Applying Sliding Window to On-Off Sketch Model

CS5234 Mini Project Report

Bennett Clement
A0200739J

Ma Jiameng
A0198964A

ABSTRACT

Streaming data processing is widely used in large data analysis. Algorithms to find properties such as frequency and cardinality of streaming data have been widely studied, and the state-of-art models achieved promising results. Recently, there has been an increase of interest in persistence, which is the number of time slices (each time slice is a fixed non-overlapping period) an event appears. Knowing the persistence of an items helps to uncover network attacks, the loyalty of website users and so on. There are two main problems related to persistence, the Persistence Estimation and Finding Persistent Items. A recent paper proposed On-Off Sketch [1] to answer these two persistence problems. Essentially, the On-Off Sketch is a Count-Min Sketch with modified counters, where a state field is attached to each counter. The state field allows us to limit each counter to be incremented at most once within a time slice. The method is simple and elegant and has been shown to have a higher accuracy and throughout compared to some state-of-art models. In our report, we did an experimental extension to the On-Off Sketch by applying the Sliding Sketch technique [2] to periodically remove the outdated persistence in the past and keep tracking with only the up-to-date persistence of our interest. Our implementation further adjusts the counter structure to become a bucket with multiple counter fields to store the persistence of the latest past periods. The experimental results shows that our implementation of sliding windows to the On-Off Sketch model achieved good accuracy.

INTRODUCTION

Define time slices as a fixed period such as 5 minutes. Multiple time slices are non-overlapping. Persistence is then defined as the number of time slices an item appears. There are 2 main questions related to persistence: Persistence Estimation (PE) and Finding Persistent Items

(FPI). Persistence Estimation returns an estimated persistence of an event in the stream while Finding Persistent Items returns a list of items whose persistence are beyond a given threshold. As the name indicates, persistence shows how persistent an item appears in a data stream. The difference between persistence and frequency is that the persistence cares about an item's existence along the time, while frequency cares more on an item's number of contributions. Persistence has many practical applications, ranging from network quality inspection to anomalies detection in advertisement, e.g., click fraud detection. To better illustrate the difference, take an example of tracking the number of weeks a user visited a website. If a user A visits a website once a week, his persistence within a month would be 4 and his frequency would also be 4. Suppose another user B signed up and browsed the website 6 times within a week and afterwards stopped using the website, his frequency would be 6, but his persistence would only be 1. In this simple example, persistence is a better metric to know who a website's potential long-term customers are and the website owners can use the information to better target that market.

Recently, Zhang, et al. [1] proposed a model called On-Off Sketch for persistence estimation. Essentially, the On-Off Sketch model is the Count-Min Sketch with modified counters. Each counter is associated with an additional state bit to adapt to the concept of time slices. With the state bit, the counter can be incremented at most once within a time slice. This behavior changes the Count-Min Sketch from storing the cumulated count (frequency) information to storing persistence information.

In general, we care more about the recent information compared to the older information. However, currently the proposed On-Off Sketch stores all persistence information from the beginning of time. Hence, we need a way to remove this "old" information. Sliding Sketches [2] is proposed as a generic model to removes this "old" information and has been applied to several sketches that stores membership, frequency, or heavy hitter information. As the On-Off Sketch structure is essentially a modified Count-Min Sketch (a sketch that stores frequency information), the Sliding Sketch technique is theoretically applicable to the On-Off Sketch model too.

In this report, we explore the viability of applying On-Off Sketch to a windowed stream setting, in which we care about the N most recent time windows instead of all seen time windows. This report consists of three sections. We will first summarize related works on On-Off sketch and

Sliding Sketches. Then we shall present our proposed Sliding On-Off Sketch. Lastly, we will provide experimental results and analysis for our implementation.

CONTRIBUTION

This report attempts to extend On-Off sketch to get persistence for recent items. To do so, we provide an implementation of a sliding window variant to On-Off Sketch and provide experimental results by varying parameters and briefly justify its correctness.

Limitations

There is no state-of-the-art sliding window algorithm for persistence to the best of the author's knowledge, but if there is (or a simple adaptation from the more studied frequency problem), it would be interesting to see how the results reported in this report compares to those of that algorithm.

Another limitation is that this paper couldn't report memory usage accurately, as our implementation uses Python which is not using manual memory management and hence the memory seen when running the experiments are an overestimate of the algorithm's real memory usage.

RELATED WORK

Our projects are based on the previous work on the On-Off sketch model for analyzing persistence and the Sliding Sketch technique for periodically obsoleting outdated data information. Below we briefly summarize the data structures, operations, and evaluation for the two models.

