Xor Filters: Faster and Smaller Than Bloom and Cuckoo Filters

THOMAS MUELLER GRAF and DANIEL LEMIRE*, University of Quebec (TELUQ), Canada

The Bloom filter provides fast approximate set membership while using little memory. Engineers often use these filters to avoid slow operations such as disk or network accesses. As an alternative, a cuckoo filter may need less space than a Bloom filter and it is faster. Chazelle et al. proposed a generalization of the Bloom filter called the Bloomier filter. Dietzfelbinger and Pagh described a variation on the Bloomier filter that can answer approximate membership queries over immutable sets. It has never been tested empirically, to our knowledge. We review an efficient implementation of their approach, which we call the xor filter. We find that xor filters can be faster than Bloom and cuckoo filters while using less memory. We further show that a more compact version of xor filters (xor+) can use even less space than highly compact alternatives (e.g., Golomb-compressed sequences) while providing speeds competitive with Bloom filters.

CCS Concepts: • **Theory of computation → Bloom filters and hashing**.

Additional Key Words and Phrases: Bloom Filters, Cuckoo Filters, Approximate Set Membership

1 INTRODUCTION

The classical data structure for approximate membership is the Bloom filter [4]. It may be the best-known probabilistic data structure. A Bloom filter is akin to a set data structure in that we can add keys, and check whether a given key is present in the set. There is a small probability that a key is incorrectly reported as being present, an event we call a *false positive*. However, Bloom filters can use less memory than the original set. Thus, Bloom filters accept a small probability of error for a reduced memory usage.

Approximate set membership has many applications: e.g., scanning for viruses using payload signatures [18], filtering bad keywords or addresses, and fast language identification for strings [22]. Write-optimized key-value stores [11] such as log-structured merge (LSM) trees [29] are another important use case. In such stores, an in-memory data structure avoids expensive disk accesses.

We want our data structures to be fast and to use little memory. In this respect, conventional Bloom filters can be surpassed:

- Bloom filters generate many random-access queries. For efficient memory usage, a Bloom filter with a false-positive probability ϵ should use about $-\log_2 \epsilon$ hash functions [10]. At a false-positive probability of 1%, seven hash functions are thus required. Even if the computation of the hash functions were free, doing many random memory accesses can be expensive.
- The theoretical lower bound for an approximate membership data structure with a false-positive probability ϵ is $-\log_2 \epsilon$ bits per key [10]. When applied in an optimal manner, Bloom filters use 44% more memory than the theoretical lower bound.

Practically, Bloom filters are often slower and larger than alternatives such as cuckoo filters [20]. Can we do better than even cuckoo filters?

Bonomi et al. [5] as well as Broder and Mitzenmacher [10] remarked that for static sets, essentially optimal memory usage is possible using a *perfect hash function* and fingerprints. They dismissed this possibility in part because perfect hash functions might be too expensive to compute. Yet Dietzfelbinger and Pagh [17] described a seemingly practical implementation of this idea which we call an xor filter. It builds on closely related work such as Bloomier filters [12, 13].

Authors' address: Thomas Mueller Graf, thomas.tom.mueller@gmail.com; Daniel Lemire, lemire@gmail.com, University of Quebec (TELUQ), 5800 Saint-Denis, Office 1105, Montreal, Quebec, Canada, H2S 3L5.

To our knowledge, xor filters were never implemented and benchmarked. We present the first experimental evaluation. We find that they perform well, being often faster than both Bloom and cuckoo filters. For common use cases, they require less memory. Furthermore, we can improve their memory usage with only a modest performance penalty, using a relatively simple compression technique (see § 3.3). We make our software freely available to ensure reproducibility.

Our main result is that xor filters have merit as a practical data structure. They are fast, compact and we found them easy to implement.

2 RELATED WORK

We find many Bloom filters and related data structures within database systems [11] to avoid disk accesses. A popular strategy for designing database engines that must support frequent updates is the log-structured merge (LSM) tree [29]. At a high-level, LSM trees maintain a fast in-memory component that is merged, in batches, to data in persistent storage. The in-memory component accumulates database updates thus amortizing the update cost to persistent storage. To accelerate lookups, many LSM tree implementations (e.g., levelDB, RocksDB, WiredTiger) use Bloom filters. When merging the components, usually a new filter is built. We could, instead, update existing filters. However, data structures that support fast merging (e.g., Bloom filters) require either the original filters to have extra capacity, or the result of the merger to have higher false-positive probabilities [2].

Many applications of Bloom filters and related data structures are found in networking, where we seek to avoid unnecessary network access. Generally, whenever a filter must be sent through a network connection to other computers (e.g., to cache and prevent network queries), we might be able to consider the filter as immutable [27] on the receiving machine.

2.1 Bloom Filter Variants

Standard Bloom filters [4] consist of a collection of hash functions $h_1, h_2, \ldots, h_k$, which map each possible key to a fixed integer which we interpret as an index value, and an array of bits B, initialized with zeros. The size of the array and the number of hash functions k are parameters of the filter. When we add a key x, we hash it with each hash function, and set the corresponding bits:

$$\begin{aligned} B[h_1(x)] &\leftarrow 1, \\ B[h_2(x)] &\leftarrow 1, \\ &\vdots \\ B[h_k(x)] &\leftarrow 1. \end{aligned}$$

To determine whether a given key is likely present, we check that the corresponding bits in our array are set:

$$(B[h_1(x)] = 1) \text{ and } (B[h_2(x)] = 1) \text{ and } \cdots \text{ and } (B[h_k(x)] = 1).$$

Thus, if there are k hash functions, we might need to check up to k bits. For keys that were added, we are guaranteed that all bits are set: there can never be a false negative. But false positives are possible, if the bits were set by other keys. The standard Bloom filter does not allow us to remove keys. Bloom filters support adding keys irrespective of the size of the bit array and of the number of hash functions, but the false-positive probability increases as more entries are added, and so more bits are set.

The size of the array B is typically chosen so that a certain false-positive probability can be guaranteed up to a maximal number of entries, and the optimal parameter k is calculated. The

expected space overhead for optimal Bloom filters is 44%: it requires setting $k = -\log \epsilon$ where ϵ is the desired bound on the false-positive probability. Bloom filters can be made concurrent [39].

