Recency Queries with Succinct Representation

William L. Holland
School of Computing and Information Systems, The University of Melbourne, Parkville, Australia
w.holland@student.unimelb.edu.au

Anthony Wirth
School of Computing and Information Systems, The University of Melbourne, Parkville, Australia
awirth@unimelb.edu.au

Justin Zobel
School of Computing and Information Systems, The University of Melbourne, Parkville, Australia
jzobel@unimelb.edu.au

Abstract

In the context of the sliding-window set membership problem, and caching policies that require knowledge of item recency, we formalize the problem of Recency on a stream. Informally, the query asks, "when was the last time I saw item x?" Existing structures, such as hash tables, can support a recency query by augmenting item occurrences with timestamps. To support recency queries on a window of W items, this might require $\Theta(W \log W)$ bits.

We propose a succinct data structure for Recency. By combining sliding-window dictionaries in a hierarchical structure, and careful design of the underlying hash tables, we achieve a data structure that returns a $1+\varepsilon$ approximation to the recency of every item in $O(\log(\varepsilon W))$ time, in only $(1+o(1))(1+\varepsilon)(\mathcal{B}+W\log(\varepsilon^{-1}))$ bits. Here, $\mathcal{B}$ is the information-theoretic lower bound on the number of bits for a set of size W, in a universe of cardinality N.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis

Keywords and phrases Succinct Data Structures, Data Streams, Sliding Dictionary

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2020.49

Funding This work was supported by the Australian Research Council, grant number DP190102078, and an Australian Government RTP Scholarship.

Acknowledgements We acknowledge the Wurundjeri People of the Kulin Nations as traditional owners of the land on which we live and work.

1 Introduction

In evolving data streams, applications are often interested in *recent* history. This is evident in recency-sensitive applications such as in-network caching [5], web-crawling and the detection of duplicates [7]. Data structures that provide summaries of item-histories in support of these applications can be found in the *sliding membership*[1] literature [2, 6, 8, 11]. The summaries are set membership structures fused with estimates of item recency; a statistic identical to the question "when was the last time I saw item x?". Item recency can be supported by a hash table by augmenting item occurrences with timestamps or some identifier of a point in history. However, this strategy has not been fulfilled in both "small" space and with sufficient accuracy. Thus, we look at the problem of balancing these demands in the design of a data structure that supports a recency query.

What makes the recency and the sliding membership problems compelling, particularly in the context of the broader dictionary literature, is *decay*: the data structure must forget old items. Thus, a mechanism is required to determine the age of an item and, in turn,

[1] The sliding membership problem asks: "has item x occurred among the last W items?"

identify candidates for omission. If we want to keep the representation of the set of items *succinct* (a representation of a set that occupies an amount of space close to the theoretic information lower bound), the question of *how* to determine item recency is non-trivial. Additional information must be associated with each item. So we may ask whether the size of this information is dependent on the number of items present and, in turn, whether this precludes a succinct representation. In this paper, we present a space-efficient data structure to support recency queries with relative accuracy, maintaining succinct representations of the occurred items.

1.1 Formalizing the problem

In the sliding membership problem, on parameters $D, W \in \mathbb{Z}^+$, a process observes a sequence of items $S(t) = \langle s_1, s_2, \ldots, s_t \rangle$, from a universe $[N]$, we seek to answer membership queries on item x in the length-W window of *most recent* items, $S_W(t) = \langle s_{t-W+1}, \ldots, s_t \rangle$:

$$x \stackrel{?}{\in} S_W(t) = \begin{cases} \mathsf{yes} & \text{if } x \in S_W(t), \\ \mathsf{no} & \text{if } x \notin S_{W+D}(t), \\ \mathsf{yes} \text{ or } \mathsf{no} & \text{otherwise.} \end{cases}$$

Including the *slack* parameter D allows for more efficient solutions [8].

Recency

A recency query, $r(x,t)$, takes sliding membership a step further, and returns a measure of the age of an item x:

$$r(x,t) = t - \max\{j \mid s_j = x, j \leq t\}\,.$$

Trivially, sliding membership reduces to recency, as $(r(x,t) \in [0,W))$ answers $x \stackrel{?}{\in} S_W(t)$.

The commonly cited naïve solution for sliding membership, with slack parameter $D > 0$, entails dividing the window into blocks of width D. Each block is stored in a *static* dictionary. To evaluate item membership on the sliding window it suffices to query each block, in turn, and return the logical disjunction of the results. Similarly, the recency of an item x, given that $x \in S_W(t)$, can be approximated by returning the recency of the (youngest) block it belongs to. This approach returns approximations with *absolute* error at most D. However, when viewed from the perspective of *relative* error, estimates are less accurate for items with low recency. Accurate estimates may be required throughout the window and are arguably more valuable for more recent items, motivating our formalization of the Recency problem.

▶ **Problem 1** (Recency). *Given $W, D \in \mathbb{Z}^+$ and $\varepsilon \in (0,1)$, and the sequence $S(t) = \langle s_1, s_2, \ldots s_t \rangle$ from universe $[N]$, when presented with some item $x \in [N]$, return an estimate $\hat{r}$ for $r(x,t)$ where*

$$\hat{r} \in \begin{cases} (1 \pm \varepsilon) r(x,t), & \text{if } x \in S_W(t)\,, \\ \{-1\}\,, & \text{if } x \notin S_{W+D}(t)\,, \\ (1 \pm \varepsilon) r(x,t) \cup \{-1\}\,, & \text{otherwise.} \end{cases}$$

To set the context of our contributions, we briefly consider approaches to Recency that are nearly immediately at hand. The solutions in the sliding membership literature take a general form: insert (item-signature,[2] timestamp) pairs into a hash table. This approach either incurs a large memory overhead [6] or does not admit bounded relative error [8]. As an alternative approach, one could store items in a circular array of length $\mathcal{O}(W)$. However, queries are linear in the length of the array. This cost could be reduced to the cardinality of the window with a move-to-front list [10]. Both approaches are non-succinct in memory.

