RESULTS reproduced

A four-dimensional Analysis of Partitioned Approximate Filters

Tobias Schmidt*
TU Munich
tobias.schmidt@in.tum.de

Maximilian Bandle*
TU Munich
bandle@in.tum.de

Jana Giceva
TU Munich
jana.giceva@in.tum.de

ABSTRACT

With today's data deluge, approximate filters are particularly attractive to avoid expensive operations like remote data/disk accesses. Among the many filter variants available, it is non-trivial to find the most suitable one and its optimal configuration for a specific use-case. We provide open-source implementations for the most relevant filters (Bloom, Cuckoo, Morton, and Xor filters) and compare them in four key dimensions: the false-positive rate, space consumption, build, and lookup throughput.

We improve upon existing state-of-the-art implementations with a new optimization, radix partitioning, which boosts the build and lookup throughput for large filters by up to 9x and 5x. Our in-depth evaluation first studies the impact of all available optimizations separately before combining them to determine the optimal filter for specific use-cases. While register-blocked Bloom filters offer the highest throughput, the new Xor filters are best suited when optimizing for small filter sizes or low false-positive rates.

PVLDB Reference Format:
Tobias Schmidt, Maximilian Bandle, and Jana Giceva. A four-dimensional Analysis of Partitioned Approximate Filters. PVLDB, 14(11): 2355 - 2368, 2021.
doi:10.14778/3476249.3476286

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/tum-db/partitioned-filters.

1 INTRODUCTION

As the volume of generated and processed data increases [41], *efficient access to only the relevant items* is necessary. The goal is both to achieve good performance for the executing workload and to reduce overall pressure on data movement channels by only loading necessary data from storage or over the network.

In this context, approximate filters are particularly useful as they compactly represent the membership of elements in a set, however, at the cost of having false positives. More specifically, the filter always reports contained items as members, i.e., there are no false negatives. For items that are not in the set, the filter returns incorrect results with a certain probability, the false-positive rate ε. Small filters can fit in a higher level of the storage (memory) hierarchy, leading to faster access times and lower bandwidth consumption, putting less pressure on the rest of the system's resources.

*Both authors contributed equally to this research.

Proceedings of the VLDB Endowment, Vol. 14, No. 11 ISSN 2150-8097.
doi:10.14778/3476249.3476286

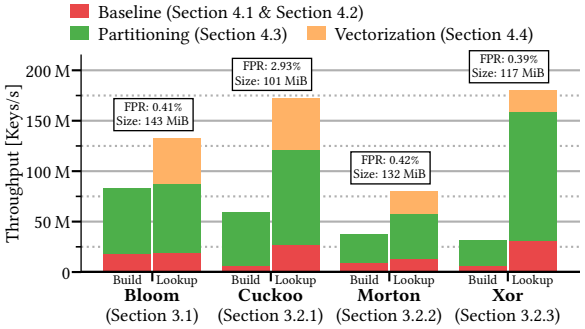


Figure 1: Speedup gained by optimizing insert and lookup operations for 100 M keys.

It is therefore not a surprise that filters are often used to speed up applications. Log-structured merge (LSM) trees, for instance, check the filter before fetching a page from disk [31]. In databases, filters improve query execution through selective join pushdown, which drops tuples not needed for probing early in the pipeline [28]. Other applications include distributed joins or network applications, where filters reduce the amount of transferred data [13, 27].

We distinguish between two filter families: Bloom filter variants and fingerprint filters. The Bloom filter accesses several bits in a bitmap on lookup or insert [6] and is the most popular filter today [32]. However, fingerprint filters have recently emerged, which store small *signatures of the key* in a hash table-like structure. They are smaller in size and have lower false-positive rates than Bloom filters, at the cost of higher access latencies. Some of the more notable fingerprint filters are the Quotient [5], the Cuckoo [21], the Morton filter [11], and more recently, the Xor filter [24].