Blocked Bloom filters [24, 35] consist of many small Bloom filters, maybe one per CPU cache line, so that they need only one memory access per operation. However, the load of those small filters is likely to be uneven, and so for the same false-positive probability, they often need about 30% more space than standard Bloom filters. Advanced CPU instructions allow to speed up membership tests for both regular and blocked Bloom filters [32].

There are many other variations on Bloom filters including counting Bloom filters [5, 36] which support removing keys at the expense of more storage, compressed Bloom filters [27], multidimensional Bloom filters [14], Stable Bloom filters [15] and so forth.

2.2 Fingerprint Based Variants

Fingerprint-based variants store a fingerprint per key, where a fingerprint is the result of hash function h; typically, it is a word having a fixed number of bits. The membership test consists of the retrieval and comparison with the relevant fingerprints for the given key. The general intuition is as follows. For each value x in the set, we store the fingerprint $h(x)$ in a key-fingerprint data structure. Given a candidate value y, we access its fingerprint from the data structure and we compare the result with $h(y)$. Whenever y was part of the set, the fingerprints match, otherwise they are likely different with a probability that depends on the size of the fingerprint.

- *Golomb-compressed sequences* [35] store the sorted fingerprints by encoding the differences between fingerprint values. The overhead of this encoding is at least 1.5 bits per key, but it is difficult to achieve competitive speed.
- *Cuckoo filters* [20] are based on cuckoo hashing. At full capacity, and with a low false-positive probability, they use less space than Bloom filters, and membership tests are often faster. The overhead is 3 bits per key for the standard cuckoo filter, and 2 bits per key for the slower semi-sorted variant. We are not aware of a cuckoo filter implementation that supports concurrent updates though there are related cuckoo hashing concurrency strategies [26].
- *Quotient filters* [31] store fingerprints in a compact hash table. Quotient filters and cuckoo filters use a similar amount of memory.
- *Morton filters* [8] are similar to cuckoo filters, but use underloaded buckets, like Horton tables [9]. Many sparse buckets are combined into a block so that data is stored more densely.
- *Bloomier filters* [12, 13] support approximate evaluation of arbitrary functions, in addition to approximate membership queries. We are interested in a variant of the Bloomier filter [17] that can be used for approximate membership queries. We call this variant the xor filter (§ 3).

Other variants have been proposed [33, 40] but authors sometimes omit to provide and benchmark practical implementations. Dietzfelbinger and Pagh [17] observe that fingerprint techniques can be extended by storing auxiliary data with the fingerprint.

3 XOR FILTERS

Given a key x, we produce its k-bit fingerprint (noted fingerprint(x)) using a randomly chosen hash function. We assume an idealized fully independent hash function; all fingerprints are equally likely so that $P(\text{fingerprint}(x) = c) = 1/2^k$ for any x and c. This probability $\epsilon = 1/2^k$ determines the false-positive probability of our filter. We summarize our notation in Table 1.

We want to construct a map F from all possible elements to k-bit integers such that it maps all keys y from a set S to their k-bit fingerprint(x). Thus, if we pick any element of the set, it gets mapped to its fingerprint by design $F(y) = \text{fingerprint}(y)$. Any value that is not part of the filter gets mapped to a value distinct from its fingerprint with a probability $1 - \epsilon = 1 - 1/2^k$.

Table 1. Notation

U	universe of all possible elements (e.g., all strings)
S	a set of elements from universe U (also called "keys")
$\|S\|$	cardinality of the set S
B	array of k-bit values
$c = \|B\|$	size (or capacity) of the array B, we set $c = \lfloor 1.23 \cdot \|S\| \rfloor + 32$
fingerprint	random hash function mapping elements of U to k-bit values (integers in $[0, 2^k)$)
h_0, h_1, h_2	hash functions from S to integers in $[0, \lfloor c/3 \rfloor)$, $[\lfloor c/3 \rfloor, \lfloor 2c/3 \rfloor)$, $[\lfloor 2c/3 \rfloor, c)$ respectively
x xor y	bitwise exclusive-or between two values
$B[i]$	the k-bit values at index i (indexes start at zero)
ϵ	false-positive probability

We store the fingerprints in an array B with capacity c slightly larger than the cardinality of the set $|S|$ (i.e., $c \approx 1.23 \times |S|$). We randomly and independently choose three hash functions h_0, h_1, h_2 from S to consecutive ranges of integer values ($h_0 : S \to \{0, \ldots, c/3 - 1\}$, $h_1 : S \to \{c/3, \ldots, 2c/3 - 1\}$, $h_2 : S \to \{2c/3, \ldots, c - 1\}$). For example, if $c = 12$, we might have the ranges $\{0, \ldots, 3\}$, $\{4, \ldots, 7\}$, and $\{8, \ldots, 11\}$. Our goal is to have that the exclusive-or aggregate of the values in array B at the locations given by the three hash functions agree with the fingerprint ($B[h_0(x)]$ xor $B[h_1(x)]$ xor $B[h_2(x)]$ = fingerprint(x)) for all elements $x \in S$. The hash functions h_0, h_1, h_2 are assumed to be independent from the hash function used for the fingerprint.

3.1 Membership Tests

The membership-test function (Algorithm 1) calculates the hash functions h_0, h_1, h_2, then constructs the expected fingerprint from those entries in table B, and compares it against the fingerprint of the given key. If the key is in the set, the table contains the fingerprint and so it matches.

The processing time includes the computation of three hash functions as well as three random memory accesses. Though other related data structures may need fewer memory accesses, most modern processors can issue more than three memory accesses concurrently thanks to memory-level parallelism [1, 23, 34]. Hence, we should not expect the processing time to increase directly with the number of memory accesses.

Algorithm 1 Membership test: returns true if the key x is likely in S, false otherwise

Require: key $x \in U$
  return fingerprint(x) = $B[h_0(x)]$ xor $B[h_1(x)]$ xor $B[h_2(x)]$

3.2 Construction

The construction follows the algorithm from Botelho et al. [6] to build acyclic 3-partite random hypergraphs. We apply Algorithm 2 which calls Algorithm 3 one or more times until it succeeds, passing randomly chosen hash functions h_0, h_1, h_2 with each call. In practice, we pick hash functions by generating a new pseudo-random *seed*. Finally, we apply Algorithm 4.