1. On-off Sketch

a. Model structure

The On-Off Sketch model is in essence a Count-Min sketch. The Count-Min sketch model, as illustrated in Figure.1, consists of d arrays of l counters. Each array is associated to a hash function h_i which maps incoming data of x to the corresponding counters $C(i, h_i(x))$. The selected counter in each array would increment by 1 when an item is inserted. Upon query of an item x, the frequency of the x is obtained by finding the minimum value of among the counters $C(i, h_i(x))$ in the d arrays.

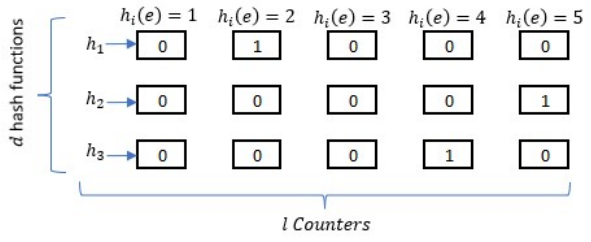
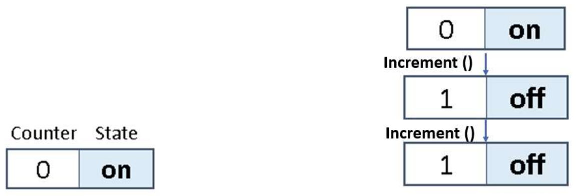


Figure 1. Count-Min Sketch Model Structure

The On-Off sketch model modifies the counter structure by adding a state field of state 'On' or 'Off'. The new counter structure is referred to as StateCounter shown in Figure 2 which is the building block for On-Off Sketch. When the counter state is 'On', the counter can be incremented. When the counter state is 'Off', the counter would not allow modification to itself. The data structure for Persistence Estimation and Finding Persistent Items are shown in Figure 3 and Figure 4.



(a) **(b)**

Figure 2. a single StateCounter (a) and its basic operation (b)

b. Operation: Persistence Estimation (PE)

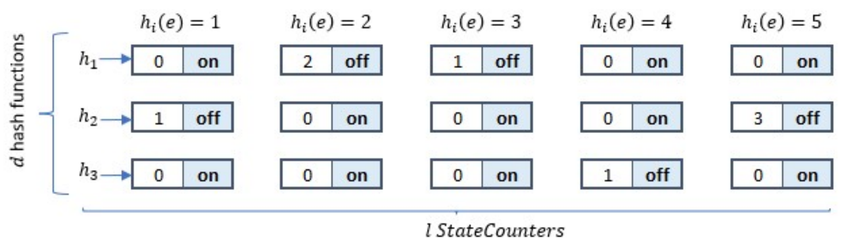


Figure 3. On-Off Sketch Model Structure for Persistence Estimation

There are four main operations for Persistence Estimation:

- **pe.init (x) :** initializing the data structure
  - i. set counters to 0;
  - ii. set states to 'on'
- **pe.insert (x)** : to insert an item x
  - i. locate one counters $C(i, h_i(x))$ in each array by its hash functions.
  - ii. if counter state is 'On', increment counter by 1 and set state to 'Off'.
  - iii. if counter state is 'Off', do nothing.

  It limits the counter to increment at most once in a time slice
- **pe.new_slice ()** : invoked when a new time slice starts
  - reset all counter states to 'On'.

  This allows counter to be incremented in the next time slice.
- **pe.query (x) :** query the persistence of an item x,
  - locate one counters $C(i, h_i(x))$ in each array by its hash functions
  - return the minimum value among the counters as item persistence.

c. Operation: Finding Persistent Items

For Finding Persistence item, the model structure is modified as Figure 4. There are two changes to the existing model. First, we only use one hash function, mapped to l StateCounters. Secondly, each StateCounter is now associated to w StateBuckets. A StateBucket is a key-value pair with the item id as key and a StateCounter as its value.

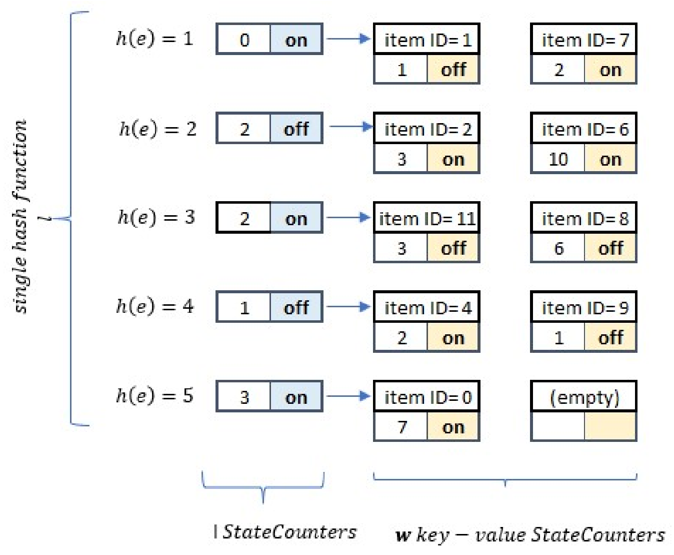


Figure 4. On-Off Sketch Model Structure for Finding Persistent Item

The four operations for Finding Persistent Items are:

a. **fpi.init ($\boldsymbol{x}$)** : initializing data structure
   i. set counter to 0;
   ii. set state to 'on'
   iii. the item ids in StateBuckets are empty

b. **fpi.insert ($\boldsymbol{x}$)** : to insert an item x
   i. the hash function first look for the group associated with counter $C(i, h(x))$.
      i. if item x is one of the key inside the w StateBuckets and the state is 'On', increment the StateBucket by 1 and set the bucket state to 'Off'. Otherwise, if item is found but the state is 'Off', do nothing.
      ii. if the item x is not found inside any StateBuckets, it takes one of the actions following the condition checks below:
         1. if there is an empty StateBucket, fill the item to the empty StateBuckets, set value be 1 and state to 'Off'.
         2. Otherwise: if the StateCounter is 'On', increments StateCounter by 1 and set state to 'Off'. If the StateCounter's value is larger than the minimum StateBucket value. Swap the StateCounter and item id with the StateBucket. Otherwise: do nothing.

c. **fpi.new_slice ()** : invoked when a new time slice starts,
   i. reset all counter states to 'On'.

d. **fpi.query ($\boldsymbol{th}$) :** query for all items id with persistence larger than the given threshold

i. scan through all StateBucket, return the item id if the StateBucket value larger than the threshold

d. Model Performance

Both sketches are proven in the paper [1] to have small error bounds and is only vulnerable to overestimation error. It is also shown experimentally in the paper to have smaller errors and higher throughput than other state-of-art algorithms.

2. Sliding Window

a. Model structure

Sliding window adds another layer of operation to the sketch model. Take the Count-Min sketch (Figure 1) as an example, the original counter structure inside the Count-Min sketch model with Sliding Window (Figure 5) would now be a SlidingCounter containing ***k*** counters storing the information in the past. Let the most recent value be stored in B^{new} and the least recent one in B^{old}. The counters are then periodically selected by a scanning pointer. When a counter is selected, it would remove B^{old}, move all fields to an older field, and insert a new field B^{new} initialized to 0.

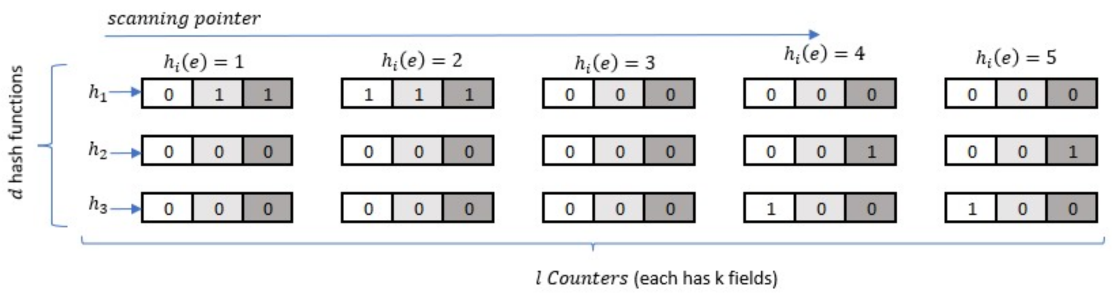


Figure 5. Count-Min Sketch Model with Sliding Window

Count-Min Sketch is composed of arrays of counters. For each array, we introduce a scanning pointer, an index (integer) from 0 to *N-1,* that indicates the phase of our sliding window with length *N*. When the scanning pointer reaches the end of the array, it will loop back to the beginning of the array. Before every operation, the scanning pointer would scan $\frac{(k-1)\times l}{N}$ counters along each array and updates the fields inside the counters. Hence, within one sliding window duration, the scanning pointer would scan the whole array of *l* SlidingCounters and ends up pointing to the start of the array, ready for the next sliding window.

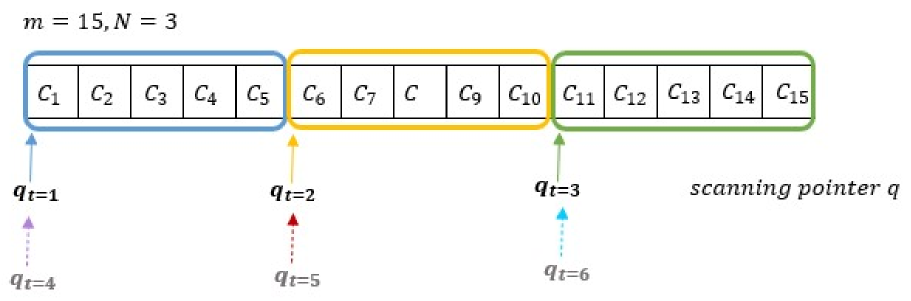


Figure 6. An illustration of the StateBucket update process with scanning pointer

Figure 6 gives an example to show the movement of scanning pointer. Let $m = 15$, the number of SlidingCounters in an array and $N = 3$, the sliding window size. Here, we assume k = 2.