With this plethora of available alternatives, it is unclear *which filter* to use *when*. Very often, one has to consider multiple dimensions that are relevant to the use-case in mind. Thus, in this paper, we evaluate the four most promising filters — Bloom, Cuckoo, Morton, and Xor filters — on the following four key dimensions:

- **False-positive rate (FPR):** it affects the application's performance and hints at the extra bandwidth overhead on shared I/O resources.
- **Space consumption:** we want to minimize the precious space in caches/DRAM to store auxiliary data structures.
- **Lookup performance:** it directly affects the performance of the application.
- **Build performance:** the time it takes to construct the filter.

All four aspects are closely interlinked, and improving one dimension may result in a decline in another (e.g., reducing the FPR may necessitate an increase in size). Which dimension to prioritize when choosing the most suitable filter is application-specific. On the one hand, LSM-Trees primarily aim to reduce the FPR to avoid unnecessary expensive I/O operations while limiting the memory

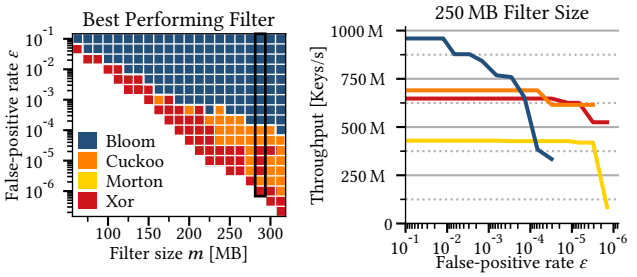


Figure 2: Performance with 100 M elements (10 threads).

assigned to the filters [17]. On the other hand, an in-memory join cares more about lookup performance than filter size.

Unfortunately, a fair comparison between the filters is not possible today, as most authors introduced different optimizations in their implementations [24, 28, 48]. We therefore provide *open-source implementations* for all filters and integrate all relevant optimizations. Furthermore, we apply a new optimization, *radix partitioning* [7, 46], that considerably increases build/lookup performance for all filters when their size exceeds the last-level cache. Figure 1 shows the performance of state-of-the-art baselines compared to the acceleration we get with vectorization and partitioning. However, as not all optimizations are always applicable and usually entail tradeoffs, we also investigate *when* to apply *which optimization*.

In the second part of the paper, we compare the performance of several Bloom filter variants and the three fingerprint filters. Figure 2 (left) shows which filter performs best for a given FPR and filter size. Figure 2 (right) shows how each filter performs for a fixed size. Here, an LSM-Tree would prefer one of the fingerprint filters (e.g., Cuckoo or Xor) as they offer low FPRs with good performance under a strict memory budget. In contrast, in-memory joins would favor the Bloom filter, as it achieves twice the performance and the cost of a false positive is rather inexpensive [28].

The rest of this paper is organized as follows. We first give a brief overview of prior work before introducing the filters and their implementation in Section 3. In Section 4, we describe the optimizations we applied and evaluate the impact of each optimization on the baseline. Finally, in Section 5, we evaluate the filters for the four key dimensions and propose guidelines for choosing the right filter and which optimizations to enable.

2 RELATED WORK

The Bloom filter was first presented in 1970 [6]. Since then, more than 60 variants have been proposed [32]. Most of them extend the functionality to support deleting keys [15, 22, 44] or resizing the filter [2, 25]. Others tailor it to specific use cases, such as network applications [23, 34, 45]. However, more functionality usually results in reduced throughput, larger filter size, or higher FPRs. We, instead, focus on variants that improve the performance with optimizations like blocking [40] or sectorization [30].

Bonomi et al. [8] improved the space efficiency of counting Bloom filters [22] that support deletions using fingerprints and d-left hashing [47]. However, their fingerprint filter was still inferior to the vanilla Bloom filter, using twice the space. The Quotient filter [5] and the Cuckoo filter [21] recently revived this idea and increased the lookup throughput compared to the original Bloom filter using optimized hash table structures. The counting Quotient filter [37] and Morton filter [11] improve the space efficiency of hash table-based filters using a rank-based compression scheme. For our analysis, we consider the Cuckoo and Morton filters, since Quotient filters were repeatedly reported as suboptimal [21, 37, 48]. We also analyze the Xor filter [24], which promises high lookup performance and low false-positive rates.