Algorithm 3 works as follows. We initialize a (temporary) array H of sets of keys of size $\lfloor 1.23 \cdot |S| \rfloor + 32$. At the beginning, all sets are empty. Then we take each key x from the set S, and we hash

Algorithm 2 Construction

Require: set of keys S
Require: a fingerprint function
  repeat
    pick three hash functions h_0, h_1, h_2 at random, independently from the fingerprint function
  until map(S, h_0, h_1, h_2) returns success with a stack σ (see Algorithm 3)
  $B \leftarrow$ an array of size $\lfloor 1.23 \cdot |S| \rfloor + 32$ containing k-bit values (uninitialized)
  assign$(\sigma, B, h_0, h_1, h_2)$ (see Algorithm 4)
  return the array B and the hash functions h_0, h_1, h_2

it three times $(h_0(x), h_1(x), h_2(x))$. We append the key x to the three sets indicated by the three hash values (sets $H[h_0(x)], H[h_1(x)], H[h_2(x)]$). Most sets in the table H contain multiple keys, but almost surely some contain exactly one key. We keep track of the sets containing just one key. Repeatedly, we pick one such location, append it to the output stack together with the key x it contains; each time we remove the key x from its three locations $(h_0(x), h_1(x), h_2(x))$. The process either terminates with a stack containing all of the keys in which case we have a success, or with a failure.

The probability of success approaches 100% if the set is large [28]. For sets of size 10^7, Botelho et al. [6] found that the probability is almost 1. For smaller sets, we experimentally found that the estimated probability is always greater than 0.8 with $c = 1.23 \cdot |S| + 32$, as shown in Fig. 1.

Algorithm 3 runs in linear time with respect to the size of the input set S as long as adding and removing a key x from a set in H is done in constant time. Indeed, each key x of S is initially added to three sets in H and removed at most once from the same three sets.

In practice, if the keys in S are integer values or other fixed-length objects, we can implement the sets using an integer-value counter and a fixed-length mask (both initialized with zeros). When adding a key, we increment the counter and compute the exclusive-or of the key with the mask, storing the result as the new mask. We similarly remove a key by decrementing the counter and computing the same exclusive-or. Even when the set is made of large or variable-length elements, it may still be practical to represent them as small fixed-length (e.g., 64-bit or 128-bit) integers by hashing: it only comes at the cost of introducing a small error when two hash values collide, an improbable event that may only minutely increase the false-probability probability.

We find it interesting to work backward through the second part of Algorithm 3 when it succeeds. At the end, we have that H is made of only empty sets. We then pick the last key x from the stack and add it to sets at locations $h_0(x), h_1(x), h_2(x)$. Then, still working in reverse, when we pick a key x and its index i, we have that $H[i]$ is empty right before we add x to $H[h_0(x)], H[h_1(x)], H[h_2(x)]$. Thus, by construction, i is different from $h_0(x'), h_1(x'), h_2(x')$ for all keys x' encountered so far (still working in reverse).

To construct the xor filter, we allocate an array B large enough to store $\lfloor 1.23 \cdot |S| \rfloor + 32$ fingerprints. We iterate over the keys and their indexes in the reverse order, compared to how they were identified in the "Mapping Step" (Algorithm 3). For each key, there are three corresponding locations $h_0(x), h_1(x), h_2(x)$ in the table B; the index associated with the key is one of $h_0(x), h_1(x), h_2(x)$. We set the value of $B[i]$ so that $B[h_0(x)]$ xor $B[h_1(x)]$ xor $B[h_2(x)]$ = fingerprint(x). We repeat this for each key. Each key is processed once.

By our construction, an entry in B is modified at most once. After we modify an entry $B[i]$, then none of the values $B[h_0(x)]$, $B[h_1(x)]$, $B[h_2(x)]$ will ever be modified again. This follows by our argument where we work through Algorithm 3 in reverse: i is different from $h_0(x'), h_1(x'), h_2(x')$

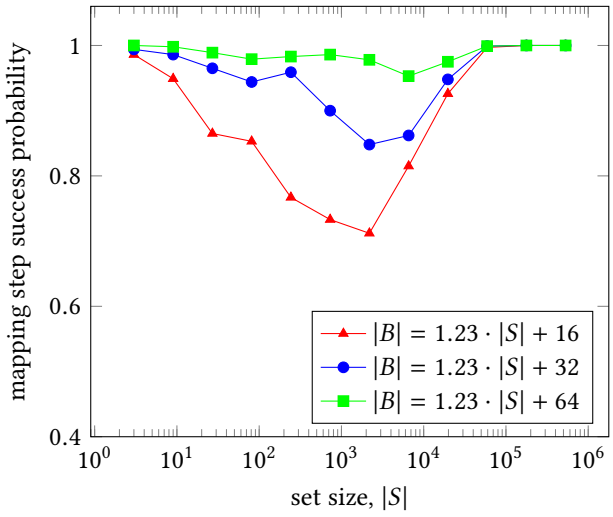


Fig. 1. Probability of mapping step, found experimentally with 1000 randomly generated sets.

Algorithm 3 Mapping Step (map)

Require: set of keys S, k-bit integer-valued hash functions h_0, h_1, h_2.
  let $c \leftarrow \lfloor 1.23 \cdot |S| \rfloor + 32$
  $H \leftarrow$ an array of size c containing a set of keys (values from S), initially empty
  for all x in S **do**
    append x to $H[h_0(x)]$
    append x to $H[h_1(x)]$
    append x to $H[h_2(x)]$
  end for
  $Q \leftarrow$ initially empty queue
  for $i = 0$ **to** $|H|$ **do**
    if the set $H[i]$ contains a single key **then** add i to Q **endif**
  end for
  $\sigma \leftarrow$ initially empty stack
  while queue Q is not empty **do**
    remove an element i from the queue Q
    if the set $H[i]$ contains a single key **then**
      let x be the sole value in the set $H[i]$
      push the pair (x, i) on the stack σ
      for $j = 0$ **to** 2 **do**
        remove x from the set $H[h_j(x)]$
        if the set $H[h_j(x)]$ contains a single key **then** add $h_j(x)$ to Q **endif**
      end for
    end if
  end while
  return success and the stack σ **if** $|\sigma| = |S|$, **else** return failure

for all keys x' encountered so far. Thus, our construction is correct: we have that

$$B[h_0(x)] \text{ xor } B[h_1(x)] \text{ xor } B[h_2(x)] = \text{fingerprint}(x)$$

for all keys x in S at the end of Algorithm 4.

