A Dynamic Space-Efficient Filter with Constant Time Operations

Ioana O. Bercea
Tel Aviv University, Israel
ioana@cs.umd.edu

Guy Even
Tel Aviv University, Israel
guy@eng.tau.ac.il

Abstract

A dynamic dictionary is a data structure that maintains sets of cardinality at most n from a given universe and supports insertions, deletions, and membership queries. A filter approximates membership queries with a one-sided error that occurs with probability at most ε. The goal is to obtain dynamic filters that are space-efficient (the space is $1 + o(1)$ times the information-theoretic lower bound) and support all operations in constant time with high probability. One approach to designing filters is to reduce to the retrieval problem. When the size of the universe is polynomial in n, this approach yields a space-efficient dynamic filter as long as the error parameter ε satisfies $\log(1/\varepsilon) = \omega(\log \log n)$. For the case that $\log(1/\varepsilon) = O(\log \log n)$, we present the first space-efficient dynamic filter with constant time operations in the worst case (whp). In contrast, the space-efficient dynamic filter of Pagh et al. [29] supports insertions and deletions in amortized expected constant time. Our approach employs the classic reduction of Carter et al. [9] on a new type of dictionary construction that supports random multisets.

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

Keywords and phrases Data Structures

Digital Object Identifier 10.4230/LIPIcs.SWAT.2020.11

Related Version A full version of the paper is available at `https://arxiv.org/abs/1911.05060`.

Funding This research was supported by a grant from the United States-Israel Binational Science Foundation (BSF), Jerusalem, Israel, and the United States National Science Foundation (NSF).

Acknowledgements We would like to thank Michael Bender, Martin Farach-Colton, and Rob Johnson for introducing us to this topic and for interesting conversations. Many thanks to Tomer Even for helpful and thoughtful remarks.

1 Introduction

We consider the problem of maintaining datasets subject to insert, delete, and membership query operations. Given a set $\mathcal{D}$ of n elements from a universe $\hat{\mathcal{U}}$, a membership query asks if the queried element $x \in \hat{\mathcal{U}}$ belongs to the set $\mathcal{D}$. When exact answers are required, the associated data structure is called a *dictionary*. When one-sided errors are allowed, the associated data structure is called a *filter*. Formally, given an error parameter $\varepsilon > 0$, a filter always answers "yes" when $x \in \mathcal{D}$, and when $x \notin \mathcal{D}$, it makes a mistake with probability at most ε. We refer to such an error as a *false positive* event[1].

[1] The probability is taken over the random choices that the filter makes.

When false positives can be tolerated, the main advantage of using a filter instead of a dictionary is that the filter requires much less space than a dictionary [9, 25]. Let $\hat{u} \triangleq \left|\hat{\mathcal{U}}\right|$ be the size of the universe and n denote an upper bound on the size of the set at all points in time. The information theoretic lower bound for the space of dictionaries is $\lceil \log_2 \binom{\hat{u}}{n} \rceil = n \log(\hat{u}/n) + \Theta(n)$ bits.[2][3] On the other hand, the lower bound for the space of filters is $n \log(1/\varepsilon)$ bits [9]. In light of these lower bounds, we call a dictionary *space-efficient* when it requires $(1 + o(1)) \cdot n \log(\hat{u}/n)) + \Theta(n)$ bits, where the term $o(1)$ converges to zero as n tends to infinity. Similarly, a space-efficient filter requires $(1 + o(1)) \cdot n \log(1/\varepsilon) + O(n)$ bits.[4]

When the set $\mathcal{D}$ is fixed, we say that the data structure is *static*. When the data structure also supports insertions, we say that it is *incremental*. Data structures that handle both deletions and insertions are called *dynamic*.

The goal is to design dynamic dictionaries and filters that achieve "the best of both worlds" [2]: they are space-efficient and perform operations in constant time in the worst case with high probability.[5]

The Dynamic Setting. One approach for designing dynamic filters was suggested by Pagh et al. [29], outlined as follows. Static (resp., incremental) filters can be obtained from static (resp., incremental) dictionaries for sets by a reduction of Carter et al. [9]. This reduction simply hashes the universe to a set of cardinality n/ε. Due to collisions, this reduction does not directly lead to dynamic filters. Indeed, if two elements x and y in the dataset collide, and x is deleted, how is y kept in the filter? To overcome the problem with deletions, an extension of the reduction to the dynamic setting was proposed by Pagh et al. [29]. This proposal is based on employing a dictionary that maintains multisets rather than sets (i.e., elements in multisets have arbitrary multiplicities). This extension combined with a dynamic dictionary for multisets yields a dynamic filter [29]. In fact, Pagh et al. obtain a dynamic filter that is space-efficient but performs insertions and deletions in amortized constant time (but not in the worst case). Until recently, the design of a dynamic dictionary on multisets that is space-efficient and performs operations in constant time in the worst case whp was open [2]. In this paper, we avoid the need for supporting arbitrary multisets by observing that it suffices to support random multisets (see Sec. 3).[6]

Another approach for designing filters employs retrieval data structures. In the retrieval problem, we are given a function $f : \mathcal{D} \to \{0,1\}^k$, where $f(x)$ is called the *satellite data* associated with $x \in \mathcal{D}$. When an element $x \in \hat{\mathcal{U}}$ is queried, the output y must satisfy $y = f(x)$ if $x \in \mathcal{D}$ (if $x \notin \mathcal{D}$, any output is allowed). By storing as satellite data a random fingerprint of length $\log(1/\varepsilon)$, a retrieval data structure can be employed as a filter at no additional cost in space and with an $O(1)$ increase in time per operation [14, 35] (the increase in time is for computing the fingerprint). This reduction was employed in the static case and it also holds in the dynamic case (see Sec. 6).

---

[2] All logarithms are base 2 unless otherwise stated. $\ln x$ is used to denote the natural logarithm.

[3] This equality holds when $\hat{u}$ is significantly larger than n.

[4] An asymptotic expression that mixes big-O and small-o calls for elaboration. If $\varepsilon = o(1)$, then the asymptotic expression does not require the $O(n)$ addend. If ε is constant, the $O(n)$ addend only emphasizes the fact that the constant that multiplies n is, in fact, the sum of two constants: one is almost $\log(1/\varepsilon)$, and the other does not depend on ε. Indeed, the lower bound in [25] excludes space $(1 + o(1)) \cdot n \log(1/\varepsilon)$ in the dynamic setting for constant values of ε.

[5] By with high probability (whp), we mean with probability at least $1 - 1/n^{\Omega(1)}$. The constant in the exponent can be controlled by the designer and only affects the $o(1)$ term in the space of the dictionary or the filter.

[6] We recently resolved the problem of supporting arbitrary multisets in [6] (thus the dictionary in [6] can support arbitrary multisets vs. the dictionary presented here that only supports random multisets).

Using the dynamic retrieval data structure of Demaine et al. [12], one can obtain a filter that requires $(1+o(1)) \cdot n\log(1/\varepsilon) + \Theta(n \log\log(\hat{u}/n))$ bits and performs operations in constant time in the worst case whp. When the size of the universe satisfies $\hat{u} = \mathsf{poly}(n)$, this reduction yields a space-efficient filter when the false positive probability ε satisfies $\log(1/\varepsilon) = \omega(\log\log n)$ (which we call the *sparse case*).[7] This approach is inherently limited to the sparse case since dynamic retrieval data structures have a space lower bound of $\Theta(n \log\log(\hat{u}/n))$ regardless of the time each operation takes and even when storing two bits of satellite data [28].