1.2 Contribution

We introduce a data structure named (HistoricalMembership) that achieves *tight* memory allocation and bounded relative error in return for logarithmic update and query times. Our solution builds on existing approaches, particularly the tactic of dividing the window into blocks of items of equivalent age. However, to achieve both relative $(1 \pm \varepsilon)$ accuracy in item recency and also space efficiency, the structure is hierarchical, comprising levels of geometrically increasing size. Level l of HistoricalMembership is a sliding dictionary with a window of $\varepsilon^{-1}2^l$ items and slack of 2^l, divided into blocks of size 2^l.

We illustrate the formal validity of our approach and conjecture whether improvements are possible. Our main result is captured in the following theorem.

▶ **Theorem 2.** *On a sequence of $S = \langle s_1, s_2, \ldots \rangle$, where $s_i \in [N]$, for parameters W and $\varepsilon > 0$, at each timestamp $t \geq 1$, HistoricalMembership solves the Recency problem in*

$$(1 + o(1))(1 + \varepsilon)(\mathcal{B} + W \log(\varepsilon^{-1}))$$

bits of memory, admitting query and update times of item x in $\mathcal{O}(\log(\varepsilon \cdot r(x, t)))$. This bound is with high probability worst case for queries and expected amortized for updates. Here, $\mathcal{B}$ is the information-theoretic lower bound for storing a subset of size W from the universe $[N]$.

Importantly, HistoricalMembership achieves ε-approximation to item recency in space asymptotically identical to the state-of-the-art data structure for sliding membership [8]. The auxiliary information is not dependent on W and the representation of the items, including the slack, is succinct.

2 Background

Succinct data structures

We adhere to the word RAM model of computation with word size $w = \Theta(\log N)$. Elements from the universe N can be stored in $\mathcal{O}(1)$ machine words and bitwise operations such as arithmetic require constant time. There exist $\binom{N}{m}$ distinct size-m subsets of the universe $[N]$ so an encoding of such a subset requires, on average, at least

$$B(N, m) = \log \left\lceil \binom{N}{m} \right\rceil = m \log \frac{N}{m} + \mathcal{O}(m) \tag{1}$$

bits. If the information theoretic memory lower bound for a data structure that supports a *particular* query is $\mathcal{B}$ bits, then a *succinct* data structure is one that requires $(1 + o(1))\mathcal{B}$ bits. We refer to a data structure that solves (sliding) set membership as a (sliding) *dictionary*.

[2] The type of item-signature depends on the context. For *exact* set membership the key $x \in [N]$ is sufficient. For approximate set membership, where collisions are allowed, the hash value $h(x)$ may be used to identify the item. Many solutions to approximate set membership are simply reductions to exact set membership via a hash function that reduces the key space.

Table 1 Comparison of HistoricalMembership with existing art. Term $\mathcal{B}$ denotes the information-theoretic lower bound for storing a set of W items from the universe $[N]$. ExactCuckoo solves exact Recency. OptimalSM solves approximate Recency with bounded absolute error $\leq D-1$, with $D \leq W$. HistoricalMembership solves approximate Recency with bound relative error.

	Update time	Query time	Space
ExactCuckoo [6]	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(W(\log N + \log W))$
OptimalSM [8]	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$(1+o(1))(1+D/W)(\mathcal{B} + W\log(W/D))$
HistoricalMembership	$\mathcal{O}(\log(\varepsilon W)))$	$\mathcal{O}(\log(\varepsilon W))$	$(1+o(1))(1+\varepsilon)(\mathcal{B} + W\log(\varepsilon^{-1}))$

BackyardCuckoo

The foremost primitive in HistoricalMembership is the backyard cuckoo hash table (BackyardCuckoo) [1], a two-level variant of cuckoo hashing [9] that achieves improved memory utilization[3] and (with high probability) worst-case constant update and query times. The attributes of BackyardCuckoo are summarized in the following theorem.

▶ **Theorem 3** ([1]). *A dynamic set of items, drawn from the universe $[N]$, of size at most m, can be stored in $(1+o(1))B(N,m)$ bits. With probability at least $1-1/poly(m)$, insert, delete and query are performed in worst-case constant time.*

In addition, backyard cuckoo hashing allows auxiliary information to be stored with each item. When a query is lodged, via an item-signature, the auxiliary information is (also) returned. With this extension, for auxiliary information of at most K bits, the hash table uses $(1+o(1))(B(N,m)+mK)$ bits of memory.

Sliding approximate membership

Sliding dictionaries are an established class of data structure and some of them (indirectly) solve the recency problem. We seek sliding dictionaries that can return their internal measure of Recency. One class of solutions fails to do this [2, 7]. However, the solutions based on hash tables [5, 8], which store items with an amount of auxiliary information indicative of the age of the item, do provide recency estimates and are a starting point in our inquiry.