Algorithm 4 Assigning Step (assign)

Require: σ, target array for fingerprint data B, hash functions h_0, h_1, h_2
  for (x, i) in stack σ **do**
    $B[i] \leftarrow 0$
    $B[i] \leftarrow \text{fingerprint}(x) \text{ xor } B[h_0(x)] \text{ xor } B[h_1(x)] \text{ xor } B[h_2(x)]$
  end for

3.3 Space Optimization: Xor+ Filter

About 19% of the entries in table B are empty: for each 100 keys, we need 123 entries, and 23 are empty. For transmission, much of this empty space can be saved as follows: before sending B, send a bit array that contains '0' for empty entries and '1' for occupied entries. Then we only send the data of the occupied entries. If we use $k = 8$ bits, the regular xor filter needs $8 \times 1.23 = 9.84$ bits per entry, which we can compress in this way to $8 + 1.23 = 9.23$ bits per entry. If space usage at runtime is more important than query speed, compression can be used at runtime. We can get a constant time access using a rank data structure such as Rank9 [38], at the expense of a small storage overhead (≈25%), or poppy [41] for an even smaller overhead (≈3%) at the expense of some speed.

By changing the construction algorithm slightly, we can move most of the empty entries to the last third of the table B. To do so, we change the mapping algorithm so that three queues are used instead of one: one for each hash function—each hash function represents a third of the table B. We then process entries of the first two queues until those are empty, before we process entries from the third queue. Experimentally, we find that 36% of the entries in the last third of table B are empty on average. If the rank data structure is then only constructed for this part of the table, space can be saved without affecting the membership-test performance as much, as only one rank operation is needed. We refer to this algorithm as "xor+ filter", using Rank9 as the default rank data structure. With the fingerprint size in bits k, it needs $k \times 1.23 \times 2/3$ bits per key for the first two thirds of the table B, $k \times 1.23 \times 1/3 \times (1 - 0.36)$ for the last third, plus $1.23 \times 1/3 \times 1.25$ for the Rank9 data structure. In summary, xor+ filters use $1.0824k + 0.5125$ bits per entry as opposed to $1.23k$ bit per entry for xor filters.

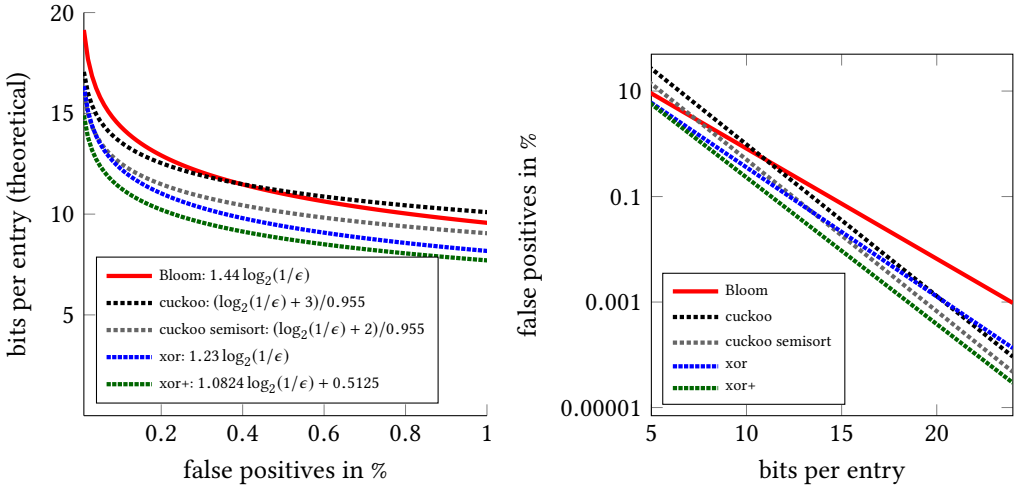
3.4 Space Comparison

We compare the space usage of some of the most important filters in Fig. 2. Bloom filters are more space efficient than cuckoo filters at a false-positive probability of 0.4% or higher.

For very low false-positive probabilities (5.6×10^{-6}), cuckoo filters at full capacity use less space than xor filters. However, we are not aware of any system that uses such a low false-positive probability: most systems seem to use between 8 and 20 bits per key [16, 37]. Thus we expect xor and xor+ filters to use less memory in practice.

4 EXPERIMENTS

We follow Fan et al.'s testing procedure [20]; we started from their software project [19]. Like them, we use 64-bit keys as set elements. We build a filter based on a set of 10M or 100M keys. We build a distinct set made of 10M queried keys. This set of queried keys is created by mixing some of the



(a) Space vs. false-positive probability

(b) False-positive probability vs. space

Fig. 2. Theoretical memory usage for Bloom filters (optimized for space), cuckoo filter (at max. capacity) and xor filters given a desired bound on the false-positive probability.

keys from the original set, and some keys not present in the original set. We use different fractions (e.g., 0%, 25%, 50%, 75% and 100%) of the keys in the original set. The benchmark counts the number of queried keys that are possibly in the set according to the filter. The benchmark is single threaded and calls the membership-test functions with different keys in a loop. We disable inlining of the functions to prevent compilers from unduly optimizing the benchmark which counts the number of matching keys.

We run benchmarks on Intel processors with Skylake microarchitecture: an Intel i7-6700 processor running at 3.4 GHz, with 8 MB of L3 cache. The software is compiled with the GNU GCC 8.1.0 compiler to a 64-bit Linux executable with the flags `-O3 -march=native`. For each filter, we run 3 tests, and report the median. Our error margin is less than 3%. The C++ source code of the filter implementations and the benchmark is available[1]. For some algorithms including all the xor and xor+ filters, we have also implemented Java versions[2] and well as a Go version[3] and a pure C version[4], but the benchmarks are using C++.

For all implementations, we use a randomly seeded Murmur finalizer [21] to compute the fingerprint from the key, as described in Algorithm 5. We choose this option instead of faster alternatives so that even non-random keys work well and do not result in higher-than-expected false-positive probabilities, or construction failure in the case of the cuckoo filter. For our tests, we use pseudo-randomly generated keys; we also tested with sequentially generated keys and found no statistically significant difference compared to using random keys after introducing the Murmur finalizer.

All implementations need to reduce a hash value x to the range $\{0, \ldots, m-1\}$ where m is not necessarily a power of two. Where this is needed, we do not use the relatively slow modulo

[1]https://github.com/FastFilter/fastfilter_cpp (release 1.0), see "Benchmarking" section.
[2]https://github.com/FastFilter/fastfilter_java
[3]https://github.com/FastFilter/xorfilter
[4]https://github.com/FastFilter/xor_singleheader

operation $x \mod m$ for performance reasons. Instead, starting with 32-bit values x and m and computing their full 64-bit product $x \times m$, we use the faster multiply-shift combination $(x \times m) \div 2^{32} = (x \times m) >> 32$ [25].

