses circular cleaning to handle out-dated information at low memory cost and uses age-sensitive selection to choose proper cells for query. Additionally, SHE proposes group update and on-demand update to limit concurrent memory accesses. We implemented SHE on FPGA, a representative of dedicated hardware platform. We show case application of SHE to five well-known fixed window algorithms, achieving up to 100 times lower error compared with the state-of-the-art. All related source codes are anomalously released at Github [30].

REFERENCES

[1] Yongpeng Zhang and Frank Mueller. Gstream: A general-purpose data streaming framework on gpu clusters. In *ICPP*, pages 245–254, 2011.

[2] Aiyou Chen, Li Erran Li, and Jin Cao. Tracking cardinality distributions in network traffic. In *INFOCOM' 09*, pages 819–827, 2009.

[3] Chen Qian, Hoilun Ngan, Yunhao Liu, and Lionel M Ni. Cardinality estimation for large-scale rfid systems. *IEEE transactions on parallel and distributed systems*, 22(9):1441–1454, 2011.

[4] Fang Hao, Murali Kodialam, TV Lakshman, and Haoyu Song. Fast multiset membership testing using combinatorial bloom filters. In *IEEE INFOCOM 2009*, pages 513–521. IEEE, 2009.

[5] Yu Hua, Bin Xiao, Bharadwaj Veeravalli, and Dan Feng. Locality-sensitive bloom filter for approximate membership query. *IEEE Transactions on Computers*, 61(6):817–830, 2011.

[6] Graham Cormode and Shan Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[7] Ondrej Chum, James Philbin, Andrew Zisserman, and et al. Near duplicate image detection: min-hash and tf-idf weighting. In *BMVC' 08*, pages 812–815, 2008.

[8] Bryan Ball, Mark Flood, Hosagrahar Visvesvaraya Jagadish, and et al. A flexible and extensible contract aggregation framework (caf) for financial data stream analytics. In *DSMM' 14*, pages 1–6, 2014.

[9] Lajos Gergely Gyurkó, Terry Lyons, Mark Kontkowski, and Jonathan Field. Extracting information from the signature of a financial data stream. *arXiv preprint arXiv:1307.7244*, 2013.

[10] Sang-Hyun Oh, Jin-Suk Kang, Yung-Cheol Byun, and et al. Anomaly intrusion detection based on clustering a data stream. In *ISC' 06*, pages 415–426, 2006.

[11] Mustafa Amir Faisal, Zeyar Aung, John R Williams, and et al. Securing advanced metering infrastructure using intrusion detection system with data stream mining. In *PAISI' 12*, pages 96–111, 2012.

[12] Introducing data center fabric, the next-generation facebook data center network. https://goo.gl/makVDo.

[13] Virtex 7 series fpga white paper. https://www.xilinx.com/products/silicon-devices/fpga/virtex-7.html, 2020.

[14] Ran Ben-Basat, Xiaoqi Chen, Gil Einziger, and et al. Efficient measurement on programmable switches using probabilistic recirculation. In *ICNP' 18*, pages 313–323, 2018.

[15] Cristian Estan and George Varghese. New directions in traffic measurement and accounting. In *Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 323–336, 2002.

[16] Robert Schweller, Zhichun Li, Yan Chen, Yan Gao, Ashish Gupta, Yin Zhang, Peter A Dinda, Ming-Yang Kao, and Gokhan Memik. Reversible sketches: enabling monitoring and analysis over high-speed data streams. *IEEE/ACM Transactions on Networking*, 15(5):1059–1072, 2007.

[17] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[18] Kyu-Young Whang, Brad T Vander-Zanden, and Howard M Taylor. A linear-time probabilistic counting algorithm for database applications. *ACM Transactions on Database Systems (TODS)*, 15(2):208–229, 1990.

[19] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and et al. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. In *DMTCS' 07*, pages 137–156, 2007.

[20] Frédéric Giroire. Order statistics and estimating cardinalities of massive data sets. *Discrete Applied Mathematics*, 157(2):406–427, 2009.

[21] Andrei Z Broder. On the resemblance and containment of documents. In *Sequences '97*, pages 21–29, 1997.

[22] Yousra Chabchoub and Georges Hébrail. Sliding hyperloglog: Estimating cardinality in a data stream over a sliding window. In *ICDMW' 10*, pages 1297–1303, 2010.

[23] Jingsong Shan, Jianxin Luo, Guiqiang Ni, and et al. Cvs: fast cardinality estimation for large-scale data streams over sliding windows. *Neurocomputing*, 194:107–116, 2016.

[24] Hyang-Ah Kim and David R O'Hallaron. Counting network flows in real time. In *GLOBECOM' 03*, pages 3888–3893, 2003.

[25] Shijin Kong, Tao He, Xiaoxin Shao, and et al. Time-out bloom filter: A new sampling method for recording more flows. In *ICOIN' 06*, pages 590–599, 2006.

[26] Linfeng Zhang and Yong Guan. Detecting click fraud in pay-per-click streams of online advertising networks. In *ICDCS' 08*, pages 77–84, 2008.

[27] Eran Assaf, Ran Ben Basat, Gil Einziger, and et al. Pay for a sliding bloom filter and get counting, distinct elements, and entropy for free. In *INFOCOM' 18*, pages 2204–2212, 2018.

[28] Gil Einziger and Roy Friedman. Counting with tinytable: Every bit counts! In *2015 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 77–78. IEEE, 2015.

[29] Bojie Li, Kun Tan, Layong Luo, and et al. Clicknp: Highly flexible and high performance network processing with reconfigurable hardware. In *SIGCOMM '16*, pages 1–14, 2016.

[30] The source codes of our and other related algorithms. https://github.com/Sliding-Hardware-Estimator/SlidingHardwareEstimator.

[31] The caida anonymized internet traces, 2020. http://www.caida.org/data/overview/.

[32] Theophilus Benson, Aditya Akella, and David A Maltz. Network traffic characteristics of data centers in the wild. In *IMC' 10*, pages 267–280, 2010.

[33] Frequent itemset mining dataset repository. http://fimi.ua.ac.be/data, 2004.

[34] Wen Xia, Hong Jiang, Dan Feng, and Yu Hua. Similarity and locality based indexing for high performance data deduplication. *IEEE Transactions on Computers*, 64(4):1162–1176, 2014.

[35] Bob jenkins' hash function web page, paper published in dr dobb's journal, 2008. http://burtleburtle.net/bob/hash/doobs.html.

[36] Odysseas Papapetrou, Minos Garofalakis, and Antonios Deligiannakis. Sketch-based querying of distributed sliding-window data streams. *VLDB Endow.*, 2012.