As a baseline for the *exact* Recency problem, Liu et al. [6] propose storing (item-signature, timestamp) pairs in a cuckoo hash table. *Expired* items are identified as those with timestamps that sit outside the window. For time efficiency, the deletion of expired items is performed *lazily*. This is done in one of two ways. First, if an expired item is in encountered during an insert or query, it is deleted. Second, at the completion of an insertion, a constant number of cells are scanned and expired items are deleted. The process records the finishing point of the scan and resumes at this location during the next update. These measures ensure that an expired item is deleted within a time frame of W updates and to save space, timestamps are assigned modulo $2W$. Insertions take expected amortized constant time, while queries are worst-case constant as per cuckoo hashing theory [9]. Memory utilization is almost $1/2$, so the table contains a large proportion of empty cells. In total, the table requires $\mathcal{O}(W(A+\log W))$ bits, where A is the size of the item-signature.[4] Notably, as timestamps are stored, the data structure solves the exact Recency problem. We refer to this solution as ExactCuckoo.

[3] Utilization arbitrarily close to 1, as opposed to arbitrarily close to 1/2 for standard cuckoo hashing.
[4] $\lceil \log 1/\delta \rceil$ for approximate set membership and $\lceil \log N \rceil$ for exact set membership.

The theoretical state-of-the-art solution for *absolute-error* Recency is by Naor and Yogev [8]. It is the only approach that accounts for slack, and moreover measures its benefit. We refer to the data structure as Optimal Sliding Membership (OptimalSM). The solution entails partitioning the window into blocks of size D. Then (item-signature, block-number) pairs are stored in a hash table. As above, block IDs are assigned on a circular field and evictions are lazy. With this approach, and by introducing slack, OptimalSM reduces the cost of the timestamp compared with ExactCuckoo.

Blocks that overlap with the window are called *active* and those that sit outside the window as *expired*. During an insertion, the procedure assigns the item to the youngest active block, which we call the *contemporary* block. At each timestamp, there are $W/D + 1$ active blocks and the circular field of block IDs is modulo $2(W/D + 1)$. The hash table is BackyardCuckoo and a succinct representation of the items is obtained. A key contribution is a lower bound on the sliding approximate membership problem.

▶ **Theorem 4** ([8]). *For parameters $W, D \in \mathbb{Z}^+$ and failure probability $\delta \in (0, 1)$, a data structure that returns approximate set membership queries on the length-W sliding window, with slack D, requires at least the following number of bits*

$$W \log \tfrac{1}{\delta} + W \cdot \max\{\log \tfrac{W}{D}, \log\log \tfrac{1}{\delta}\} - \mathcal{O}(W)\,.$$

Naor and Yogev's construction is tight up to the first two terms, and OptimalSM solves Recency with bounded *absolute* error.

The ExactCuckoo and OptimalSM share this approach: store an item-signature with a time indicator, such as a timestamp or block ID in a hash table. In other words, we can think of the hash table as a black box, and so we collectively refer to these two approaches as Hash Sliding Membership (HashSM). HistoricalMembership employs multiple instances of HashSM to construct a Recency data structure with bounded relative error. A comparison between ExactCuckoo, OptimalSM and HistoricalMembership is available in Table 1.

3 HistoricalMembership

With ExactCuckoo, the sliding membership literature provides a solution for exact Recency. However, the structure does not suggest an approximate solution, nor how to trade (relative) accuracy for space, or rather, to reduce the $\mathcal{O}(\log W)$ bit allocation for timestamps. The approach of Noar and Yogev [8], which involves dividing the window into blocks of fixed length, provides an approximation of bounded absolute error. This approach is sufficient for identifying old items, with large recency values, and is therefore ideal for sliding membership. Through the notion of a recency *equivalence class*, we refine their approach towards a structure that is sensitive to relative error.

3.1 Equivalence classes

Following OptimalSM, suppose we partition the window into blocks, and assign each item a block ID. The most recent item in the fourth block has a recency value of $3D$ and we can approximate the recency of all items in the block by assigning them the same value of $3D$. As the oldest item in the block has actual recency $3D + D - 1$, the absolute error of the estimate is at most $D - 1$ and hence incurs relative error approximately $1/4$. To provide a recency value with (parameterized) bounded relative error, pertinent for items with low recency values, HistoricalMembership partitions the window into a sequence of blocks of non-decreasing size. Every item in a block is deemed to have recency *equivalent* to the

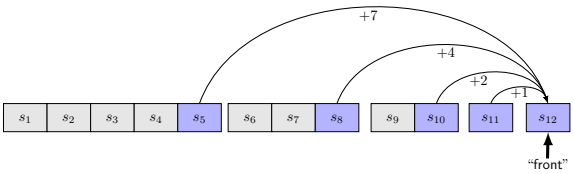

Figure 1 A sequence $\langle s_1, \ldots, s_{12} \rangle$ is partitioned into blocks of *non-decreasing* size. The timestamp of the "front" item of each block becomes an implicit timestamp for all items in the block. For example, the token s_1 is assigned a recency estimate of 7. Since its actual recency is 11, it incurs a relative error of $({}^{11}/_{7} - 1)$. Storing each item in the set $S = \{s_1, \ldots, s_{12}\}$ with its corresponding block number gives a *static* solution to approximate recency.

recency of the most recent item (the *front* item) in that block. An example is shown in Figure 1. As a byproduct, we lose order and granularity within the blocks themselves, with each block now a homogeneous zone of recency with a single *representative* timestamp. In other words, HistoricalMembership maintains a *partial order* of the underlying sequence. We want the front item to be a *good representative* for the block. We hence refer to a block that is unified by a $(1 + \varepsilon)$-approximation as an *equivalence class*. To make this notion more rigorous, a pair of timestamps, (t_a, t_b), satisfying the two conditions

$$t_a < t_b \quad \text{and} \quad (t - t_a) \leq (1 + \varepsilon)(t - t_b) \tag{2}$$

defines an equivalence class. The difference, $t_b - t_a$, is the *width* of the equivalence class.