Algorithm 5 64-bit hash function

Require: key x, seed s
  $h \leftarrow x + s$
  $h \leftarrow (h \text{ xor } (h >> 33)) * 0\text{xff51afd7ed558ccd}$
  $h \leftarrow (h \text{ xor } (h >> 33)) * 0\text{xc4ceb9fe1a85ec53}$
  return $h \text{ xor } (h >> 33)$

4.1 Filter Implementations

We run tests against the following filters:

- Bloom filter: We implemented the standard Bloom filter algorithm with configurable false-positive probability (FPP) and size. We test with 8, 12, and 16 bits per key, and the respective number of hash functions k that are needed for the lowest false-positive probability. For fast construction and membership test, we hash only once with a 64-bit function, treated as two 32-bit values $h_1(k)$ and $h_2(k)$. The Bloom filter hash functions are $g_i(k) = h_1(k) + i \cdot h_2(k)$ for $i = 0, \ldots, k-1$.
- Blocked Bloom filter: We use a highly optimized blocked Bloom filter from Apache Impala[5], which is also used in the cuckoo filter software project [19]. We modified it so the size is flexible and not restricted to 2^n. It is designed for Intel AVX2 256-bit operations; it is written using low-level Intel intrinsic functions. The advantage of this algorithm is the membership-test speed: each membership test is resolved from one cache line only using few instructions. The main disadvantage is that it is larger than regular Bloom filters.
- Cuckoo filter (C): We started with the cuckoo filter implementation from the original authors [19]. We reduce the maximum load from 0.96 to 0.94, as otherwise construction occasionally fails. The reduced maximum load is apparently the recommended workaround suggested by the cuckoo filter authors. Though it is outside our scope to evaluate whether it is always a reliable fix, it was sufficient in our case. This reduction of the maximum load slightly worsens (≈2%) the memory usage of cuckoo filters. In the original reference implementation [20], the size of the filter is restricted to be a power of two, which means up to 50% of the space is unused. Wasting so much space seems problematic, especially since it does not improve the false-positive probability. Therefore, we modified it so the size is flexible and not restricted to 2^n. This required us to slightly change the calculation for the alternate location $l_2(x)$ for a key x from the first location $l_1(x)$ and the fingerprint $f(x)$. Instead of $l_2(x) = l_1(x) \text{ xor } h(f(x))$ as in Fan et al. [20], we use $l_2(x) = \text{bucketCount} - l_1(x) - h(f(x))$, and if the result is negative we add bucketCount. We use 12-bit and 16-bit fingerprints.
- Cuckoo semi-sorted (Css): We use the semi-sorted cuckoo filter reference implementation, modified in the same way as the regular cuckoo filter. From the original Fan et al. [20] source release, we could only get one variant to work correctly, the version with a fingerprint size of 13 bits. Other versions have a non-zero false negative probability.
- Golomb-compressed sequence (GCS): Our implementation uses an average bucket size of 16, and Golomb Rice coding. We use a fingerprint size of 8 bits.
- Xor: Our xor and xor+ filters as described in § 3. We use 8-bit and 16-bit fingerprints.

[5] https://impala.apache.org

Table 2. Construction time in nanoseconds per key, rounded to 10 nanoseconds.

algorithm	10 million keys	100 million keys
Blocked Bloom	10 ns/key	20 ns/key
Bloom 8	40 ns/key	70 ns/key
Bloom 12	60 ns/key	90 ns/key
Bloom 16	90 ns/key	130 ns/key
Cuckoo semiSort 13	130 ns/key	200 ns/key
Cuckoo 12	80 ns/key	130 ns/key
Cuckoo 16	90 ns/key	120 ns/key
GCS	160 ns/key	190 ns/key
Xor 8	110 ns/key	130 ns/key
Xor 16	120 ns/key	130 ns/key
Xor+ 8	160 ns/key	180 ns/key
Xor+ 16	160 ns/key	180 ns/key
(Sorting the keys)	80 ns/key	90 ns/key

4.2 Construction Performance

We present the construction times for 10 million and 100 million keys in Table 2. All construction algorithms are single-threaded; we did not investigate multi-threaded construction. For reference, we also present the time needed to sort the 64-bit keys using the C++ standard sorting algorithm (`std::sort`), on the same platform.

During construction, the blocked Bloom filter is clearly the fastest data structure. For the 100 million case, the semi-sorted variant of the cuckoo filter is the slowest. Construction of the xor filter with our implementation is roughly half as fast as the cuckoo filter and the Bloom filter, which have similar performance.

4.3 Query Time Versus Space Overhead

We present the performance numbers for the case where 25% of the searched entries are in the set in Fig. 3, and in the case where all searched entries are in the set in Fig. 4. The results are presented in tabular form in Table 3, where we include the Golomb-compressed sequence.

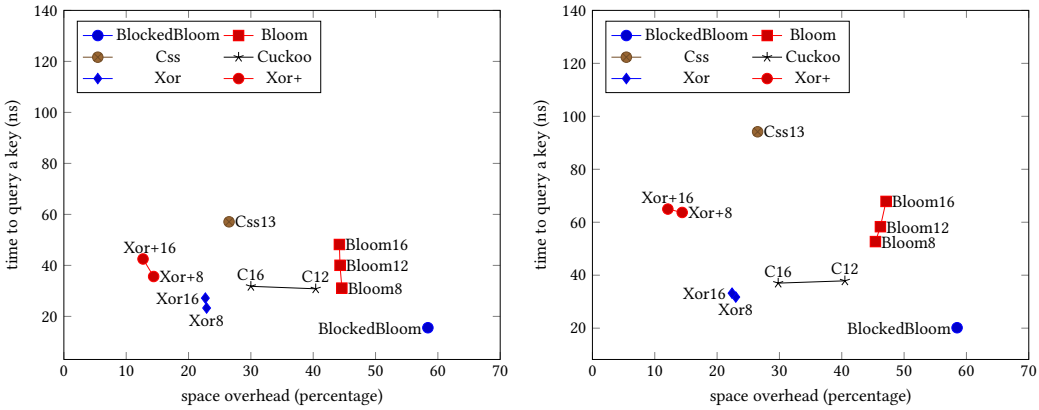
Unlike xor and cuckoo filters, the Bloom filter membership-test timings are sensitive to the fraction of keys present in the set. When an entry is not in the set, only a few bits need to be accessed, until the query function finds an unset bit and returns. The Bloom filter is slower if an entry exists in the set, as it has to check all bits; this is especially the case for low false-positive probabilities. See Fig. 4.