Thus, the only case in which a space-efficient dynamic filter with constant time operations is not known is when $\log(1/\varepsilon) = O(\log\log n)$. We refer to this case as the *dense case*. The dense case occurs, for example, in applications in which n is large and ε is a constant (say $\varepsilon = 1\%$).

1.1 Our Contributions

In this paper, we present the first dynamic space-efficient filter for the dense case with constant time operations in the worst case whp. In the following theorem, we assume that the size of the universe $\hat{\mathcal{U}}$ is polynomial in n.[8] We allow ε to be as small as $n/|\hat{\mathcal{U}}|$ (below this threshold, simply use a dictionary). Memory accesses are in the RAM model in which every memory access reads/writes a word of $\Theta(\log n)$ contiguous bits. All computations we perform over one word take constant time (see Sec. 4.1). *Overflow* refers to the event that the space allocated for the filter does not suffice.

▶ **Theorem 1.** *There exists a dynamic filter that maintains a set of at most n elements from a universe $\hat{\mathcal{U}} = [\hat{u}]$, where $\hat{u} = \mathsf{poly}(n)$ with the following guarantees: (1) For every polynomial in n sequence of insert, delete, and query operations, the filter does not overflow whp. (2) If the filter does not overflow, then every operation (query, insert, and delete) can be completed in constant time. (3) The required space is $(1+o(1)) \cdot n\log(1/\varepsilon) + O(n)$ bits. (4) For every query, the probability of a false positive event is bounded by ε.*

Our result is based on the observation that it suffices to use the reduction of Carter et al. [9] on dictionaries that support *random* multisets rather than arbitrary multisets. A random multiset is a uniform random sample (with replacements) of the universe. In Sec. 3, we prove that the reduction of Carter et al. [9] can be applied in this new setting. We then design a dynamic space-efficient dictionary that works on random multisets from a universe $\mathcal{U} = [u]$ with $\log(u/n) = O(\log\log n)$ (Sec. 4). The dictionary supports operations in constant time in the worst case whp. Applying the reduction of Carter et al. [9] to this new dictionary yields our dynamic filter in the dense case. Together with the filter construction for the sparse case (included, for completeness, in Sec. 6), we obtain Theorem 1.

1.2 Our Model

Memory Access Model. We assume that the data structures are implemented in the RAM model in which the basic unit of one memory access is a word. Let w denote the memory word length in bits. We assume that $w = \Theta(\log n)$. See Sec. 4.1 for a discussion of how computations over words are implemented in constant time.

[7] The terms "sparse" and "dense" stem from the fact that the reduction of Carter et al. [9] is employed. Thus, the filter is implemented by a dictionary that stores n elements from a universe of cardinality n/ε.

[8] This is justified by mapping $\hat{\mathcal{U}}$ to $[\mathsf{poly}(n)]$ using 2-independent hash functions [12].

Success Probability. We prove that overflow occurs with probability at most $1/\mathsf{poly}(n)$ and that one can control the degree of the polynomial (the degree of the polynomial only affects the $o(1)$ term in the size bound). In the case of random multisets, the probability of an overflow is a joint probability distribution over the random choices of the dictionary and the distribution over the realizations of the multiset. In the case of sets, the probability of an overflow depends only on the random choices that the filter makes.

Hash Functions. The filter for the dense case employs pairwise independent hash functions and invertible permutations of the universe that can be evaluated in constant time and that have a small space representation (i.e., the one-round Feistel permutations of Arbitman et al. [2] or the quotient hash functions of Demaine et al. [12]). For simplicity, we first analyze the filter construction assuming fully random hash functions (Sec. 4.5). In Sec. 5, we prove that the same arguments hold when we use succinct hash functions.

Worst Case vs. Amortized. An interesting application that emphasizes the importance of worst-case performance is that of handling search engine queries. Such queries are sent in parallel to multiple servers, whose responses are then accumulated to generate the final output. The latency of this final output is determined by the slowest response, thus reducing the average latency of the final response to the worst latency among the servers. See [1,2,7,23] for further discussion on the shortcomings of expected or amortized performance in practical scenarios.

The Extendable Setting. This paper deals with the non-extendable setting in which the bound n on the cardinality of the dataset is known in advance. The filter is allocated space that is efficient with respect to the lower bound on the space of a filter with parameters u, n, ε. The extendable scenario in which space must adapt to the current cardinality of the dataset is addressed in Pagh et al. [31]. In fact, they prove that extendible filters require an extra $\Omega(\log \log n)$ bits per element.

1.3 Related Work

The topic of dictionary and filter design is a fundamental theme in the theory and practice of data structures. We restrict our focus to the results that are closest to our setting (i.e., are space-efficient, take constant time per operation, support dynamic sets).

Dictionaries. The dictionary of Arbitman et al. [2] is the only space-efficient dynamic dictionary for sets that performs all operations in constant time in the worst case with high probability. They leave it as an open question whether one can design a dictionary on multisets with similar guarantees. Indeed, their construction does not seem to extend even to the case of random multisets. The main reason is that the second level of their dictionary (the backyard), implemented as a de-amortized cuckoo hash table, does not support duplicate elements. Moreover, the upper bound on the number of elements that the backyard stores is $\Omega\left(\frac{\log \log n}{(\log n)^{1/3}} \cdot n\right)$. As such, it cannot accommodate storing naive fixed-length counters of elements (which would require $\Theta(\log n)$ bits per element) without rendering the dictionary space-inefficient.

The space-efficient dynamic dictionary for multisets of Pagh et al. [29] supports queries in constant time, and insertions/deletions in amortized expected constant time. For dictionaries on sets, several dynamic constructions support operations in constant time with high

probability but are not space-efficient [1, 11–13]. On the other hand, some dictionaries are space-efficient but do not have constant time guarantees with high probability for all of their operations [16, 21, 33, 36]. For the static case, several space-efficient constructions exist that perform queries in constant time [8, 30, 34, 39].

Filters. The filters of Pagh et al. [29] and Arbitman et al. [2] follow from their respective dictionaries by employing the reduction of Carter et al. [9]. Specifically, the dynamic filter of Pagh et al. [29] supports queries in constant time and insertions and deletions in amortized expected constant time. The incremental filter of Arbitman et al. [2] performs queries in constant time and insertions in constant time with high probability. It does not support deletions.

The construction of Bender et al. [4] describes a dynamic adaptive filter that assumes access to fully random hash functions. [9] The adaptive filter works in conjunction with an external memory dictionary (on the set of elements) and supports operations in constant time with high probability (however, an insert or query operation may require accessing the external memory dictionary). The space of the external memory dictionary is not counted in the space of their filter. The (in-memory) filter they employ is a variant of the dynamic quotient filter [5, 10, 29, 32]. The space-efficient quotient filter employs linear probing and performs operations only in expected constant time for large values of ε [29]. The filter in [4] tries to avoid a large running time per insert operation by bounding the displacement of the inserted element. Hence, if (Robin Hood) linear probing does not succeed after a constant number of words, then the element is inserted in a secondary structure (see Sec. 5.3 in [3]). There is a gap in [3] regarding the question of whether bounded displacements guarantee constant time operations in the worst case. Specifically, searching for an element requires finding the beginning of the "cluster" that contains the "run" associated with that particular element. No description or proof is provided in [3] that the beginning of the cluster is a constant number of words away from the "quotient" in the worst case.