As an equivalence class demarcates a contiguous neighbourhood of items from the sequence, the approach of HistoricalMembership is to *dynamically* re-organize the sequence into an appropriate collection of equivalence classes, as each item arrives in the stream. This constitutes a high-level view of HistoricalMembership, but the efficiency of the structure depends on how the classes are stored and accessed.

3.2 Coordinating the equivalence classes

To perform efficient maintenance of the dynamic collection of equivalence classes, we propose merging adjacent classes. This tactic follows from the observation that older classes can be wider. As a class ages, and moves further away from the present, it implicitly becomes more (relatively) accurate. Suppose the two intervals $[t_a, t_b)$ and $[t_b, t_c)$ represent blocks within the division of the window. If the pair (t_a, t_c) satisfies the conditions (2), the structure has permission to merge the blocks.

Invariant

Accordingly, within the framework of merge-type equivalence class maintenance, HistoricalMembership organizes the classes into $L = \lfloor \log(\varepsilon W) \rfloor - 1$ levels. At level l, the width of each equivalence class is 2^l. Therefore, merging two clases at level l, of width 2^l, creates a new class of width 2^{l+1} and a resident of level $(l + 1)$. To maintain the equivalence class constraints of relation (2), we insist that at least ε^{-1} and at most $\varepsilon^{-1} + 1$ classes at reside at each level. This ensures that each item at level l has recency at least

$$\varepsilon^{-1} \sum_{i=0}^{l-1} 2^i = \varepsilon^{-1}(2^l - 1)\,,$$

which, in turn, justifies the width of the classes (refer to Lemma 5 below). Consequently, we require that ε^{-1} is an integer. Reducing the ε term in the approximation factor increases the number of classes at each level. Observe that $\sum_{l=0}^{L} \varepsilon^{-1} 2^l \leq W$; we refer to the difference

$$E = W - \sum_{l=0}^{L} \varepsilon^{-1} 2^l \tag{3}$$

as the *excess*. We can either extend level L to include the excess or create a new level for it. HistoricalMembership opts for the former approach, as it is more space effective to extend the top level than to create a new level that operates below its conceptual cardinality.

Updates

As each item arrives in the stream, it is prepended to level 0, whose items are stored in "equivalence classes" of width 1. When level 0 becomes full, which is to say it contains $(2 + \varepsilon^{-1})$ items, the two *oldest* items become an equivalence class of width 2, which is promoted to level 1. Similarly, when level l acquires $2+\varepsilon^{-1}$ equivalence classes, its two oldest equivalence classes are merged, and become the newest equivalence class at level $l+1$. At the top level, when an equivalence class "falls off the window", it is (conceptually) deleted.

4 Upper bound: level = sliding dictionary

The preceding section presents an overview of the structure of HistoricalMembership. We now turn to the question of how to store and maintain this equivalence class partition of the window. An initial temptation would be to store each equivalence class as a (succinct) static dictionary. Periodically, the dictionary structures can be merged, and techniques are available to do this [3]. However, this leads to expensive worst-case queries, taking $\Omega(\varepsilon^{-1}L)$ time, in which every dictionary in each level is queried, level by level.

Levels as sliding windows

To reduce the number of internal queries to the dictionary primitive, we observe that every level in fact constitutes a window partitioned into blocks of fixed length. Thus, we can engage a HashSM sliding dictionary to store an entire level of HistoricalMembership. The problem, for levels $l \in \{0, 1, \ldots, (L-1)\}$, reduces to sliding membership with window length $W_l = \varepsilon^{-1} 2^l$ and slack $D_l = 2^l$. Following the approach of OptimalSM, it suffices to divide the window into ε^{-1} blocks of width 2^l and store (item-signature, block ID) pairs in a hash table. At level L, extended to contain the excess of the level structure, reduced to sliding membership with window length $W_L = E + \varepsilon^{-1} 2^L$ and slack $D_L = 2^L$, where E is defined in relation (3). Block IDs can be assigned in a circular fashion, and, in our context, are identical to an equivalence class ID. Items that belong to an expired equivalence class are promoted lazily; the item may sit expired at a level l, but it is understood to conceptually belong at level $l+1$. item-signatures must be stored under a representation that is *invertible*, as is the case for BackyardCuckoo hashing.

Insertion operation

An insertion into a level proceeds according to the logic of HashSM. Following a protocol of lazy promotions, if, in the process of allocating a free cell to an (item-signature, equivalence class) pair,[5] the procedure encounters an item with a lapsed equivalence class, that is, a class

[5] Hash tables such as Cuckoo and BackyardCuckoo offer multiple choices for each item allocation.

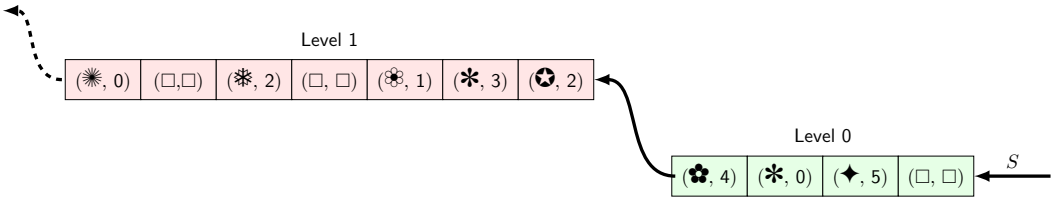

Figure 2 Image of levels 0 & 1 of HistoricalMembership, for $\varepsilon^{-1} = 2$ and $L \geq 2$, on the sequence $S = \langle \ldots, ✺, ❀, ✻, ✪, ❄, ✻, ✿, ✦, ✻ \rangle$. The global clock is at $t = 18$. Each level is a hash table that stores (item-signature, equivalence class ID) pairs and implements a sliding dictionary. Each level has $\varepsilon^{-1} + 1 = 3$ equivalence classes. The circular field of block IDs is $2(\varepsilon^{-1} + 1) = 6$. The contemporary block ID at level 0 is 0 and the contemporary block ID at level 1 is 3. The equivalence class width at level 1 is $2^1 = 2$. Note, ✻ occurs twice at level 1 and does not appear in block 1. The item ✺ has expired at level 1 and awaits (a lazy) promotion to level 2.