- After the first item arrives, the scanning pointer, which currently points to the first SlidingCounter C_1 in the array, performs updates for 5 SlidingCounter ($\frac{m}{N} = 5$) C_1 to C_5. After the SlidingCounter update, the scanning pointer now points to C_6
- After the second item arrives, the SlidingCounters C_6 to C_{10} are updated, and the scanning pointer ends up pointing to SlidingCounter C_{10}.
- Similarly, after the third item arrives, the SlidingCounter sC_{11} to C_{15} are updated. Now the scanning pointer, hitting the last SlidingCounter in the array, would point back to the starting point of the array, which is C_1.
- Through the process, all SlidingCounter updates are completed within one sliding window period.

b. Model Performance

Recall that the Count-Min Sketch queries all hash functions and return the minimum value. To preserve its overestimation error feature, the paper [2] uses the "sum" aggregation strategy, that is taking the sum of all k counters in a SlidingCounter as the counter value.

Jet lag δ is introduced in the paper of Sliding Sketch [2] to indicate the progress of the sliding window at the point of time when a query is made. The paper proved that this model would only have an overestimation error because the returned results are "a summation in a period of $1 \sim \frac{k+2}{k}$ sliding window".

Sliding Window Variant - MODEL DESCRIPTION

In this section, we shall describe our implementation of Sliding On-Off sketch model.

Problem Definition

The Sliding On-Off sketch model is used to answer below questions:

(1) Persistence Estimation: <u>Based on the latest N time slices</u>, return the number of time windows in which item x appears

(2) Finding Persistent Items: <u>Based on the latest N time slices</u>, return all the items x_i whose persistence exceeds a given threshold.

Recall from the previous section that a *StateCounter* is a regular counter with an additional state field. We introduce a *SlidingStateCounter* which is essentially a group of *k StateCounters*, each storing information for $\frac{N}{k-1}$ time slices. The notation of parameters for describing the model and experimental results is given below.

- **Key parameters:**

  t: duration of single time slice

  N: sliding window size (duration of a sliding window is tN)

  d: number of hash functions for PE

  L: the output range of the hash function

  w: number of *StateBuckets* (a mapping from user id to a *SlidingStateCounter*) for FPI

  k: number of past *StateCounters* stored for each *SlidingStateCounter*.

  $threshold$: the threshold to be queried in FPI

- **Other notations:**

  x : a data item

  p_x: actual persistence of the item x

  $\widehat{p_x}$: estimated persistence of the item x

  B: a *SlidingStateCounter.*

  B^{new}: the *StateCounter* in B for the most recent items

  B^{old}: the *StateCounter* in B for the least recent items

Model structure

Compared to initial Persistence Estimation data structure in Figure 3 and the Finding Persistence Item data structure in Figure 4, two main modifications are described below.

- We substitute all *StateCounters* with *SlidingStateCounters*, each storing *k StateCounter*.
- Recall that both PE and FPI data structures are composed of arrays of counters. Similar to [2], we add a scanning pointer which goes through the counters and periodically remove the old values.
- Similar to Count-Min Sketch, both PE and FPI only have overestimation error, so we use the same "sum" aggregation strategy, that is taking the sum of all *k* counters in a SlidingCounter as the counter value.

Operations

Our proposed structure for Persistence Estimation with Sliding Window is shown in Figure 7 and Finding Persistent Items with Sliding Window is shown in Figure 8. The operations follows the On-Off Sketch model introduced in previous section. We shall only highlight the modifications to the changes:

a. Sliding Persistence Estimation (Sl_PE)