Ignoring query time, Fig. 5 shows that Cuckoo 12 (C12) has memory usage that is close to Bloom filters. The cuckoo filter only uses much less space than Bloom filters for false-positive probabilities well below 1% (Cuckoo 16 or C16). In our experiments, the cuckoo filter, and the slower semi-sorted cuckoo filter (Css), always use more space than the xor filter. These experimental results match the theoretical results presented in Fig. 2.

The xor filter provides good query-time performance while using little space, even for moderate false-positive probabilities.

Table 3. Membership-test benchmark results. Timings are in nanosecond per query.

	(a) 10M keys			(b) 100M keys		
Name	Time (ns)	Bits/key	FPP	Time (ns)	Bits/key	FPP
Blocked Bloom	16	10.7	0.939	20	10.7	0.941
Bloom 8	31	8.0	2.161	53	8.0	2.205
Bloom 12	40	12.0	0.313	58	12.0	0.339
Bloom 16	48	16.0	0.046	68	16.0	0.053
Cuckoo semiSort 13	57	12.8	0.092	94	12.8	0.092
Cuckoo 12	31	12.8	0.183	38	12.8	0.184
Cuckoo 16	32	17.0	0.012	37	17.0	0.011
GCS	137	10.0	0.389	220	10.0	0.390
Xor 8	23	9.8	0.389	32	9.8	0.391
Xor 16	27	19.7	0.002	33	19.7	0.001
Xor+ 8	36	9.2	0.390	64	9.2	0.389
Xor+ 16	43	17.8	0.002	65	17.8	0.002



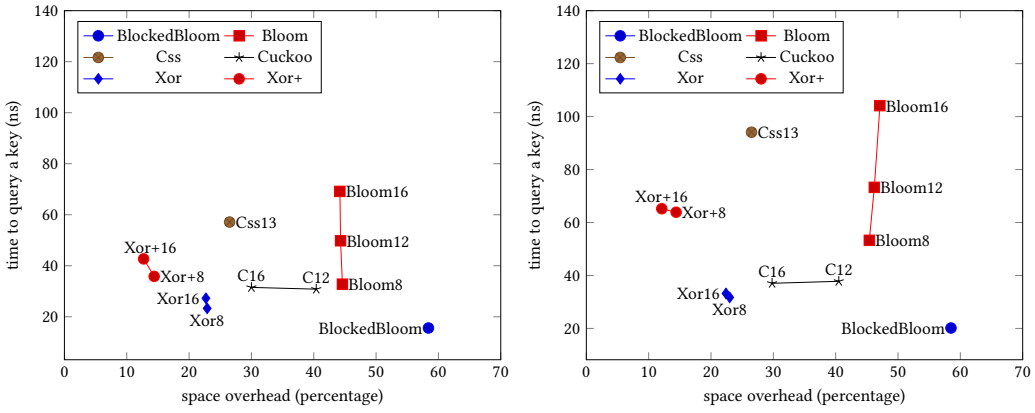
(a) 10M keys (b) 100M keys

Fig. 3. Query time vs. space overhead, 25% find

4.4 Discussion

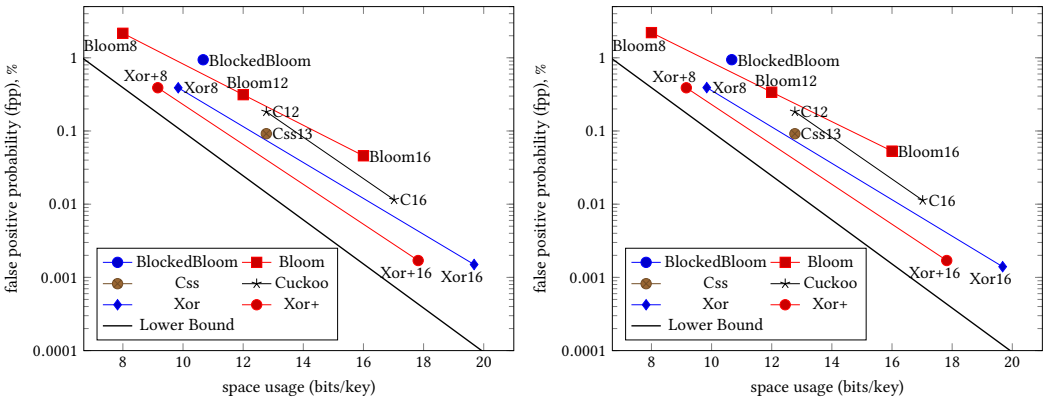
We attribute the good membership-test performance of xor filters mainly to the following reasons. Xor filters use exactly 3 memory accesses, independent of the false-positive probability. These memory accesses can be executed in parallel by the memory subsystem. The number of instructions meanwhile is small and there are no branches.

For a false-positive probability of 1%, the standard Bloom filter needs more memory accesses for a match, and even more so for lower false-positive probabilities. The Bloom filter uses between 41 and 105 instructions per key, depending on the number of set bits set and false-positive probability. For a miss (if the key is not in the set), on average fewer memory accesses are needed, but there might be mispredicted branches with accompanying penalties.



(a) 10M keys (b) 100M keys

Fig. 4. Query time vs. space overhead, 100% find



(a) 10M keys (b) 100M keys

Fig. 5. FPP vs. space usage in bits/key, log scale FPP

The cuckoo filter uses exactly 2 memory accesses, and 66 to 68 instructions per key (depending on fingerprint size). The xor filter uses exactly 3 memory accesses, but only about 48 instructions per key. Processors execute complex machine instructions using low-level instructions called μops. A processor like our Skylake can support up to 10 outstanding memory requests per core, limited by an instruction reorder buffer of 200 μops. In the absence of mispredicted branches and long dependency chains, the capacity of the instruction buffer becomes a limitation [3]. It is likely the reason why the cuckoo filter and the xor filter have similar membership-test performance. That is, while the cuckoo filter has fewer memory accesses, it generates more instructions which makes it harder for the processor to fetch as many memory requests as it could.