Other filters of interest include the dynamic filter of Pagh et al. [31] that adjusts its space on the fly to the cardinality of the dataset (hence, works without knowing the size of the dataset in advance) and performs operations in constant time. Pagh et al. [31] also prove a lower bound that forces a penalty of $O(\log \log n)$ per element for such "self-adjusting" dynamic filters. Another filter is the cuckoo filter, whose performance depends on the number of elements currently stored in the filter but that has been reported to work well in practice [18, 19]. Space-efficient filters for the static case have been studied extensively [14, 16, 26, 35].

1.4 Paper Organization

Preliminaries are in Sec. 2. The proof that the reduction of Carter et al. [9] can be employed to construct dynamic filters from dynamic dictionaries on random multisets can be found in Sec. 3. The filter for the dense case is described and analyzed in Sec. 4. Section 5 includes a discussion on how to remove the assumption of access to fully random hash functions from Sec. 4.5. Section 6 reviews the construction of a filter in the sparse case based on a retrieval data structure. Theorem 1 is proved in Sec. 7.

[9] Loosely speaking, an adaptive filter is one that fixes false positives after they occur [4, 27].

2 Preliminaries

Notation. The *indicator function* of a set S is the function $\mathbb{1}_S : S \to \{0,1\}$ defined by

$$\mathbb{1}_S(x) \triangleq \begin{cases} 1 & \text{if } x \in S, \\ 0 & \text{if } x \notin S\,. \end{cases}$$

For any positive k, let $[k]$ denote the set $\{0,\ldots,\lceil k\rceil - 1\}$. For a string $a \in \{0,1\}^*$, let $|a|$ denote the length of a in bits.

We define the range of a hash function h to be a set of natural numbers $[k]$ and also treat the image $h(x)$ as a binary string, i.e., the binary representation of $h(x)$ using $\lceil \log_2 k\rceil$ bits.

2.1 Filter and Dictionary Definitions

Let $\hat{\mathcal{U}}$ denote the universe of all possible elements.

Operations. We consider three types of operations:
- $\mathsf{insert}(x_t)$ - insert $x_t \in \hat{\mathcal{U}}$ to the dataset.
- $\mathsf{delete}(x_t)$ - delete $x_t \in \hat{\mathcal{U}}$ from the dataset.
- $\mathsf{query}(x_t)$ - is $x_t \in \hat{\mathcal{U}}$ in the dataset?

Dynamic Sets and Random Multisets. Every sequence of operations $R = \{\mathsf{op}_t\}_{t=1}^{T}$ defines a *dynamic set* $\mathcal{D}(t)$ over $\hat{\mathcal{U}}$ as follows.[10]

$$\mathcal{D}(t) \triangleq \begin{cases} \emptyset & \text{if } t = 0 \\ \mathcal{D}(t-1) \cup \{x_t\} & \text{if } \mathsf{op}_t = \mathsf{insert}(x_t) \\ \mathcal{D}(t-1) \setminus \{x_t\} & \text{if } \mathsf{op}_t = \mathsf{delete}(x_t) \\ \mathcal{D}(t-1) & \text{if } t > 0 \text{ and } \mathsf{op}_t = \mathsf{query}(x_t). \end{cases} \tag{1}$$

▶ **Definition 2.** *A* multiset $\mathcal{M}$ *over* $\hat{\mathcal{U}}$ *is a function* $\mathcal{M} : \hat{\mathcal{U}} \to \mathbb{N}$. *We refer to* $\mathcal{M}(x)$ *as the* multiplicity *of* x. *If* $\mathcal{M}(x) = 0$, *we say that* x *is not in the multiset. We refer to* $\sum_{x\in\hat{\mathcal{U}}} \mathcal{M}(x)$ *as the* cardinality *of the multiset and denote it by* $|\mathcal{M}|$.

The *support* of the multiset is the set $\{x \mid \mathcal{M}(x) \neq 0\}$. The *maximum multiplicity* of a multiset is $\max_{x\in\hat{\mathcal{U}}} \mathcal{M}(x)$.

A *dynamic multiset* $\{\mathcal{M}_t\}_t$ is specified by a sequence of insert and delete operations. Let $\mathcal{M}_t$ denote the multiset after t operations.[11]

$$\mathcal{M}_t(x) \triangleq \begin{cases} 0 & \text{if } t = 0 \\ \mathcal{M}_{t-1}(x) + 1 & \text{if } \mathsf{op}_t = \mathsf{insert}(x) \\ \mathcal{M}_{t-1}(x) - 1 & \text{if } \mathsf{op}_t = \mathsf{delete}(x) \\ \mathcal{M}_{t-1}(x) & \text{otherwise.} \end{cases}$$

We say that a dynamic multiset $\{\mathcal{M}_t\}_t$ has cardinality at most n if $|\mathcal{M}_t| \leq n$, for every t.

▶ **Definition 3.** *A dynamic multiset* $\mathcal{M}$ *over* $\hat{\mathcal{U}}$ *is a* random multiset *if for every* t, *the multiset* $\mathcal{M}_t$ *is the outcome of independent uniform samples (with replacements) from* $\hat{\mathcal{U}}$.

[10] The definition of state in Equation 1 does not rule out a deletion of $x \notin \mathcal{D}(t-1)$. However, we assume that $op_t = \mathsf{delete}(x_t)$ only if $x_t \in \mathcal{D}(t-1)$.

[11] As in the case of dynamic sets, we require that $\mathsf{op}_t = \mathsf{delete}(x_t)$ only if $\mathcal{M}_{t-1}(x_t) > 0$.

Dynamic Filters. A *dynamic filter* is a data structure that maintains a dynamic set $\mathcal{D}(t) \subseteq \hat{\mathcal{U}}$ and is parameterized by an error parameter $\varepsilon \in (0,1)$. Consider an input sequence that specifies a dynamic set $\mathcal{D}(t)$, for every t. The filter outputs a bit for every query operation. We denote the output that corresponds to $\mathsf{query}(x_t)$ by $\mathsf{out}_t \in \{0,1\}$. We require that the output satisfy the following condition:

$$\mathsf{op}_t = \mathsf{query}(x_t) \Rightarrow \mathsf{out}_t \geq \mathbb{1}_{\mathcal{D}(t)}(x_t) \ . \tag{2}$$

The output out_t is an approximation of $\mathbb{1}_{\mathcal{D}(t)}(x_t)$ with a one-sided error. Namely, if $x_t \in \mathcal{D}(t)$, then b_t must equal 1.

▶ **Definition 4** (false positive event)**.** *Let FP_t denote the event that $\mathsf{op}_t = \mathsf{query}(x_t)$, $\mathsf{out}_t = 1$ and $x_t \notin \mathcal{D}(t)$.*

The error parameter $\varepsilon \in (0,1)$ is used to bound the probability of a false positive error.

▶ **Definition 5.** *We say that the* false positive probability *in a filter is bounded by ε if it satisfies the following property. For every sequence R of operations and every t,*

$\Pr[FP_t] \leq \varepsilon$.

The probability space in a filter is induced only by the random choices (i.e., choice of hash functions) that the filter makes. Note also that if $\mathsf{op}_t = \mathsf{op}_{t'} = \mathsf{query}(x)$, where $x \notin \mathcal{D}(t) \cup \mathcal{D}(t')$, then the events FP_t and $\mathrm{FP}_{t'}$ may not be independent (see [4, 27] for a discussion of repeated false positive events and adaptivity).