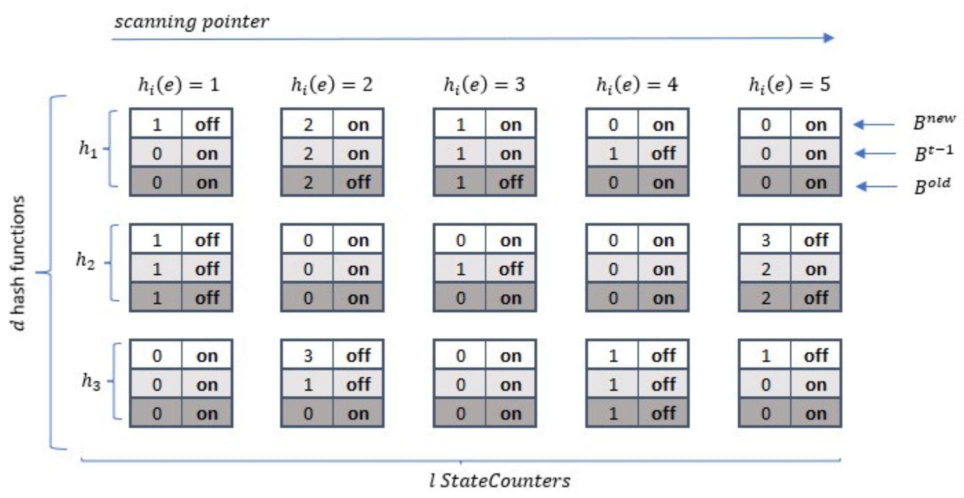


Figure 7. On-Off Sketch Model with Sliding Window for Persistence Estimation

There are four main operations for Sliding Persistence Estimation that follows original PE:

- **sl_pe.init** (x) for data structure initialization
  - every *StateCounter* in every *SlidingStateCounter* is set to 0 and 'On'

- **sl_pe.insert ($\boldsymbol{x}$)** for inserting item
    - the insertion applies to B^{new} only
- **sl_pe.new_slice ()** for reset before every time slice
    - the normal counter reset is applied to every B^{new}
    - for all *SlidingStateCounters* that lies within the sliding window,
        - B^{old} is popped out
        - data in every B^{t} shift to an older *StateCounter* B^{t-1}
        - B^{new} is reset to value 0 and state 'On'
- **sl_pe.query ($\boldsymbol{x}$)** to query for an item's estimated persistence
    - the item x is passed to all hash functions to identify which *SlidingStateCounters* are participants for the query
    - the value of a *SlidingStateCounter* is the sum of all *k StateCounter* values
    - the returned estimated persistence of an item x is the minimum value among all the identified *SlidingStateCounters* for the item x

b. Sliding Persistence Estimation (Sl_PE)

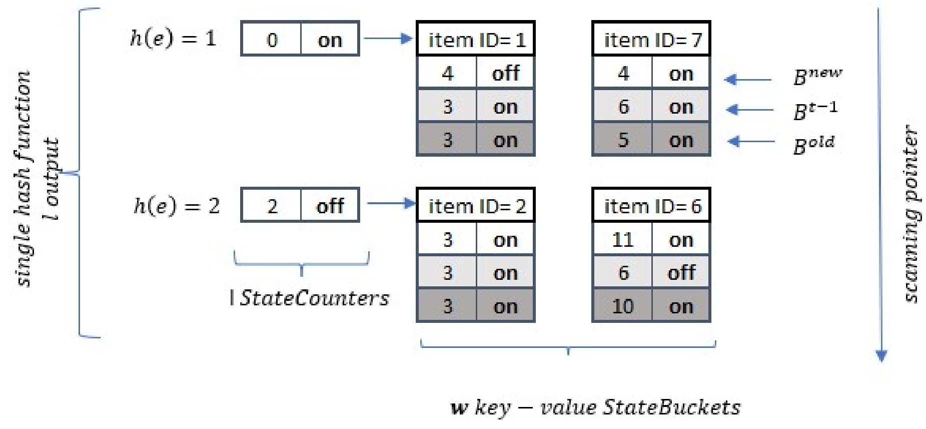


Figure 8. On-Off Sketch Model with Sliding Window for Finding Persistent Item

There are four main operations for Finding Persistence Item with sliding window:

- **sl_fpi.init ($\boldsymbol{x}$)** : for data structure initialization
    - every *StateCounter* in every *SlidingStateCounter* is set to 0 and 'On'
- **sl_fpi.insert ($\boldsymbol{x}$)** : for inserting item
    - similarly, the insertion applies to B^{new} only

- **sl_fpi.new_slice ()** : for reset before every time slice
    - the normal counter reset is applied to every B^{new}
    - for all *SlidingStateCounters* that lies within the sliding window,
        - B^{old} is popped out
        - data in every B^{t} shift to an older bucket B^{t-1}
        - B^{new} is reset to value 0 and state 'On'
- **sl_fpi.query ($\boldsymbol{threshold}$):** return all the items' id, whose estimated persistence is greater than the $threshold$.
    - every *StateBucket* is queried for estimated persistence, which is defined as the sum of the *k StateCounter* values in its *SlidingStateCounter.*
    - if an item's estimated persistence is greater than the $threshold$, the item id is returned.