In our benchmarks, the blocked Bloom filter is the only algorithm that is clearly faster than the xor filter. This is most likely due to only having one memory access, and highly optimized code, using SIMD instructions specific to recent x64 processors. It needs fewer memory accesses

Table 4. Construction time, memory usage and query time (25% of the entries in the set) for original and compact cuckoo filters with 10 million keys.

Name	Construction Time	Memory	Query Time
Cuckoo 12	80 ns/key	12.8 bits/key	31 ns/key
Cuckoo 12 (original)	40 ns/key	20.1 bits/key	30 ns/key
Cuckoo 16	90 ns/key	17.0 bits/key	32 ns/key
Cuckoo 16 (original)	40 ns/key	26.8 bits/key	28 ns/key

and fewer instructions than its competitors. It might be difficult to implement a similarly efficient approach in a higher-level language like Java, or using solely portable code. If memory usage or low false-positive probability are a primary concern, the blocked Bloom filter may not be a good choice.

While an xor filter is immutable, we believe that it is not a limitation for many important applications; competitive alternatives all have limited mutability in any case. Approximate filters that support fast mergers or additions (e.g., Bloom filters) require the original filters to have extra capacity. The update may even fail in the case of Cuckoo filters. Re-building the filter can maintain an optimal size. In multithreaded systems, immutability avoids the overhead of synchronization mechanisms to maintain concurrency.

5 CONCLUSION AND FUTURE WORK

Xor filters are typically faster than Bloom filters, and they save about 15% in memory usage. While the construction of xor filters is slower than Bloom filters ($\approx 2\times$), we expect that the construction is a one-time cost amortized over many queries. Future work could consider batched queries [8] to improve performance. It might also be possible to partially parallelize the construction of the filters.

A ORIGINAL VERSUS COMPACT CUCKOO FILTERS

In Table 4, we compare the original implementations of cuckoo filters which require that the filter size be a power of two, with our more compact implementation. Given 10 million keys, the memory usage of cuckoo filters using the original implementation is not competitive. However, the construction time is reduced with an overallocated filter because hash collisions are less frequent. Similarly, the query times (25% of the entries in the set) are about 10% smaller in the original implementation. However, if we choose a number of keys near a power of two (e.g., 31.5 million keys), the original and compact implementations have nearly the same memory usage, construction times, and query speeds.

B QUOTIENT AND MORTON FILTERS

We consider quotient filters [31] (CQF) experimentally. We use the reference implementation [30]. The implementation relies on assembly code optimized for recent x64 processors. As with cuckoo filters, the original implementation requires that the capacity be a power of two. Table 5 shows that the query time of the reference quotient-filter implementation is several times the query time of competitive approaches like cuckoo or xor filters. We also consider Morton filters [8] with the reference implementation [7]. Morton filters answer one-at-a-time queries at half the speed of 8-bit xor filters, despite similar false-positive probabilities and memory usage.

Table 5. Performance of counting quotient filters (CQF) and Morton filters. We include Xor 8 results for comparison. For queries, we report the results corresponding to 25% of the entries being in the set.

Name	Query Time (ns/key)	Bits/key	FPP	volume
Xor 8	23	9.8	0.39	10M
CQF	64	17.0	0.23	10M
Morton	47	11.7	0.31	10M
Xor 8	32	9.8	0.39	100M
CQF	88	13.6	0.29	100M
Morton	65	11.7	0.31	100M

ACKNOWLEDGMENTS

We are grateful to J. Apple for his feedback.

REFERENCES

[1] Shoaib Akram, Jennifer B Sartor, Kenzo Van Craeynest, Wim Heirman, and Lieven Eeckhout. 2016. Boosting the priority of garbage: Scheduling collection on heterogeneous multicore processors. *ACM Transactions on Architecture and Code Optimization (TACO)* 13, 1 (2016), 4.

[2] Paulo Sérgio Almeida, Carlos Baquero, Nuno Preguiça, and David Hutchison. 2007. Scalable Bloom filters. *Inform. Process. Lett.* 101, 6 (2007), 255–261.

[3] Scott Beamer, Krste Asanovic, and David Patterson. 2015. Locality exists in graph processing: Workload characterization on an Ivy Bridge server. In *2015 IEEE International Symposium on Workload Characterization*. IEEE, 56–65.

[4] Burton H. Bloom. 1970. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM* 13, 7 (July 1970), 422–426.

[5] Flavio Bonomi, Michael Mitzenmacher, Rina Panigrahy, Sushil Singh, and George Varghese. 2006. An Improved Construction for Counting Bloom Filters. In *Proceedings of the 14th Conference on Annual European Symposium - Volume 14 (ESA'06)*. Springer-Verlag, London, UK, UK, 684–695.

[6] Fabiano C. Botelho, Rasmus Pagh, and Nivio Ziviani. 2007. Simple and Space-efficient Minimal Perfect Hash Functions. In *Proceedings of the 10th International Conference on Algorithms and Data Structures (WADS'07)*. Springer-Verlag, Berlin, Heidelberg, 139–150.

[7] Alex D. Breslow and Nuwan S. Jayasena. 2018. Morton Filter. https://github.com/AMDComputeLibraries/morton_filter.git, commit: 837484dad7c402db6ff08688590c4cee9c152682.

[8] Alex D. Breslow and Nuwan S. Jayasena. 2018. Morton Filters: Faster, Space-efficient Cuckoo Filters via Biasing, Compression, and Decoupled Logical Sparsity. *Proc. VLDB Endow.* 11, 9 (May 2018), 1041–1055.

[9] Alex D. Breslow, Dong Ping Zhang, Joseph L. Greathouse, Nuwan Jayasena, and Dean M. Tullsen. 2016. Horton Tables: Fast Hash Tables for In-memory Data-intensive Computing. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '16)*. USENIX Association, Berkeley, CA, USA, 281–294.

[10] Andrei Broder and Michael Mitzenmacher. 2004. Network applications of Bloom filters: A survey. *Internet mathematics* 1, 4 (2004), 485–509.

[11] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2008. BigTable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.* 26, 2, Article 4 (June 2008), 26 pages.

[12] Denis Charles and Kumar Chellapilla. 2008. Bloomier filters: A second look. In *European Symposium on Algorithms*. Springer, 259–270.

[13] Bernard Chazelle, Joe Kilian, Ronitt Rubinfeld, and Ayellet Tal. 2004. The Bloomier Filter: An Efficient Data Structure for Static Support Lookup Tables. In *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '04)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 30–39.