Dynamic Dictionaries. A *dynamic dictionary* with parameter n is a dynamic filter with parameters n and $\varepsilon = 0$. In the case of multisets, the response out_t of a dynamic dictionary to a $\mathsf{query}(x_t)$ operation must satisfy $\mathsf{out}_t = 1$ iff $\mathcal{M}_t(x_t) > 0$.[12]

When we say that a filter or a dictionary has parameter n, we mean that the cardinality of the input set/multiset is at most n at all points in time.

Success Probability and Probability Space. We say that a dictionary (filter) *works* for sets and random multisets if the probability that the dictionary does not overflow is high (i.e., it is $\geq 1 - 1/\mathsf{poly}(n)$). The probability in the case of random multisets is taken over both the random choices of the dictionary and the distribution of the random multisets. In the case of sets, the success probability depends only on the random choices of the dictionary.

Dense vs. Sparse. We differentiate between two cases in the design of filters, depending on $1/\varepsilon$.

▶ **Definition 6.** *The* dense case *occurs when* $\log(1/\varepsilon) = O(\log \log n)$. *The* sparse case *occurs when* $\log(1/\varepsilon) = \omega(\log \log n)$.

3 Reduction: Filters Based on Dictionaries

In this section, we employ the reduction of Carter et al. [9] to construct dynamic filters out of dynamic dictionaries for random multisets. Our reduction can be seen as a relaxation of the reduction of Pagh et al. [29]. Instead of requiring that the underlying dictionary support multisets, we require that it only supports random multisets. We say that a function $h : A \to B$ is *fully random* if h is sampled u.a.r. from the set of all functions from A to B.

[12] One may also define $\mathsf{out}_t = \mathcal{M}_t(x_t)$.

▷ Claim 7. Consider a fully random hash function $h : \hat{\mathcal{U}} \to \left[\frac{n}{\varepsilon}\right]$ and let $\mathcal{D} \subseteq \hat{\mathcal{U}}$. Then $h(\mathcal{D})$ is a random multiset of cardinality $|\mathcal{D}|$.

Consider a dynamic set $\mathcal{D}(t)$ specified by a sequence of insert and delete operations. Since h is random, an "adversary" that generates the sequence of insertions and deletions for $\mathcal{D}(t)$ becomes an oblivious adversary with respect to $h(\mathcal{D}(t))$ in the following sense. Insertion of x translates to an insertion of $h(x)$ which is a random element (note that $h(x)$ may be a duplicate of a previously inserted element[13]). When deleting at time t, the adversary specifies a previous time $t' < t$ in which an insertion took place, and requests to delete the element that was inserted at time t'.

Let Dict denote a dynamic dictionary for random multisets of cardinality at most n from the universe $\left[\frac{n}{\varepsilon}\right]$.

▶ **Lemma 8.** *For every dynamic set $\mathcal{D}(t)$ of cardinality at most n, the dictionary Dict with respect to the random multiset $h(\mathcal{D}(t))$ and universe $\left[\frac{n}{\varepsilon}\right]$ is a dynamic filter for $\mathcal{D}(t)$ with parameters n and ε.*

Proof Sketch. The Dict records the multiplicity of $h(x_t)$ in the multiset $h(\mathcal{D}(t))$ and so deletions are performed correctly. The filter outputs 1 if and only if the multiplicity of $h(x_t)$ is positive. False positive events are caused by collisions in h. Therefore, the probability of a false positive is bounded by ε because of the cardinality of the range of h. ◀

4 Fully Dynamic Filter (Dense Case)

In this section, we present a fully dynamic filter for the dense case, i.e., $\log(1/\varepsilon) = O(\log \log n)$. The reduction in Lemma 8 implies that it suffices to construct a dynamic dictionary for random multisets. We refer to this dictionary as the *RMS-Dictionary* (RMS - Random Multi-Set).

The RMS-Dictionary is a dynamic space-efficient dictionary for random multisets of cardinality at most n from a universe $\mathcal{U} = [u]$, where $u = n/\varepsilon$. The dense case implies that $\log(u/n) = O(\log \log n)$.

The RMS-Dictionary consists of two levels of dictionaries: a set of *bin dictionaries* (in which most of the elements are stored) and a *spare* (which stores $n_s = O(n/\log^3 n)$ elements). The number of bin dictionaries is m. Let $B \triangleq n/m$, so, in expectation, each bin stores (at most) B elements. To accommodate deviations from the expectation, extra capacity is allocated in the bin dictionaries. Namely, each bin dictionary can store up to $(1+\delta) \cdot B$ elements.

The universe of the spare dictionary is $[u]$. However, the universe of each bin dictionary is $[u/m]$. The justification for the reduced universe of bin dictionaries is that the index of the bin contains $\log m$ bits of information about the elements in it (this reduction in the universe size is often called "quotienting" [5, 12, 24, 29, 30]).

4.1 The Bin Dictionary

The bin dictionary is a (deterministic) dynamic dictionary for (small) multisets. Let u' denote the cardinality of the universe from which elements stored in the bin dictionary are taken. Let n' denote an upper bound on the cardinality of the dynamic multiset stored in a

[13] Duplicates in $h(\mathcal{D}(t))$ are caused by collisions (i.e., $h(x) = h(y)$) rather than by reinsertions.

bin dictionary (i.e., n' includes multiplicities). The bin dictionary must be space-efficient, namely, it must fit in $n' \log(u'/n') + O(n')$ bits, and must support queries, insertions, and deletions in constant time.

The specification of the bin dictionary is even more demanding than the dictionary we are trying to construct. The point is that we focus on parametrizations in which the bin dictionary fits in a constant number of words. Let $B \triangleq \Theta\left(\frac{\log n}{\log(u/n)}\right)$ and $\delta \triangleq \Theta\left(\frac{\log\log n}{\sqrt{B}}\right) = o(1)$. Recall that the number of bins is $m = n/B$.

▶ **Observation 9.** *Let $u' = u/m$ and $n' = (1+\delta) \cdot B$. The bin dictionary for u' and n' fits in $O(\log n)$ bits, and hence fits in a constant number of words.*

We propose two implementations of bin dictionaries that meet the specifications; one is based on lookup tables, and the other on Elias-Fano encoding [17, 20]. The space required by the bin dictionaries that employ global lookup tables meets the information-theoretic lower bound. The space required by the Elias-Fano encoding is within half a bit per element more than the information-theoretic lower bound [17].

Global Tables. We follow Arbitman et al. [2] and employ a global lookup table common to all the bin dictionaries. For the sake of simplicity, we discuss how insertion operations are supported. An analogous construction works for queries and deletions.

The bin dictionary has $s \triangleq \binom{u'+n'}{n'}$ states. Hence, we need to build a table that encodes a function $f : s \times u' \to s$, such that given a state $i \in [s]$ and an element $x \in [u']$, $f(i, x)$ encodes the state of the bin dictionary after x is inserted. The size of the table that stores f is $s \cdot u' \cdot \log s$ bits.

We choose the following parametrization so that the table size is $o(n)$ (recall that n is the upper bound on the cardinality of the whole multiset). Set $B = \frac{1}{2(1+\delta)} \cdot \frac{\log n}{\log(1+u/n)}$ (recall that B is the expected occupancy of a bin). Recall that $u' = u/m$ and $n' = (1+\delta) \cdot B$. Hence,

$$\begin{aligned} s = \binom{u'+n'}{n'} &\leq \left(\frac{e(u'+n')}{n'}\right)^{n'} \\ &\leq \mathsf{poly}(\log n) \cdot \left(1 + \frac{u'}{n'}\right)^{n'} \\ &\leq \mathsf{poly}(\log n) \cdot \left(1 + \frac{u}{n}\right)^{(1+\delta)\cdot\frac{1}{2(1+\delta)}\cdot\frac{\log n}{\log(1+u/n)}} \\ &\leq \mathsf{poly}(\log n) \cdot \sqrt{n}\,. \end{aligned}$$

Since $u' = u/m \leq \mathsf{poly}(\log n)$ and $\log s = O(\log n)$, we conclude that the space required to store f is $o(n)$ bits.