that has been conceptually merged and placed in the succeeding level, it evicts the expired item and inserts it into the subsequent level. After the insertion pair has been allocated a free cell, the process scans a constant number of cells and promotes expired items. For levels $l \in [L-1]$ block IDs come from the range $\{0, 1, 2, \ldots, 2(\varepsilon^{-1}+1)\}$, treated as a circular field. For level L, block IDs come from

$$2(W_L/2^L + 1) = 2(\varepsilon^{-1} + E/2^L + 1)\,. \tag{4}$$

In the HashSM approach of lazy deletions there is no control over when an expired item is evicted from a level. Rather, this is an outcome of the randomness that distributes items within the hash table. Therefore, care needs to be taken, during a promotion, when inserting into the succeeding level such that the item is placed in the correct equivalence class. To accomplish this, we make modify the approach of HashSM, specifically, installing an *external* assignment of block IDs.

Block ID assignment

In HashSM, during an insertion, the procedure assigns the item to the "contemporary" block. With the combination of the levelled structure of HistoricalMembership and lazy promotions, an item evicted from level l may not belong to the contemporary equivalence class (block) of level $l+1$. Therefore, equivalence class IDs are assigned *externally*. For example, if, at level l, an evicted item returns the (expired) equivalence class ID e from the circular field $2(\varepsilon^{-1}+1)$, the corresponding class ID in level $l+1$ can be calculated as follows. Letting e_j name the contemporary class ID at level j, the number of equivalence classes *between* class e and the local (level-l) window, assuming $l+1 \neq L$, is

$$d = e_l - e - (\varepsilon^{-1}+1) \bmod (2\varepsilon^{-1}+2).$$

The difference is equal to the contemporary class ID minus the expired ID minus the number of active classes modulo the circular field. The difference d determines the equivalence class ID e^* that e would be merged into in level $l+1$. Recall that adjacent equivalence classes are (conceptually) *merged* prior to a promotion to a succeeding level. Therefore, the number of classes between e^* and e_{l+1} is $d/2$, and hence $e^* = e_{l+1} + d/2 \bmod (2\varepsilon^{-1}+2)$.

Each sliding dictionary is synchronized by a global clock, thus, the contemporary class IDs are also updated externally. For this reason, if there are no items to insert at a level, HistoricalMembership still scans for evictions as if an item were being inserted. A formal

summary of the insertion procedure is available in Algorithm 2. The initialization algorithm is present in Algorithm 1. Together, they instruct the *external* assignment of block IDs for the sliding dictionaries. An image of HistoricalMembership is in Figure 2.

Correctness/queries

To evaluate the recency of an item x, HistoricalMembership probes the levels sequentially. Thus, assuming x is in the window, query time is proportional to the logarithm of x's recency. If x is (first) retrieved from level l, its recency is estimated from its equivalence class ID, e. The level l dictionary provides a local estimate $\hat{r}_l(x)$ with absolute error at most $2^l - 1$:

$$\hat{r}_l(x) = 2^l \cdot (e - e_l \bmod (2\varepsilon^{-1} + 2)) .$$

The local estimate can be interpreted as the recency of the item with respect to level l. To construct a global estimate $\hat{r}(x,t)$, with bounded relative error, we accumulate the widths of levels below level l and append the sum to the local estimate. Due to the effect of slack, the width of each level varies. (The width of a level is the bound on the number of active items. Each level has ε^{-1} non-contemporary active classes.) At time t in the stream, the contemporary class at level i has at most $(t \bmod 2^i)$ items, so the width of level i is $(t \bmod 2^i) + \varepsilon^{-1}2^i$. Summing the widths of the preceding levels, we arrive at an estimate.

$$\hat{r}(x,t) = \hat{r}_l(x) + \sum_{i=0}^{l-1} (\varepsilon^{-1}2^i + (t \bmod 2^i)) . \tag{5}$$

With the query algorithm in place, we can bound the relative error of the data structure.

▶ **Lemma 5.** *HistoricalMembership returns a $(1+\varepsilon)$-approximation to item recency.*

Proof. The second summand can be interpreted as the distance between level l and the front of the sequence. The distance is exact. Thus, the error on the global estimate is bound by the error on the local estimate.

$$\frac{|r(x,t) - \hat{r}(x,t)|}{\hat{r}(x,t)} \leq \frac{2^l - 1}{\hat{r}_l(x) + \sum_{i=0}^{l-1}(\varepsilon^{-1}2^i + (t \bmod 2^i))} \leq \frac{2^l - 1}{\varepsilon^{-1}\sum_{i=0}^{l-1} 2^i} = \varepsilon . \qquad \blacktriangleleft$$