[14] Adina Crainiceanu and Daniel Lemire. 2015. Bloofi. *Inf. Syst.* 54, C (Dec. 2015), 311–324.

[15] Fan Deng and Davood Rafiei. 2006. Approximately Detecting Duplicates for Streaming Data Using Stable Bloom Filters. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data (SIGMOD '06)*. ACM, New York, NY, USA, 25–36.

[16] Sarang Dharmapurikar, Praveen Krishnamurthy, Todd Sproull, and John Lockwood. 2003. Deep packet inspection using parallel Bloom filters. In *High performance interconnects, 2003. proceedings. 11th symposium on*. IEEE, 44–51.

[17] Martin Dietzfelbinger and Rasmus Pagh. 2008. Succinct Data Structures for Retrieval and Approximate Membership (Extended Abstract). In *Automata, Languages and Programming*, Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 385–396.

[18] Ozgun Erdogan and Pei Cao. 2007. Hash-AV: fast virus signature scanning by cache-resident filters. *International Journal of Security and Networks* 2, 1-2 (2007), 50–59.

[19] Bin Fan and David G. Andersen. 2013–2017. Cuckoo Filter. https://github.com/efficient/cuckoofilter, commit: aac6569cf30f0dfcf39edec1799fc3f8d6f594da.

[20] Bin Fan, David G. Andersen, Michael Kaminsky, and Michael D. Mitzenmacher. 2014. Cuckoo Filter: Practically Better Than Bloom. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies (CoNEXT '14)*. ACM, New York, NY, USA, 75–88.

[21] Dmytro Ivanchykhin, Sergey Ignatchenko, and Daniel Lemire. 2017. Regular and almost universal hashing: an efficient implementation. *Software: Practice and Experience* 47, 10 (2017), 1299–1323.

[22] Arpith Jacob and Maya Gokhale. 2007. Language Classification Using N-grams Accelerated by FPGA-based Bloom Filters. In *Proceedings of the 1st International Workshop on High-performance Reconfigurable Computing Technology and Applications: Held in Conjunction with SC07 (HPRCTA '07)*. ACM, New York, NY, USA, 31–37.

[23] Christopher Jonathan, Umar Farooq Minhas, James Hunter, Justin Levandoski, and Gor Nishanov. 2018. Exploiting coroutines to attack the killer nanoseconds. *Proceedings of the VLDB Endowment* 11, 11 (2018), 1702–1714.

[24] Harald Lang, Thomas Neumann, Alfons Kemper, and Peter Boncz. 2019. Performance-optimal filtering: Bloom overtakes Cuckoo at high throughput. *Proceedings of the VLDB Endowment* 12, 5 (2019), 502–515.

[25] Daniel Lemire. 2019. Fast Random Integer Generation in an Interval. *ACM Trans. Model. Comput. Simul.* 29, 1, Article 3 (Jan. 2019), 12 pages.

[26] Xiaozhou Li, David G. Andersen, Michael Kaminsky, and Michael J. Freedman. 2014. Algorithmic Improvements for Fast Concurrent Cuckoo Hashing. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys '14)*. ACM, New York, NY, USA, Article 27, 14 pages.

[27] Michael Mitzenmacher. 2002. Compressed Bloom Filters. *IEEE/ACM Trans. Netw.* 10, 5 (Oct. 2002), 604–612.

[28] Michael Molloy. 2005. Cores in Random Hypergraphs and Boolean Formulas. *Random Struct. Algorithms* 27, 1 (Aug. 2005), 124–135.

[29] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33, 4 (1996), 351–385.

[30] Prashant Pandey, Michael A. Bender, Rob Johnson, and Rob Patro. 2017. Counting Quotient Filter (CQF). https://github.com/splatlab/cqf.

[31] Prashant Pandey, Michael A. Bender, Rob Johnson, and Rob Patro. 2017. A General-Purpose Counting Filter: Making Every Bit Count. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*. ACM, New York, NY, USA, 775–787.

[32] Orestis Polychroniou and Kenneth A. Ross. 2014. Vectorized Bloom Filters for Advanced SIMD Processors. In *Proceedings of the Tenth International Workshop on Data Management on New Hardware (DaMoN '14)*. ACM, New York, NY, USA, Article 6, 6 pages.

[33] Ely Porat. 2009. An Optimal Bloom Filter Replacement Based on Matrix Solving. In *Computer Science - Theory and Applications*, Anna Frid, Andrey Morozov, Andrey Rybalchenko, and Klaus W. Wagner (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 263–273.

[34] Georgios Psaropoulos, Thomas Legler, Norman May, and Anastasia Ailamaki. 2017. Interleaving with coroutines: a practical approach for robust index joins. *Proceedings of the VLDB Endowment* 11, 2 (2017), 230–242.

[35] Felix Putze, Peter Sanders, and Johannes Singler. 2010. Cache-, Hash-, and Space-efficient Bloom Filters. *J. Exp. Algorithmics* 14, Article 4 (Jan. 2010), .78 pages.

[36] Ori Rottenstreich, Yossi Kanizo, and Isaac Keslassy. 2014. The Variable-increment Counting Bloom Filter. *IEEE/ACM Trans. Netw.* 22, 4 (Aug. 2014), 1092–1105.

[37] Russell Sears and Raghu Ramakrishnan. 2012. bLSM: A General Purpose Log Structured Merge Tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD '12)*. ACM, New York, NY, USA, 217–228.

[38] Sebastiano Vigna. 2008. Broadword implementation of rank/select queries. In *International Workshop on Experimental and Efficient Algorithms*. Springer, 154–168.

[39] I. Voras and M. Žagar. 2010. Adapting the Bloom filter to multithreaded environments. In *Melecon 2010 - 2010 15th IEEE Mediterranean Electrotechnical Conference*. 1488–1493.

[40] Sean A. Weaver, Hannah J. Roberts, and Michael J. Smith. 2018. XOR-Satisfiability Set Membership Filters. In *Theory and Applications of Satisfiability Testing – SAT 2018*, Olaf Beyersdorff and Christoph M. Wintersteiger (Eds.). Springer International Publishing, Cham, 401–418.

[41] Dong Zhou, David G Andersen, and Michael Kaminsky. 2013. Space-efficient, high-performance rank and select structures on uncompressed bit sequences. In *International Symposium on Experimental Algorithms*. Springer, 151–163.