Operations are supported in constant time since the table is addressed by the encoding of the current state and operation.

Elias-Fano Encoding. In this section, we present a bin dictionary implementation that employs (a version of) the Elias-Fano encoding. We refer to this implementation as the *Pocket Dictionary*.

We view each element in the universe $[u']$ as a pair (q, r), where $q \in [B]$ and $r \in [u'/B]$ (we refer to q as the *quotient* and to r as the *remainder*). Consider a multiset $F \triangleq \{(q_i, r_i)\}_{i=0}^{n'-1}$. The encoding of F uses two binary strings, denoted by $\mathsf{header}(F)$ and $\mathsf{body}(F)$, as follows. Let $n_q \triangleq |\{i \in [f] \mid q_i = q\}|$ denote the number of elements that share the same quotient q.

The vector $(n_0, \ldots, n_{B-1})$ is stored in $\mathsf{header}(F)$ in unary as the string $1^{n_0} \circ 0 \circ \cdots \circ 1^{n_{B-1}} \circ 0$. The length of the header is $B + n'$. The concatenation of the remainders is stored in $\mathsf{body}(F)$ in nondecreasing lexicographic order of $\{(q_i, r_i)\}_{i \in [n']}$. The length of the body is $n' \cdot \log(u'/B)$. The space required is $B + n'(1 + \log(u'/n)$ bits, which meets the required space bound since $B = O(n')$.

We argue that operations in a Pocket Dictionary can be executed in constant time if the Pocket Dictionary fits in a single word. Here we propose to extend the classical RAM model in which instructions such as comparison, addition, and multiplication take constant time [22].[14] These instructions require Boolean circuits of depth $\log w$, where w denotes the number of bits per word. Moreover, multiplication is implemented using circuits with $\Theta(w^2)$ gates (i.e., all the partial products are computed). Hence, we consider an extension of the RAM model in which instructions over words can be executed in constant time if there exists a Boolean circuit with constant fan-in that computes the instruction in $O(\log w)$ depth using $O(w^2)$ gates.

Indeed, operations over Pocket Dictionaries can be supported by circuits of depth $\log w$ with $O(w \log w)$ gates. Consider an insertion operation of an element (q, r). Insertion is implemented using the following steps (all implemented by circuits):

- **(i)** Locate in the header the positions of qth zero and the zero that proceeds it (this is a select operation). Let j and i denote these positions within the header. This implies that $\sum_{q' < q} n_{q'} = i - (q - 1)$ and $n_q = j - i - 1$.
- **(ii)** Update the header by shifting the suffix starting in position j by one position and inserting a 1 in position j.
- **(iii)** Read n_q remainders in the body, starting from position $i - (q - 1)$. These remainders are compared in parallel with r, to determine the position p within the body in which r should be inserted (this is a rank operation over the outcomes of the comparisons).
- **(iv)** Shift the suffix of the body starting with position p by $|r|$ bits, and copy r into the body starting at position r.

Modern instruction sets support instructions such as rank, select, and SIMD comparisons (shifts are standard instructions) [3, 32, 37]. Hence, one can implement Pocket Dictionary operations in constant time using such instruction sets.

4.2 The Spare

The spare is a dynamic dictionary that maintains (arbitrary) multisets of cardinality at most n_s from the universe $\mathcal{U}$ with the following guarantees: (1) For every $\mathsf{poly}(n)$ sequence of operations (insert, delete, or query), the spare does not overflow whp. (2) If the spare does not overflow, then every operation (query, insert, delete) takes $O(1)$ time. (3) The required space is $O(n_s \log u)$ bits.[15]

We propose to implement the spare by using a dynamic dictionary on sets with constant time operations in which counters are appended to elements. To avoid having to discuss the details of the interior modifications of the dictionary, we propose a black-box approach that employs a retrieval data structure in addition to the dictionary.

[14] In modern CPUs, addition and comparison take a single clock cycle, multiplication may take 2 cycles.
[15] Since $n_s = o(n/\log n)$ and $\log u = O(\log n)$, the space consumed by the spare is $o(n)$.

▶ **Observation 10.** *Any dynamic dictionary on sets of cardinality at most n_s from the universe $\mathcal{U}$ can be used to implement a dynamic dictionary on arbitrary multisets of cardinality at most n_s from the universe $\mathcal{U}$. This reduction increases the space of the dictionary on sets by an additional $\Theta(\log n_s + \log\log(u/n_s))$ bits per element and increases the time per operation by a constant.*

Proof. The dictionary on multisets (MS-Dict) can be obtained by employing a dictionary on sets (Dict) and the dynamic retrieval data structure (Ret) of Demaine et al. [12]. The dictionary on sets (Dict) stores the support of the input multiset. The retrieval data structure (Ret) stores as satellite data the multiplicity of each element in the support. The space that Ret requires is $\Theta(n_s \log n_s + n_s \log\log(u/n_s))$, since the satellite data occupies $\log n_s$ bits.

On membership queries, the dictionary accesses Dict. When a new element x is inserted, MS-Dict inserts x in Dict and adds x with satellite value 1 to Ret. In the case of insertions of duplicates or deletions, Ret is updated to reflect the current multiplicity of the element. If upon deletion, the multiplicity of the element reaches 0, the element is deleted from Dict and from Ret. Since the Ret supports operations in constant time, this reduction only adds $O(1)$ time to the operations on Dict. ◀

To finish the description of the spare, we set $n_s \triangleq n/(\log^3 n)$ and employ Obs. 10 with a dynamic dictionary on sets that requires $O(n_s \log(u/n_s))$ bits and performs operations in constant time whp [1, 2, 11–13]. Under our definition of n_s and since $\log(u/n) = O(\log\log n)$, the spare then requires $o(n)$ bits. Moreover, the spare does not overflow whp (see Claim 11).

4.3 Hash Functions

We consider three hash functions, the bin index, quotient, and remainder, as follows: [16] (Recall that $n = m \cdot B$.)

$$h^b : \mathcal{U} \to [m] \qquad \text{(bin index)}$$
$$q : \mathcal{U} \to [B] \qquad \text{(quotient)}$$
$$r : \mathcal{U} \to [u/n] \qquad \text{(remainder)}$$

We consider three settings for the hash functions:

(i) Fully random hash functions.

(ii) In the case that the dataset is a random multiset, the values of the hash functions are taken simply from the bits of x. Namely $h^b(x)$ is the first $\log(n/B)$ bits, $q(x)$ is the next $\log B$ bits, and $r(x)$ is the last $\log(u/n)$ bits (to be more precise, one needs to divide x and take remainders). Since x is chosen independently and uniformly at random, the hash functions are fully random.

(iii) Hash functions sampled from special distributions of hash functions (with small representation and constant evaluation time).