5 Efficiency: selecting the hash tables

The efficiency of HistoricalMembership hinges on the performance of the hash tables supporting the HashSM sliding dictionaries. BackyardCuckoo is the state-of-the-art, combining succinct representation with worst-case constant-time operations, *except* with probability proportional to $1/\text{poly}(s)$, where s is the size of the set. in the size of the underlying set. Unfortunately, this is an issue for levels with low cardinality, where the failure probability is non-negligible, particularly across long sequences. Thus our choice tables for implementing HashSM depends on the size of the level. We split the level structure in half and only assign the BackyardCuckoo table to the upper $\lceil L/2 \rceil$ levels. As the lower $\lceil L/2 \rceil - 1$ levels have an aggregated cardinality of $\mathcal{O}((\varepsilon^{-1}+1)W^{1/2})$, there is more flexibility in the hash table construction, in the sense that it does not need to be succinct. A number of options are available and we suggest the dynamic hash table of Dietzfelbinger et al. [4] (DynamicTable). The latter result allows a dynamic set of at most m items to be stored in $\mathcal{O}(m)$ words with constant worst-case query times and constant expected amortized insert and delete.

▶ **Lemma 6.** *In HistoricalMembership, a query can be evaluated in worst-case $\mathcal{O}(\log(\varepsilon W))$ time and insertions completed in expected amortized $\mathcal{O}(\log(\varepsilon W))$ time with probability $1-\mathcal{O}(1/W)$.*

Proof. A query requires at most L probes to the underlying hash tables. For DynamicTable, queries are worst-case constant. For BackyardCuckoo, when representing a set of size m, queries are non constant with *arbitrarily* small probability $1/\text{poly}(m)$. As the width of each level $l \geq \lceil L/2 \rceil$ is $\Omega(W^{1/2})$, we can initialize the BackyardCuckoo hash tables with sufficient randomness such that failures occur with probability $\mathcal{O}(1/W^2)$. Taking a union bound over the event that each level reports sliding membership in constant time, queries are worst-case $\mathcal{O}(\log(\varepsilon W))$ with probability $1-\mathcal{O}(1/W)$. Similarly, as an insertion causes at most $\mathcal{O}(1)$ item insertions at every level, the same argument follows for the insertion time for HistoricalMembership. However, due to inheritance from the DynamicTable, insertion times are only expected amortized. ◀

Further, as the levels are queried consecutively and in reverse chronological order, we can bound the query time as a function of item recency.

▶ **Lemma 7.** *In HistoricalMembership, the query cost for an item $x \in [N]$, with recency $r(x,t) \leq W$, can be evaluated in worst-case $\mathcal{O}(\log(\varepsilon \cdot r(x,t)))$ time with probability $1-\mathcal{O}(1/W)$.*

To bound the memory allocation, we disaggregate the level structure into three components, and work bottom up: the lower levels, implemented by DynamicTable; the upper-middle levels, implemented by BackyardCuckoo; and the top level, which contains the excess.

▶ **Lemma 8.** *HistoricalMembership stores levels $\{0, 1, \ldots, \lceil L/2 \rceil - 1\}$ in $(1+\varepsilon)\sqrt{W} \cdot \mathcal{O}(w)$ bits.*

Proof. Each lower level l has cardinality at most $(\varepsilon^{-1}+1)2^l$. Accumulated across all lower levels the cardinality is at most $(1+\varepsilon)\sqrt{W}$. The DynamicTable stores m via item-signatures in $\mathcal{O}(mw)$ bits, which in total leads to a bound of $(1+\varepsilon)\sqrt{W} \cdot \mathcal{O}(w)$ bits. ◀

▶ **Lemma 9.** *For levels $l \in \{\lceil L/2 \rceil \ldots, (L-1)\}$, the memory allocation of the sliding dictionaries accumulates to $(1+o(1))(\varepsilon^{-1}+1)(2^L-1)\big(\log\big(\frac{N}{W}\big) + \log \varepsilon^{-1} + \mathcal{O}(1)\big)$ bits.*

Proof. By Theorem 3, BackyardCuckoo stores a set of size m, from the universe $[N]$, with each item containing auxiliary information of K bits, in $(1+o(1))((B,m)+mK)$ bits. As level l has cardinality at most $(\varepsilon^{-1}+1)2^l$ and stores auxiliary information of $\log(2\varepsilon^{-1}+2)$ bits for the equivalence class ID at that level, we can accumulate the memory commitment across the relevant levels. To set this up, we observe that

$$W \leq \varepsilon^{-1} 2^{L+2}\,, \tag{6}$$

and that

$$\sum_{l=\lceil L/2 \rceil}^{L-1} 2^l(L-l) \leq \sum_{l=0}^{L-1} 2^l(L-l) = \sum_{l=0}^{L-1} 2^l + \sum_{l=0}^{L-2} 2^l + \cdots + \sum_{l=0}^{0} 2^l \leq 2^{L+1} - 2\,. \tag{7}$$

Hence

$$\begin{aligned}
&\sum_{l=\lceil L/2\rceil}^{L-1}(1+o(1))(B(N,2^l(\varepsilon^{-1}+1))+2^l(\varepsilon^{-1}+1)\log(2\varepsilon^{-1}+2))\\
&=(1+o(1))\sum_{l=\lceil L/2\rceil}^{L-1}2^l(\varepsilon^{-1}+1)(\log\left(\frac{N}{2^l(\varepsilon^{-1}+1)}\right)+\log\varepsilon^{-1}+\mathcal{O}(1))\\
&\leq(1+o(1))(\varepsilon^{-1}+1)\sum_{l=\lceil L/2\rceil}^{L-1}2^l\big((\log\left(\frac{W}{2^l\varepsilon^{-1}}\right)+\log\left(\frac{N}{W}\right))+\log\varepsilon^{-1}+\mathcal{O}(1)\big)\\
&\leq(1+o(1))(\varepsilon^{-1}+1)\sum_{l=\lceil L/2\rceil}^{L-1}2^l((L-l+2)+\log\left(\frac{N}{W}\right)+\log\varepsilon^{-1}+\mathcal{O}(1))\quad\text{from (6)},\\
&\leq(1+o(1))(\varepsilon^{-1}+1)(2^L-1)\big(2+\log\left(\frac{N}{W}\right)+\log\varepsilon^{-1}+\mathcal{O}(1)\big)\quad\text{from (7)}.
\end{aligned}$$