Model Analysis

Similar to the adaptation made to Count-Min Sketch with Sliding Window technique [2], we used the "sum" strategy to aggregate the values in *a SlidingStateCounter.* Hence, the model inherits the errors from the On-Off Sketch model and Sliding Sketch model, both of which are overestimation to the persistence. We note that the overestimation from Sliding Sketch model is at most $\frac{N}{k-1}$ time slices worth of information, since each *StateCounter* stores information for $\frac{1}{k-1}$ of a sliding window (in other words k-1 *StateCounter* stores the data for a sliding window). This approach is very good for gaining persistence information in the last N time-window compared to the original approach since $\frac{1}{k-1} < 1$ and a very rough analysis would give us an upper bound of a 2x time window estimation.

EXPERIMENTAL RESULTS

Setup

- **Dataset:**

  We selected open-source Network dataset [3] to evaluate our sketch. The dataset contains the users' commenting activities on the Stack Overflow website between 2008-08-01 05:17:57 to 2016-03-06 14:10:28, a span of 2774 days (~7.6 years). Each entry is of the form (user A, user B, timespan t), indicating that at a time of timespan t, user A responded to user B's post. As our problem is concerned with the persistence of a user commenting activities, the dataset is pre-processed to remove user B's id. The dataset is a 63,497,050-line plain text file of 0.93GB in size with 2,226,243 unique user A's id.

- **Environment: Python, SoC clusters**

  The original experiments for On-Off-sketch and k-hash sliding window model were written in C++ on a machine with 6-core processors.

  Based on the algorithms presented in the two papers, we implemented both the On-Off Sketch and Sliding Sketches models in Python and run the experiments on SoC's xgpc and xgpd clusters. The machines have 32-core Intel Xeon Silver 4108 CPU @ 1.8 GHz with 512KiB (L1i), 512 KiB(L1d), 16MiB (L2), 22MiB (L3) cache.

- **Source code**: The source code for our experiments can be found at: https://github.com/benclmnt/nus-cs5234-miniproject

Metrics:

We selected the same metrics as in the On-Of Sketch [1] experiment with mean absolute error (MAE) for Sliding Persistence Estimation (Sl_PE); The mean absolute error (MAE), false negative rate (FNR) and false positive rate (FPR) for Finding Persistent Items (Sl_FPI):

(1) Mean absolute error (MAE): the mean absolute error is the sum of differences between real persistence and estimated persistence divided by the total number of distinct items.

$$MAE = \frac{\sum_{i=1}^{n} |p_x - \widehat{p_x}|}{n}$$

(2) False Positive Rate (FPR): The false positive rate for Sl_FPI is the ratio of number of non-persistent items reported as persistent, divided by the total number of non-persistent items. It indicates an overestimate of persistence.

$$FPR = \frac{FP}{FP + TN}$$

(3) False Negative Rate (FNR): The false negative rate for Sl_FPI is the total number of persistent items reported as non-persistent, divided by the total number of persistent items. It indicates an under-estimate of persistence.

$$FNR = \frac{FN}{FN + TP}$$

Parameters

Throughout the experiment, the following parameters are fixed.

1. L, the range of the hash function's output is set to 10,000
2. We use Bob Jenkins' spookyhash v2 with different seeds to generate our hash functions. This choice is made to get a fast-enough hash function that can easily be used to generate pairwise-independent hash functions. Note that this is different from the original paper's choice of hash function (Bob Jenkins' EvaHash [4]).
3. The time range is divided into 1600 time slices, so each time slice is about 1.75 days. The number follows the original On-Off Sketch paper [1].
4. Each sliding window is set to contain 20 time slices.

Evaluation

Persistence Estimation with Sliding Window

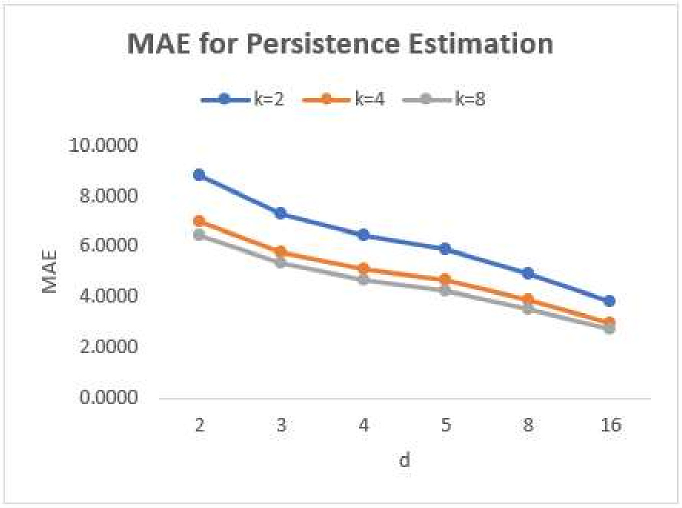


Figure 9. MAE results for Sliding On-Off Sketch Model for Persistence Estimation

Figure.9 shows the MAE result for persistence estimation with Sliding Window as the number of hash functions *d* and number of bucket fields *k* used for storing historical window data vary. Our results show that k = 8 has about 1.37 times and 1.26 times lower error compared to k = 2 and k = 4, respectively. It also shows that d = 16 has about 1.5 times lower error compared to d = 2.