4.4 Functionality

A query(x) is implemented by searching for $(q(x), r(x))$ in the bin dictionary of index $h^b(x)$. If the pair is not found, the query is forwarded to the spare. An insert(x) operation first attempts to insert $(q(x), r(x))$ in the bin dictionary of index $h^b(x)$. If the bin dictionary

[16] One could define the domain of the quotient function $q(x)$ and the remainder function $r(x)$ to be $[u/m]$ instead of $\mathcal{U}$.

is full, it forwards the insertion to the spare. A delete(x) operation searches for the pair $(q(x), r(x))$ in the bin dictionary of index $h^b(x)$ and deletes it (if found). Otherwise, it forwards the deletion to the spare.

4.5 Overflow Analysis[17]

The proposed dictionary consists of two types of dictionaries: many small bin dictionaries and one spare. The overflow of a bin dictionary is handled by sending the element to the spare. Hence, for correctness to hold, we need to prove that the spare does not overflow whp.

The first challenge that one needs to address is that the dictionary maintains a dynamic multiset $D(t)$ (see [12]). Consider the insertion of an element x at time t. If bin $h^b(x)$ is full at time t, then x is inserted in the spare. Now suppose that an element y with $h^b(y) = h^b(x)$ is deleted at time $t+1$. Then, bin $h^b(x)$ is no longer full, and x cannot "justify" the fact that it is in the spare at time $t+1$ based on the present dynamic multiset $D(t+1)$. Indeed, x is in the spare due to "historical reasons".

The second challenge is that the events that elements are sent to the spare are not independent. Indeed, if x is sent to the spare at time t, then we know that there exists a full bin. The existence of a full bin is not obvious if the cardinality of $D(t)$ is small. Hence, we cannot even argue that the indicator variables for elements being sent to the spare are negatively associated.

The following claim bounds the number of elements stored in the spare. Using the same proof, one could show that the number of elements in the spare is bounded by $n/(\log n)^c$ whp, for every constant c.

▷ Claim 11. The number of elements stored in the spare is less than $n/\log^3 n$ with high probability.

Proof. Consider a dynamic multiset $D(t)$ at time t. To simplify notation, let $\{x_1, \ldots, x_{n'}\}$ denote the multiset $D(t)$ (hence, the elements need not be distinct, and $n' \leq n$). Let t_i denote the time in which x_i was inserted to $D(t)$. Let X_i denote the random variable that indicates if x_i is stored in the spare (i.e., $X_i = 1$ iff bin $h^b(x_i)$ is full at time t_i). Our goal is to prove that the spare at time t does not overflow whp, namely:

$$\Pr\left[\sum_{i=1}^{n'} X_i \geq \frac{n}{\log^3 n}\right] \leq n^{-\omega(1)} \,. \tag{3}$$

The claim follows from Eq. 3 by applying a union bound over the whole sequence of operations.

To prove Eq. 3, we first bound the probability that a bin is full. Let $\gamma \triangleq e^{-\delta^2 \cdot B/3}$. Fix a bin b, by a Chernoff bound, the probability that bin b is full is at most γ. Indeed:

- **(i)** Each element belongs to bin b with probability B/n. Hence, the expected occupancy of a bin is B.
- **(ii)** The variables $\left\{h^b(x_j)\right\}_{j=1}^{n'}$ are independent.
- **(iii)** A bin is full if at least $(1+\delta) \cdot B$ elements belong to it.

We overcome the problem that the random variables $\{X_i\}_i$ are not independent as follows. Let F_t denote the set of full bins at time t. If $|F_t| \leq 6\gamma m$, let $\hat{F}_t$ denote an arbitrary superset of F_t that contains $6\gamma m$ bins. Note that it is unlikely that there exists a t such that $|F_t| > 6\gamma m$. Indeed, by linearity of expectation, $\mathbb{E}\left[|F_t|\right] \leq \gamma m$. By a Chernoff bound, $\Pr\left[F_t \leq 6\gamma m\right] \leq 2^{-6\gamma m}$.

[17] The proofs in this section assume that the hash functions are fully random. See Section 5 for a discussion of special families of hash functions.

Define $\hat{X}_i$ to be the random variable that indicates if $h^b(x_i) \in \hat{F}_{t_i}$. Namely, $\hat{X}_i = 1$ if x_i belongs to a full bin or a bin that was added to $\hat{F}_{t_i}$. Thus, $\hat{X}_i \geq X_i$. The key observation is that the random variables $\left\{\hat{X}_i\right\}_{i=1}^{n'}$ are independent and identically distributed because the bin indexes $\left\{h^b(x_i)\right\}_i$ are independent and uniformly distributed.

The rest of the proof is standard. Recall that $B = O(\log n)$ and $\delta^2 = \Theta\left(\frac{(\log\log n)^2}{B}\right)$.

Let G_t denote the event that $F_t \leq 6\gamma m$. Since $\Pr[G_t] \leq 2^{-6\gamma m} \leq 2^{-\sqrt{n}}$, by a union bound $\Pr\left[\bigcup_{i=1}^{n'} G_{t_i}\right] \leq n \cdot 2^{-\sqrt{n}}$.

The expectation of $\hat{X}_i$ is 6γ (conditioned on the event $\bigcap_{i=1}^{n'} G_{t_i}$). Since $n/\log^3 n = \omega(\gamma n)$, for a sufficiently large n, by Chernoff bound $\Pr\left[\sum_{i=1}^{n'} \hat{X}_i \geq \frac{n}{\log^3 n} \mid \bigcap_{i=1}^{n'} G_{t_i}\right] < 2^{-n/\log^3 n}$.

We conclude that

$$\Pr\left[\sum_{i=1}^{n'} \hat{X}_i \geq \frac{n}{\log^3 n}\right] \leq \Pr\left[\bigcup_{i=1}^{n'} G_{t_i}\right] + \Pr\left[\sum_{i=1}^{n'} \hat{X}_i \geq \frac{n}{\log^3 n} \mid \bigcap_{i=1}^{n'} G_{t_i}\right] \tag{4}$$

$$\leq n \cdot 2^{-\sqrt{n}} + 2^{-n/\log^3 n} = n^{-\omega(1)}\,, \tag{5}$$

and Eq. 3 follows. ◁

5 Succinct Hash Functions

In this section we discuss how to replace the fully random hash functions from Sec. 4 with succinct hash functions (i.e., representation requires $o(n)$ bits) that have constant evaluation time in the RAM model. Specifically, we describe how to select hash functions $h^b(x), r(x), q(x)$ for the RMS-dictionary used for constructing the dynamic filter in the dense case.

The construction proceeds in two stages and uses existing succint constructions for highly independent hash functions [15, 38]. First, we employ a permutation function π to partition the universe $\hat{\mathcal{U}}$ into $M = n^{9/10}$ equal parts. The permutation π can be either the one-round Feistel permutation from [2] or the quotient permutation from [12]. We think of π as a pair of functions, i.e., $\pi(x) = (h_1(x), h_2(x))$ with $h_1(x) \in [M]$. This induces a partition of the dynamic set into $M = n^{9/10}$ subsets of size at most $n^{1/10} + n^{3/40}$ whp. Each subset consists of the $h_2(x)$ values of the elements $x \in \mathcal{D}$ that share the same $h_1(x)$ value.

In the second step, we instantiate the RMS-Dictionary separately for each subset. Each dictionary instance employs the same k-wise independent hash function $f^b : [\hat{u}/M] \to [m]$ with $k = n^{1/10} + n^{3/40}$. We define $h^b(x) = f^b(h_2(x))$ to be the bin of x. From the perspective of each dictionary instantiation, $h^b(x)$ is sampled independently and uniformly at random, so throughout the sequence of $\mathsf{poly}(n)$ operations, the spare does not overflow whp.