◀

The excess of the level structure has $E=W-\varepsilon^{-1}\sum_{l=0}^{L}2^l$ items. We extend the top level to contain the excess, becoming a sliding dictionary with $W_L=E+\varepsilon^{-1}2^L$ and slack 2^L.

▶ **Lemma 10.** *Level L occupies $(1+o(1))(W_L+2^L)(\log\left(\frac{N}{W}\right)+\log\varepsilon^{-1}+\mathcal{O}(1))$ bits.*

Proof. The BackyardCuckoo table that implements the level-L sliding dictionary is initialised to contain W_L+2^L items. We begin by providing a lower bound on this value.

$$\begin{aligned}
W_L+2^L&=(W-\varepsilon^{-1}(2^{L+1}-1))+\varepsilon^{-1}2^L+2^L\\
&=W-\varepsilon^{-1}2^L+\varepsilon^{-1}+2^L\\
&\geq W-\varepsilon^{-1}2^L\\
&\geq W/2\,,
\end{aligned}\tag{8}$$

from inequality (6). We need to account for the cost of storing a block ID, and size of the level-L circular field is $2(\varepsilon^{-1}+E/2^L+1)$ by Equation (4). Therefore, following Theorem 3 and Equation (1), the number of bit the level-L sliding dictionary requires is

$$\begin{aligned}
&(1+o(1))(B(N,W_L+2^L)+(W_L+2^L)\log(2(\varepsilon^{-1}+E/2^L+1)))\\
&=(1+o(1))(W_L+2^L)(\log\left(\frac{N}{W_L+2^L}\right)+\log\left(\varepsilon^{-1}+\frac{W-\varepsilon^{-1}(2^{L+1}-1)}{2^L}\right)+\mathcal{O}(1))\\
&\leq(1+o(1))(W_L+2^L)(\log\left(\frac{N}{W}\right)+\log\left(\varepsilon^{-1}+\frac{\varepsilon^{-1}2^{L+2}-\varepsilon^{-1}2^{L+1}}{2^L}\right)+\mathcal{O}(1))\\
&=(1+o(1))(W_L+2^L)(\log\left(\frac{N}{W}\right)+\log\varepsilon^{-1}+\mathcal{O}(1))\,,
\end{aligned}$$

where line three follows from inequalities (6) and (8). ◀

We have now bounded the space consumption of each of the three components of HistoricalMembership. We now relate the overall space consumption to the information-theoretic lower bound.

▶ **Lemma 11.** *HistoricalMembership requires $(1+o(1))(1+\varepsilon)(\mathcal{B}+W\log\varepsilon^{-1})$ bits of memory, where $\mathcal{B}$ is the information-theoretic lower bound to store a set of size W.*

Proof. The bound in Lemma 9 is at least $(W/2)(\mathcal{O}(1) + \log \varepsilon^{-1})$. This absorbs the bound of the lower levels, of Lemma 8, under the assumption $w = o(\sqrt{W}(1 + \log \varepsilon^{-1}))$. Therefore, it suffices to focus our attention on the upper levels. By Lemmas 9 and 10, levels $\lceil L/2 \rceil$ to L can be stored in

$$\big(W_L + 2^L + (\varepsilon^{-1} + 1)(2^L - 1)\big)(1 + o(1))(\log\left(\frac{N}{W}\right) + \log \varepsilon^{-1} + \mathcal{O}(1)) \tag{9}$$

bits. It suffices to simplify the leading factor.

$$\begin{aligned} W_L + 2^L + (\varepsilon^{-1} + 1)(2^L - 1) &= W - \varepsilon^{-1}(2^{L+1} - 1) + \varepsilon^{-1}2^L + 2^L + (\varepsilon^{-1} + 1)(2^L - 1) \\ &= W + 2^{L+1} - 1 \\ &\leq W + \varepsilon W\,. \end{aligned}$$

As $\mathcal{B} = W(\log(N/W) + \mathcal{O}(1))$, expression (9) simplifies to $(1+o(1))(1+\varepsilon)(\mathcal{B}+W \log \varepsilon^{-1})$. ◀

Combining Lemmas 5, 6 and 11, we arrive at Theorem 2, in some sense, a remarkable result. The bound of Lemma 11 matches the memory commitment of OptimalSM, for $D = \varepsilon W$. HistoricalMembership achieves a representation of a window of a sequence of items with memory allocation that is tight, with respect to membership on the window, and supports Recency queries in time logarithmic in the recency value. This evolution from OptimalSM to HistoricalMembership represents a time-accuracy trade-off, where we refine our understanding of *where* items occur in the sequence paying for a small time overhead. Further, the movement between ExactCuckoo and HistoricalMembership represents a space-time trade-off and asks whether constant update and query times are possible in $o(W \log W)$ space.