The experiment confirms our theoretical expectation (following Count-Min Sketch argument) that for all *k*, the error decreases as *d* increases. We also note that the error decreases when *k* increases, which satisfies our intuition that more granular buckets can reduce our overestimation error.

It is also interesting to note the relation between *d* and *k* for a fixed amount of memory. The memory usage scales linearly with both *d* and *k*. For a fixed amount of memory, we noticed that the error is roughly the same, with an increase in *d* playing a slightly more significant effect compared to an increase in *k*. The table shown below is taken from our raw experiment data and is an example to the conclusion. Our recommendation will be "If memory is limited, prioritize increasing *d* over *k*".

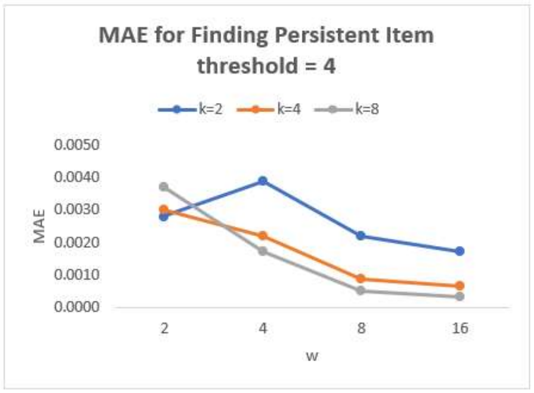
d	***k***	**PE_MAE**
2	8	6.41
4	4	5.11
8	2	4.9

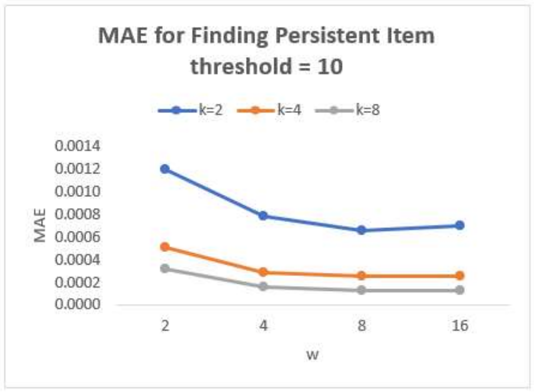
Another interesting thing to note is the improvement from k = 2 to k = 4 is much more significant than from k = 4 to k = 8. This phenomenon can be explained as follows. The overestimation for *k*=*k* is 1/(*k*-1) sliding window worth of data. Thus, increasing *k* from 2 to 4 reduces the overestimation by 1/1 – 1/3 = 2/3 sliding window compared to from 4 to 8 which reduces the overestimation by only 1/3 – 1/7 = 4/21 sliding window.

Finding Persistent Items with Sliding Window

To analyze the performance of the algorithm to find persistent item, we vary two parameters: (1) *w*: the number of key-value pairs stored per counter, and (2) *k*: number of bucket fields used to store historical window data. We run the experiment twice, once with a low threshold — an item is "persistent" if it appears in at least 20% of the past 20 time slices, resulting in more "persistent" items — and another with a high threshold — an item is considered "persistent" if it appears in at least half of the past 20 time slices, resulting in less "persistent" items.

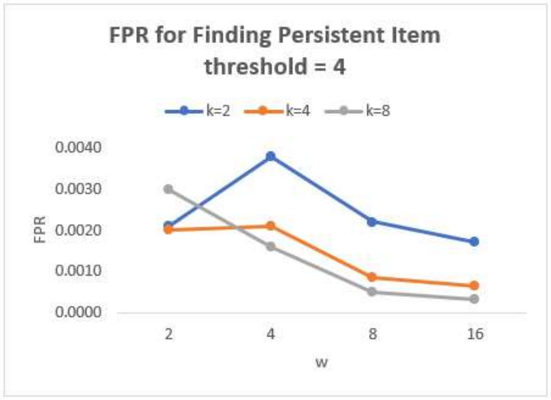
The MAE (Figure 10), FNR(Figure 11), and FPR (Figure 12) of both runs are shown below.