Over the years, there has been a lot of research and discussion on which filter is the best. The key characteristics of interest are FPR, space consumption, build, and lookup performance. Every filter mentioned above shows improvements over the alternatives in at least one dimension. However, these comparisons primarily focus on the benefits of the newly introduced structures and optimizations. Breslow et al. [11], for example, demonstrated that batching improves the performance of their Morton filter but did not apply it to the Cuckoo filter. Lang et al. [28] provide the most extensive comparison of the Bloom and Cuckoo filters. Their performance-optimal analysis, however, primarily focuses on FPR and lookup throughput. Hence, it finds the optimal filter for performance-driven applications but fails to account for the memory footprint or excessive I/O bandwidth usage. In contrast, our analysis considers all the relevant characteristics for a wider range of filters.

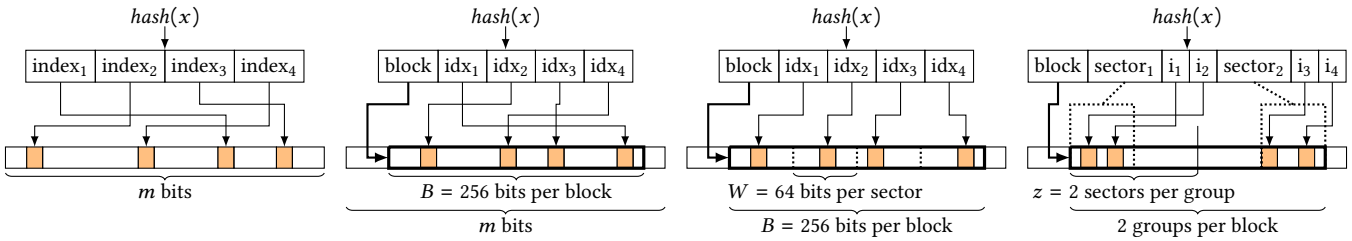
3 APPROXIMATE FILTERS

All filters have two configuration parameters in common: the number of keys to insert n and the fingerprint size k. For Bloom filters, k determines the number of bits to set/test for one key (often referred to as the number of hash functions). The minimum number of bits m allocated to the filters is $k \cdot n$. However, while insertions into Bloom filters always succeed, constructing a fingerprint filter can fail if not enough memory is available. Therefore, we allocate s-times more memory than needed, i.e., $m = k \cdot n \cdot s$, and use $k \cdot s$ bits per key (m/n). Table 1 summarizes all parameters used to configure the filters. A noteworthy difference between the two filter classes is that increasing the data structure's size improves the false-positive rate only for Bloom filters. For fingerprint filters, the size mainly decides whether the structure can be built at all and has no significant impact on the false-positive rate.

In the following section, we provide a brief overview of the four filters and their variants. We describe their implementation and the changes we make to support arbitrary fingerprint sizes. Furthermore, we analyze the FPR in the two filter classes and empirically determine the optimal values for the configuration parameters.

Table 1: Common configuration parameters

Symbol	Description
n	Number of elements to insert into filter.
k	Number of bits to check / set (Bloom filter); Fingerprint size in bits (Cuckoo, Morton, Xor filter)
s	Memory scale factor: $m = k \cdot n \cdot s$.
B	Block size in bits (Bloom filter).
W	Sector/word size in bits (Bloom filter).
z	Number of groups per block (Bloom filter).
b	Fingerprints per bucket (Cuckoo, Morton filter).



(a) Naïve **(b) Blocked** ($B = 256$) **(c) Sectorized** ($B = 256$, $W = 64$) **(d) Cache-sectorized** ($B = 256$, $W = 64$, $z = 2$)

Figure 3: Access patterns for different Bloom filter variants.