We finalize our theoretical development of Recency by observing that Theorem 2 can be applied to approximate set membership. By applying a universal hash function to the sequence of items, we can reduce the size of the universe at the expense of introducing collisions. The result is a space bound that is proportional to W, and tight, up to the first two terms, by Theorem 4.

▶ **Corollary 12.** *For the universal hash function $h : [N] \to [(1 + \varepsilon)W/\delta]$, on input $h(S(t)) = \langle h(s_1), h(s_2), \ldots h(s_t)\rangle$, HistoricalMembership returns the approximate recency on an item with probability $1 - \delta$. The data structure uses $(1 + \varepsilon)W \log(\varepsilon^{-1}\delta^{-1}) + \mathcal{O}(W)$ bits.*

6 Conclusion and future work

We have investigated and defined the notion of Recency through the concept of the sliding window. Existing work, from the sliding membership literature, realizes our definition but cannot accommodate a combination of accuracy and small memory. Our primary innovation is carried through the data structure HistoricalMembership, which supports Recency queries with bounded relative error on top of a succinct representation of the occurred items. The logic of HistoricalMembership is tied to the impression of an equivalence class, wherein items occurring at an equivalent moment in history can be assigned the same estimate. If we think of accuracy-space-time as a triangle, the data structures ExactCuckoo, OptimalSM and HistoricalMembership each occupy a unique face. This leads to the question as to whether operations for Recency data structures can be supported in constant time without sacrificing the memory and accuracy attributes of HistoricalMembership. Alternatively, the three data structures represent the contours and boundaries of the Recency problem.

References

1 Yuriy Arbitman, Moni Naor, and Gil Segev. Backyard cuckoo hashing: Constant worst-case operations with a succinct representation. In *FOCS*, pages 787–796, 2010.

2 Eran Assaf, Ran Ben Basat, Gil Einziger, and Roy Friedman. Pay for a sliding Bloom filter and get counting, distinct elements, and entropy for free. In *INFOCOM*, pages 2204–2212, 2018.

3 Samy Chambi, Daniel Lemire, Owen Kaser, and Robert Godin. Better bitmap performance with roaring bitmaps. *Software: Practice and Experience*, 46(5):709–719, 2016.

4 Martin Dietzfelbinger, Anna Karlin, Kurt Mehlhorn, Friedhelm Meyer Auf Der Heide, Hans Rohnert, and Robert E Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM Journal on Computing*, 23(4):738–761, 1994.

5 Zhe Li, Gwendal Simon, and Annie Gravey. Caching policies for in-network caching. In *ICCCN*, pages 1–7, 2012.

6 Yang Liu, Wenji Chen, and Yong Guan. Near-optimal approximate membership query over time-decaying windows. In *INFOCOM*, pages 1447–1455, 2013.

7 Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. Duplicate detection in click streams. In *WWW*, pages 12–21, 2005.

8 Moni Naor and Eylon Yogev. Tight bounds for sliding Bloom filters. *Algorithmica*, 73(4):652–672, 2015.

9 Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. In *ESA*, pages 121–133, 2001.

10 Daniel D Sleator and Robert E Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985.

11 Jiansheng Wei, Hong Jiang, Ke Zhou, Dan Feng, and Hua Wang. Detecting duplicates over sliding windows with RAM-efficient detached counting Bloom filter arrays. In *NAS*, pages 382–391, 2011.

A Appendix

Algorithm 1 HistoricalMembership.

Procedure `initialise`(W, ε)

- $L \leftarrow \lfloor \log(\varepsilon W) \rfloor - 1$; `// number of levels`
- **for** $l \in \{0, 1, \ldots L\}$ **do**
  - $\Lambda_l \leftarrow$ initialise a hash table that stores at most $2^l(\varepsilon^{-1} + 1)$ items ;
  - $e_l \leftarrow 0$; `// contemporary equivalence class ID`
  - $n_l \leftarrow 2(\varepsilon^{-1} + 1)$; `// circular field of class IDs`
- $E \leftarrow W - \varepsilon^{-1} \sum_{l=0}^{L-1} 2^l$; `// the excess`
- $\Lambda_L \leftarrow$ initialise a hash table that stores at most $2^L(\varepsilon^{-1} + 1) + E$ items;
- $e_L \leftarrow 0$;
- $n_l \leftarrow 2(\varepsilon^{-1} + E/2^L + 1)$; `// circular field of class IDs at level` L
- $t \leftarrow 0$; `// the global clock`
- **return**;

Procedure `query`(x)

- $c \leftarrow 0$;
- **for** $l \in \{0, 1, \ldots, L\}$ **do**
  - $e^* \leftarrow$ retrieve x from Λ_l;
  - **if** $e^* \neq -1$ **then**
    - **return** $c + 2^l \cdot (e_l - e^* \mod n_l)$; `// item located at level` l
  - $c \leftarrow c + \varepsilon^{-1}2^l + (t \mod 2^l)$;
- **return** -1;

Algorithm 2 HistoricalMembership.

```
1  Procedure insert(x)
2      t ← t + 1 ;                                          // update global clock
3      for l ∈ {0, 1, ... L} do
4          if t mod 2^l = 0 then
5              e_l ← e_l + 1 mod n_l;                       // synchronize block IDs
6      E ← insert (x, e_0) into Λ_0;   // insertion returns a set of evicted items
7      for l ∈ {0, 1, ..., L} do
8          E' ← ∅;
9          for (y, e) ∈ E do
10             e* ← e_{l+1} + (e_l − e − (ε^{-1} + 1) mod n_l)/2;
11             E' ← E' ∪ insert (y, e*) into Λ_l;
12         E ← E' ∪ scan Λ_l;
14     return;
```