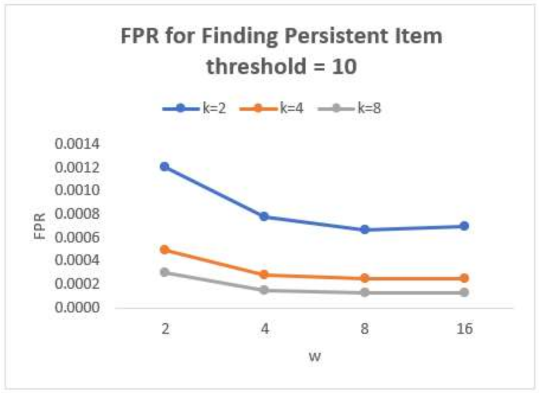




(a) **(b)**

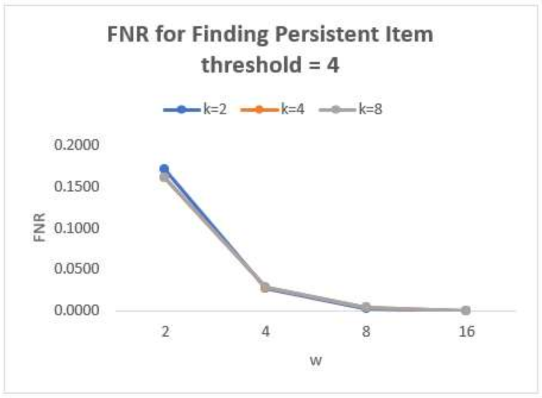
Figure 10. MAE results for Sliding On-Off Sketch Model for Finding Persistent Item

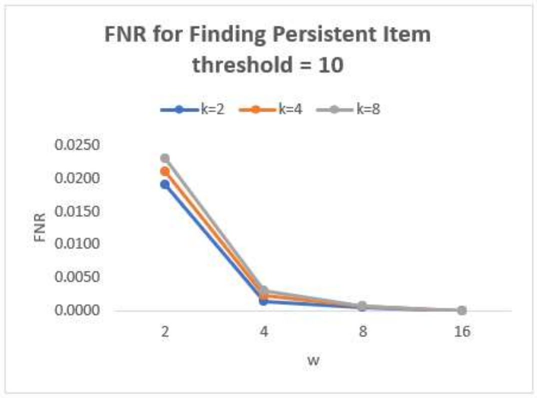




(a) **(b)**

Figure 11. FPR for Finding Persistent Item using Sliding On-Off Sketch Model





(a) **(b)**

Figure 12. FNR for Finding Persistent Item using Sliding On-Off Sketch Model

In general, we have similar results to the sliding PE. As *k* increases (and hence memory usage increases), MAE, FPR and FNR generally decreases. Our results show that k = 8 has 2-5 times lower error compared to k = 2. Similarly, as *w* increases, MAE, FPR and FNR generally decreases. Similar to the analysis made in the PE case above, this experiment confirms the theoretical expectation.

An interesting result happens when w = 2 and threshold = 4. We saw that k = 2 and k = 4 outperforms k = 8 for both MAE and FPR. This could be because when w = 2, and threshold = 4, there is many persistent items (as the threshold is low) but there is not enough *StateBuckets* to uniquely identify all of these persistent items. As a result, the *StateCounter* is incremented much quicker. When k = 8, this causes the number of non-persistent counter reported as persistent increases and hence the FPR is much higher.

The FNR is low for w > 2 because the sketch only has overestimation error. A persistent item will most likely end up in one of the *StateBucket* (it will not be in the *StateBucket* only if enough persistent item maps to the same hash value) and hence with large enough w, there is almost no persistent items that is reported as non-persistent.

CONCLUSION

Persistence is a rarely explored property of streaming data. However, it still has its importance in many areas of applications. On-Off Sketch is a simple yet brilliant way to store persistence information. As an extension, we learned and applied the Sliding Sketch technique to maintain only the most recent persistence information. Experimentally, it proves to be feasible and have a promising lower error rate. Further work could be done to explore the model accuracy with more types of data.

REFERENCES

[1] Yinda Zhang, Jinyang Li, Yutian Lei, Tong Yang, Zhetao Li and Gong Zhang. On-Off Sketch: A Fast and Accurate Sketch on Persistence. In *47th International Conference on Very Large Data Bases, Copenhagen, Denmark, August 16-20, 2021*, vol14, pages 128-140.

[2] Xiangyang Gou, Long He, Yinda Zhang, KeWang, Xilai Liu, Tong Yang, YiWang, and Bin Cui. Sliding sketches: A framework using time zones for data stream processing in sliding windows. In *KDD '20: The 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Virtual Event, CA, USA, August 23-27, 2020*, pages 1015–1025. ACM, 2020.

[3] The Network dataset Internet Traces. http://snap.stanford.edu/data/.

[4] Bob Jenkins' EvaHash. http://burtleburtle.net/bob/hash/evahash.html.