3.1 Bloom Filter

We begin by analyzing the naïve Bloom filter [6], which consists of an array of m bits. Each key inserted into the filter sets k bits in the array. We calculate the indices based on the key's hash value. The membership query tests if all k bits are set. Therefore, the lower bound for the false-positive rate [9] is

$$\varepsilon_{\text{bloom}} = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx \left(1 - e^{-kn/m}\right)^k. \quad (1)$$

For a fixed number of bits per key, the false-positive rate $\varepsilon_{\text{bloom}}$ is minimized by $k = \ln 2 \cdot (m/n)$ and, thus, $\varepsilon_{\text{bloom}} \approx 2^{-k}$ [34]. As a result, the optimal s for the naïve Bloom filter is $1/\ln 2 \approx 1.44$.

Figure 3a illustrates the insert and lookup operations' access patterns for $k = 4$. The performance correlates with two characteristics of the filter: the number of hash bits required to compute the array indices and the number of cache lines accessed. The number of hash bits determines how often the hash function needs to be evaluated and thus the computational effort. For large filters, accessing different cache lines results in cache and TLB misses. The number of misses can be reduced by accessing a single cache line for each key. We explore three Bloom filter variants that reduce random memory accesses: blocked, sectorized, and cache-sectorized.

3.1.1 Blocking. Blocked Bloom filters divide the bit array into blocks of B bits [40]. Insert and lookup operations first select one block, based on the hash value, and set/test only bits within that block. The size B is a power of two to facilitate bit addressing. Figure 3b shows the access pattern for 256-bit blocks.

If the block fits into one cache line (512 bits on x86 processors), each operation causes at most one cache miss. The naïve Bloom filter uses $k \cdot \log_2(m)$ bits to compute the array indices. The blocked Bloom filter uses $\log_2(m/B)$ bits for addressing the block and $k \cdot \log_2(B)$ bits to access the bits in the block. The hash function is evaluated less often, at the expense of higher FPRs, compared to the naïve filter. Register-blocked Bloom filters take it a step further and only use block sizes of 32 or 64 bits, ensuring that all subsequent operations access only the register.

3.1.2 Sectorization. Apache Impala [27] combines blocked Bloom filters with an m/k-partitioning [26] scheme at block level. Each block is further split into register-sized sectors of W bits. In every sector, we set or test $k/(B/W)$ bits, as shown in Figure 3c. Since each sector is accessed independently, we can parallelize the computation using SIMD instructions [30]. Sectorized Bloom filters restrict k to multiples of the number of sectors B/W.

3.1.3 Cache-Sectorization. Cache-sectorized Bloom filters [28] allow more flexibility for choosing k while also minimizing memory accesses. They assign the sectors to z equally-sized groups and set or test k/z bits in each group. However, the bits are not evenly distributed among the sectors within a group. Instead, we choose one sector from each group and access only the selected sectors. Figure 3d illustrates the access pattern for 256-bit blocks, 64-bit sectors, and $z = 2$ groups. Although k still has to be a multiple of z, more values are possible than in a sectorized Bloom filter.

3.1.4 Comparison. All the variants discussed so far focus on improving the performance. In Figure 4, we compare their FPRs: the naïve Bloom filter, two blocked versions (cache: 512-bit, register: 64-bit), the sectorized variant using four 64-bit words, and a cache-sectorized Bloom filter that splits one cache line into two groups of 64-bit words. The naïve Bloom filter's FPR matches the predicted 2^{-k}. For the other variants, the FPR initially deteriorates only slightly before the decay grows exponentially: for 25 bits per key, the FPR of the register-blocked Bloom filter is two orders of magnitude larger compared to the naïve Bloom filter. The cache-blocked and (cache-)sectorized variants are roughly one order of magnitude inferior. This corroborates the trade-off between FPR and lookup time.

Another disadvantage that comes with these optimizations is the convenience of choosing an optimal k that minimizes the filter's FPR. While the naïve Bloom filter accesses approximately $m/n \cdot 1.44$ bits per operation, it is more involved to determine k for the other variants. At first, the value grows linearly, but as the filter size increases, the growth declines, which is consistent with the FPR's behavior. Thus, optimized variants are beneficial for a small number of bits per key or when performance is more critical than FPR.