We now describe how the quotient $q(x)$ and the remainder $r(x)$ are chosen. We sample a 2-independent hash function $(f, g) : [\hat{u}/M] \to [B] \times [1/\varepsilon]$. Define $q(x) \triangleq f(h_2(x))$ and $r(x) \triangleq g(h_2(x))$.

▷ Claim 12. Consider a filter based on an RMS-dictionary that employs the hash functions $h^b(x), q(x), r(x)$ described in this section. Then the probability of a false positive event is bounded by 2ε.

Proof. Consider a query y that is not in the dataset $D(t)$ at time t. One cause of failure is when too many elements of $\mathcal{D}(t)$ are mapped to bin $h^b(y)$. The probability that more that $(1+\delta)B$ elements are mapped to bin $h^b(y)$ is bounded by $2e^{-\delta^2 B/3}$ (see the proof of Claim 11). Since $\delta^2 B = (\log\log n)^2$, and since $\log(1/\varepsilon) = O(\log\log n)$, this probability is $o(\varepsilon)$.

Now assume that at most $(1+\delta)B$ elements in $\mathcal{D}(t)$ were mapped to bin $h^b(y)$. Since h is bijective and (f, g) are selected from a family of 2-independent hash functions, the probability of a collision with an element in bin $h^b(y)$ is at most $(1+\delta)B \cdot \frac{\varepsilon}{B} = (1+\delta) \cdot \varepsilon$. The claim follows since $\delta = o(1)$. ◁

6 Dynamic Filter via Retrieval (Sparse Case)

In this section, we present a space-efficient dynamic filter for the case that $\log\left(\frac{1}{\varepsilon}\right) = \omega(\log\log n)$ (sparse case). We let n denote an upper bound on the cardinality of a dynamic set over a $\mathsf{poly}(n)$ sequence of insertions and deletions. We let $\hat{\mathcal{U}}$ denote a universe of cardinality $\hat{u}$ that satisfies $\hat{u} = \mathsf{poly}(n)$. The construction is based on a reduction from dynamic retrieval. The construction relies on the fact that in dynamic retrieval structures (e.g., [12]), the overhead per element is $O(\log\log n)$. This overhead is $o(\log(1/\varepsilon))$ in the sparse case.

Dietzfelbinger and Pagh [14] formulate a reduction that uses a static retrieval data structure storing k bits of satellite data per element to implement a static filter with false positive probability 2^{-k}. The reduction is based on the assumption that retrieval data structure is "well behaved" with respect to negative queries. Namely, a query for $x \in \hat{\mathcal{U}} \setminus \mathcal{D}$ returns either "fail" or the satellite data of an (arbitrary) element $y \in \mathcal{D}$. The reduction incurs no additional cost in space and adds $O(1)$ extra time to the query operations (to evaluate the fingerprint). We note that the same reduction can be employed in the dynamic case. Specifically, the following holds:

▶ **Observation 13.** *Assume access to a family of pairwise independent hash functions $h : \hat{\mathcal{U}} \to [k]$.* [18] *Then any dynamic retrieval data structure that stores $h(x)$ as satellite data for element x can be used to implement a dynamic filter with false positive probability 2^{-k}. The space of the resulting filter is the same as the space of the retrieval data structure and the time per operation increases by a constant (due to the computation of $h(x)$).*

The question of designing a space-efficient dynamic filter now boils down to:

(i) Choose a suitable dynamic retrieval data structure.

(ii) Determine the range of false positive probabilities ε for which the reduction yields a space-efficient dynamic filter.

We resolve this question by employing the retrieval data structure of Demaine et al. [12] in the sparse case.

▷ Claim 14. There exists a dynamic filter in the sparse case that maintains a set of at most n elements from the universe $\hat{\mathcal{U}} = [\hat{u}]$, where $\hat{u} = \mathsf{poly}(n)$ such that, for any ε such that $\log(1/\varepsilon) = \omega(\log\log n)$, the following hold: (1) For every sequence of $\mathsf{poly}(n)$ opertations (i.e, insert, delete, or query), the filter does not overflow whp. (2) If the filter does not overflow, then every operation (query, insert, and delete) takes $O(1)$ time. (3) The required space is $(1+o(1)) \cdot n\log(1/\varepsilon)$ bits. (4) For every query, the probability of a false positive event is bounded by ε.

Proof. The dynamic retrieval data structure in [12] uses a dynamic perfect hashing data structure for n elements from the universe $\mathcal{U}$ of size u that maps each element to a unique value in a given range $[n+t]$, for any $t > 0$. [19] The space that the perfect hashing data structure

[18] We note that Dietzfelbinger and Pagh [14] assume access to fully random hash functions. Pairwise independence suffices, however, as noted by [35].

[19] We note that one could also use the dynamic perfect hashing scheme proposed in Mortesen et al. [28] with similar space and performance guarantees.

occupies is $\Theta\left(n \log\log \frac{\hat{u}}{n} + n \log \frac{n}{t+1}\right)$. All operations are performed in $O(1)$ time and the perfect hashing data structure fails with $1/\,\mathsf{poly}(n)$ probability over a sequence of $\mathsf{poly}(n)$ operations. A retrieval data structure can be obtained by allocating an array of $n+t$ entries, each used to store satellite data of k bits. The satellite data associated with an element x is stored at the position in the array that corresponds to the hash code associated with x. The retrieval data structured obtained this way occupies $(n+t)\cdot k + \Theta\left(n \log\log \frac{\hat{u}}{n} + n \log \frac{n}{t+1}\right)$ bits. It performs every operation in constant time and fails with probability $1/\,\mathsf{poly}(n)$ over a sequence of $\mathsf{poly}(n)$ operations.

For the filter construction, we set $k \triangleq \log(1/\varepsilon)$ and $t \triangleq n/\log n$. Since $\log\log(\hat{u}/n) = O(\log\log n)$ and $\log(1/\varepsilon) = \omega(\log\log n)$, the filter we obtain occupies $(1+o(1))\cdot n\log(1/\varepsilon)$ bits. It performs all operations in constant time and does not overflow whp over a sequence of $\mathsf{poly}(n)$ operations. ◁

7 Proof of Theorem 1

The proof of Thm. 1 deals with the sparse case and dense case separately. The theorem for the sparse case, in which $\log(1/\varepsilon) = \omega(\log\log n)$, is proven in Sec. 6.

The proof for the dense case employs the reduction in Lemma 8 with the RMS-Dictionary construction described in Sec. 4. Let $\mathcal{U} = [u]$ where $u = n/\varepsilon$ denotes the universe of an RMS-Dictionary that can store a random multiset of cardinality at most n. In this case, the assumption that $\log(1/\varepsilon) = O(\log\log n)$ translates into $\log(u/n) = O(\log\log n)$.

The time per operation is constant because the RMS-dictionary supports operations in constant time. The space consumed by the RMS-Dictionary equals the sum of spaces for the bin dictionaries and the spare. This amounts to $m\cdot n'\cdot\log(u'/n') + m\cdot O(n') + o(n) \leq (1+\delta)\cdot n\cdot\log(1/\varepsilon) + O(n)$. Since $\delta = o(1)$, the filter is space-efficient, as required. Finally, by Claim 11, the spare does not overflow whp.