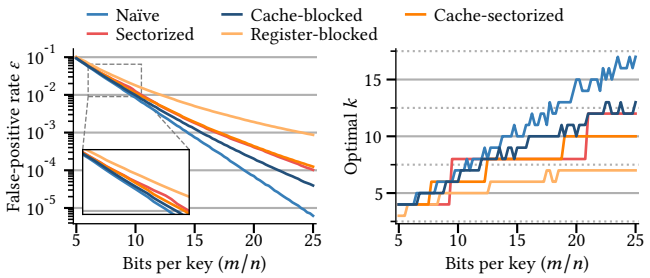
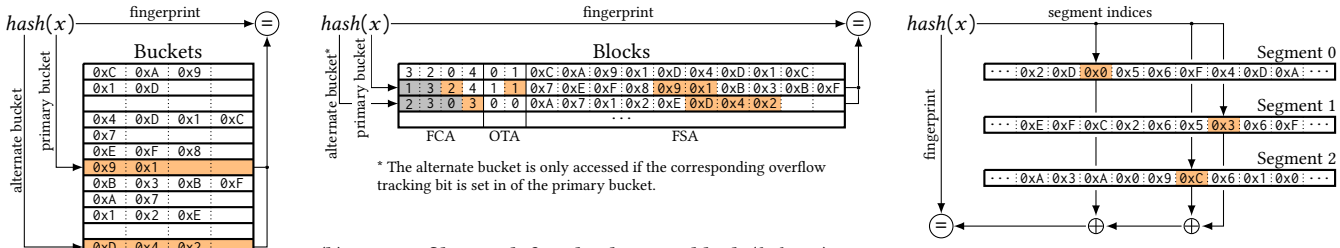


Figure 4: Comparison of different Bloom filter variants. Measurements are repeated a 100 times with random data.



(a) Cuckoo filter with four fingerprints per bucket ($b = 4$).

(b) Morton filter with four buckets per block ($bpb = 4$), ten fingerprints per block ($fpb = 10$) and a 2-bit overflow tracking array ($o = 2$). The filter compresses the buckets shown in Figure 5a.

(c) Xor filter with three segments ($l = 3$).

Figure 5: Lookup operations for the three fingerprint filters. We highlight the accessed fields in orange.

3.2 Fingerprint Filter

For this filter family, we consider three representatives: the Cuckoo filter, the Morton filter, and the Xor filter. They all store small *fingerprints* instead of entire keys in a hash table-like data structure.

3.2.1 Cuckoo Filter. Fan et al. describe the Cuckoo filter as "a compact variant of a cuckoo hash table that stores only fingerprints" [21]. Cuckoo hash tables use the Cuckoo hashing scheme [36] to resolve collisions. For every key x, it computes two candidate buckets, i_1 and i_2, which can hold the key. A lookup operation checks only these two buckets. Unlike the Bloom filter, inserting a key in the Cuckoo filter can fail. If both buckets are full, one occupying item is removed and inserted in its alternate bucket. The process relocates items until an empty slot is found or a threshold is reached.

Cuckoo filters store fingerprints instead of keys in a hash table. Each bucket in the table holds up to b fingerprints to achieve high load factors. Figure 5a illustrates lookup operations for $b = 4$ and 4-bit fingerprints. We reserve the value `0x0` to denote an empty slot in the bucket. The key's fingerprint f and the bucket index i_1 are derived from the hash value:

$$\begin{aligned} f &= \mathit{fingerprint}(\mathit{hash}(x)) \\ i_1 &= \mathit{index}(\mathit{hash}(x)) \end{aligned} \tag{2}$$

The fingerprint function extracts k non-zero bits, and the *index* function computes the bucket index.