References

1. Yuriy Arbitman, Moni Naor, and Gil Segev. De-amortized cuckoo hashing: Provable worst-case performance and experimental results. In *International Colloquium on Automata, Languages, and Programming*, pages 107–118. Springer, 2009.
2. Yuriy Arbitman, Moni Naor, and Gil Segev. Backyard cuckoo hashing: Constant worst-case operations with a succinct representation. In *2010 IEEE 51st Annual Symposium on Foundations of Computer Science*, pages 787–796. IEEE, 2010.
3. Michael A. Bender, Martin Farach-Colton, Mayank Goswami, Rob Johnson, Samuel McCauley, and Shikha Singh. Bloom filters, adaptivity, and the dictionary problem. *CoRR*, abs/1711.01616, 2017. `arXiv:1711.01616`.
4. Michael A. Bender, Martin Farach-Colton, Mayank Goswami, Rob Johnson, Samuel McCauley, and Shikha Singh. Bloom filters, adaptivity, and the dictionary problem. In *59th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2018, Paris, France, October 7-9, 2018*, pages 182–193, 2018. `doi:10.1109/FOCS.2018.00026`.
5. Michael A. Bender, Martin Farach-Colton, Rob Johnson, Russell Kraner, Bradley C. Kuszmaul, Dzejla Medjedovic, Pablo Montes, Pradeep Shetty, Richard P. Spillane, and Erez Zadok. Don't thrash: How to cache your hash on flash. *PVLDB*, 5(11):1627–1637, 2012. `doi:10.14778/2350229.2350275`.
6. Ioana O. Bercea and Guy Even. A dynamic dictionary for multisets. work in progress, 2020.
7. Andrei Broder and Michael Mitzenmacher. Using multiple hash functions to improve ip lookups. In *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications.*

Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No. 01CH37213), volume 3, pages 1454–1463. IEEE, 2001.

8 Andrej Brodnik and J. Ian Munro. Membership in constant time and almost-minimum space. *SIAM Journal on Computing*, 28(5):1627–1640, 1999.

9 Larry Carter, Robert Floyd, John Gill, George Markowsky, and Mark Wegman. Exact and approximate membership testers. In *Proceedings of the tenth annual ACM symposium on Theory of computing*, pages 59–65. ACM, 1978.

10 J.G. Cleary. Compact hash tables using bidirectional linear probing. *IEEE Transactions on Computers*, (9):828–834, 1984.

11 Ketan Dalal, Luc Devroye, Ebrahim Malalla, and Erin McLeish. Two-way chaining with reassignment. *SIAM Journal on Computing*, 35(2):327–340, 2005.

12 Erik D Demaine, Friedhelm Meyer auf der Heide, Rasmus Pagh, and Mihai Pătraşcu. De dictionariis dynamicis pauco spatio utentibus. In *Latin American Symposium on Theoretical Informatics*, pages 349–361. Springer, 2006.

13 Martin Dietzfelbinger and Friedhelm Meyer auf der Heide. A new universal class of hash functions and dynamic hashing in real time. In *International Colloquium on Automata, Languages, and Programming*, pages 6–19. Springer, 1990.

14 Martin Dietzfelbinger and Rasmus Pagh. Succinct data structures for retrieval and approximate membership. In *International Colloquium on Automata, Languages, and Programming*, pages 385–396. Springer, 2008.

15 Martin Dietzfelbinger and Michael Rink. Applications of a splitting trick. In *International Colloquium on Automata, Languages, and Programming*, pages 354–365. Springer, 2009.

16 Martin Dietzfelbinger and Christoph Weidling. Balanced allocation and dictionaries with tightly packed constant size bins. *Theoretical Computer Science*, 380(1-2):47–68, 2007.

17 Peter Elias. Efficient storage and retrieval by content and address of static files. *Journal of the ACM (JACM)*, 21(2):246–260, 1974.

18 David Eppstein. Cuckoo filter: Simplification and analysis. In *15th Scandinavian Symposium and Workshops on Algorithm Theory, SWAT 2016, June 22-24, 2016, Reykjavik, Iceland*, pages 8:1–8:12, 2016. doi:10.4230/LIPIcs.SWAT.2016.8.

19 Bin Fan, David G. Andersen, Michael Kaminsky, and Michael Mitzenmacher. Cuckoo filter: Practically better than Bloom. In *CoNEXT*, pages 75–88. ACM, 2014.

20 Robert Mario Fano. On the number of bits required to implement an associative memory. memorandum 61. *Computer Structures Group, Project MAC, MIT, Cambridge, Mass.*, 1971.

21 Dimitris Fotakis, Rasmus Pagh, Peter Sanders, and Paul Spirakis. Space efficient hash tables with worst case constant access time. *Theory of Computing Systems*, 38(2):229–248, 2005.

22 Torben Hagerup. Sorting and searching on the word ram. In *Annual Symposium on Theoretical Aspects of Computer Science*, pages 366–398. Springer, 1998.

23 Adam Kirsch and Michael Mitzenmacher. Using a queue to de-amortize cuckoo hashing in hardware. In *Proceedings of the Forty-Fifth Annual Allerton Conference on Communication, Control, and Computing*, volume 75, 2007.

24 Donald E Knuth. The art of computer programming, vol. 3: Searching and sorting. *Reading MA: Addison-Wisley*, 1973.

25 Shachar Lovett and Ely Porat. A lower bound for dynamic approximate membership data structures. In *2010 IEEE 51st Annual Symposium on Foundations of Computer Science*, pages 797–804. IEEE, 2010.

26 Michael Mitzenmacher. Compressed Bloom filters. *IEEE/ACM Transactions on Networking (TON)*, 10(5):604–612, 2002.

27 Michael Mitzenmacher, Salvatore Pontarelli, and Pedro Reviriego. Adaptive cuckoo filters. In *2018 Proceedings of the Twentieth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 36–47. SIAM, 2018.

28 Christian Worm Mortensen, Rasmus Pagh, and Mihai Pătraşcu. On dynamic range reporting in one dimension. In *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*, pages 104–111. ACM, 2005.

29 Anna Pagh, Rasmus Pagh, and S. Srinivasa Rao. An optimal Bloom filter replacement. In *SODA*, pages 823–829. SIAM, 2005.

30 Rasmus Pagh. Low redundancy in static dictionaries with constant query time. *SIAM Journal on Computing*, 31(2):353–363, 2001.

31 Rasmus Pagh, Gil Segev, and Udi Wieder. How to approximate a set without knowing its size in advance. In *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, pages 80–89. IEEE, 2013.

32 Prashant Pandey, Michael A. Bender, Rob Johnson, and Robert Patro. A general-purpose counting filter: Making every bit count. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 775–787, 2017. doi:10.1145/3035918.3035963.

33 Rina Panigrahy. Efficient hashing with lookups in two memory accesses. In *Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 830–839. Society for Industrial and Applied Mathematics, 2005.

34 Mihai Pătraşcu. Succincter. In *2008 49th Annual IEEE Symposium on Foundations of Computer Science*, pages 305–313. IEEE, 2008.

35 Ely Porat. An optimal Bloom filter replacement based on matrix solving. In *International Computer Science Symposium in Russia*, pages 263–273. Springer, 2009.

36 Rajeev Raman and Satti Srinivasa Rao. Succinct dynamic dictionaries and trees. In *International Colloquium on Automata, Languages, and Programming*, pages 357–368. Springer, 2003.

37 James Reinders. AVX-512 Instructions. https://software.intel.com/en-us/articles/intel-avx-512-instructions, 2013.

38 Alan Siegel. On universal classes of extremely random constant-time hash functions. *SIAM Journal on Computing*, 33(3):505–543, 2004.

39 Huacheng Yu. Nearly optimal static las vegas succinct dictionary. *arXiv preprint arXiv:1911.01348*, 2019.