In the case of a relocation, it is impossible to retrieve the original key based on the fingerprint. We, therefore, use the key's fingerprint f and the bucket index to determine the alternate bucket:

$$i_2 = \mathit{alternate\_index}(i_1, \mathit{index}(\mathit{hash}(f))) \tag{3}$$

In particular, the function must be self-inversive:

$$i_1 = \mathit{alternate\_index}(i_2, \mathit{index}(\mathit{hash}(f))) \tag{4}$$

Most open-source implementations of the Cuckoo filter support only a few different configuration options: the fingerprint size k can be 8, 12, or 16, while the number of fingerprints per bucket b is restricted to 2, 4, or 8. Our implementation eliminates these restrictions. We support fingerprints of up to 32 bits and between 2 and 8 fingerprints per bucket. However, to perform efficient lookups, a bucket still has to fit into a processor register, i.e., $k \cdot b \leq 64$. Furthermore, we do not pad buckets that are not byte-aligned, since wasting even one bit significantly reduces memory efficiency. Thus, some configurations, for instance, $k = 15$ and $b = 4$, perform either one unaligned or two consecutively aligned memory loads to access a bucket.

Lang et al. [28] give the following approximation for the false positive rate:

$$\varepsilon_{\text{cuckoo}} = 1 - \left(1 - 2^{-k}\right)^{2b\alpha}, \text{ with } \alpha = \frac{k \cdot n}{m} = \frac{1}{s} \tag{5}$$

where α denotes the load factor of the hash table. The FPR primarily depends on k and b, i.e., the rate improves as b decreases and the fingerprint size k increases. Increasing the hash table or bucket size decreases the probability of a failed build.

3.2.2 Morton Filter. The Morton filter [11] combines Cuckoo filters with Horton hash tables [12] to improve the load factor. It applies a rank-based compression scheme to store multiple buckets in a cache line-sized block. Figure 5b illustrates this principle: the twelve buckets from the Cuckoo filter in Figure 5a are compressed into three blocks. A block in the Morton filters consists of the fingerprint counter array (FCA) and the fingerprint storage array (FSA). The FCA has four entries that count the number of fingerprints in the corresponding bucket. To access the fingerprints of a bucket, we first compute the offset into the FSA by summing up the FCA entries. Then we load the fingerprints from the storage array.

The overflow tracking array (OTA) occupies the remaining space and tracks relocations. It is smaller than the number of buckets per block and maps multiple buckets to the same entry. When remapping a fingerprint from one of the buckets, we set the corresponding overflow tracking bit. If the primary bucket's OTA bit is zero, a lookup operation does not have to access the alternate bucket. The array reduces both the number of accessed blocks and the FPR. In contrast to the Cuckoo filter, the Morton filter ideally tests only one bucket with b fingerprints instead of two buckets.

Besides the number of fingerprints per bucket b, the Morton filter uses three additional parameters to control the arrays' size: the number of buckets per block bpb (i.e., the length of the FCA), the number of fingerprints per block fpb (i.e., the length of the FSA), and the number of bits in the OTA o. Optimally, a block is as large as a cache line to fully utilize the memory bandwidth:

$$bpb \cdot \left\lceil \log_2(b+1) \right\rceil + fpb \cdot k + o \leq 512 \tag{6}$$

3.2.3 Xor Filter. The Xor filter [24] implements a Bloomier filter [14], which maps a set of keys to their k-bit fingerprints. However, this mapping is only correct if the key is in the filter; otherwise, the result will be undefined. Filter queries use this characteristic to determine whether a key x is in the underlying set. The query

compares the computed fingerprint to the one stored in the filter. If x is in the filter, the retrieved and the computed fingerprint are equal. If x is not in the filter, the probability that both fingerprints are identical, i.e., the FPR, is 2^{-k}.

Following the design by Dietzelfelbinger and Pagh [18], we split the filter into three segments, S_0, S_1, and S_2, of size $m/3$ bits. The fingerprint of a key x is retrieved by `xor`-ing the three values from the corresponding segments (cf. Figure 5c):

$$f = \oplus_{i\in\{0,1,2\}} S_i[index(hash_i(x))] \tag{7}$$

A lookup compares the retrieved and the computed fingerprints:

$$fingerprint(hash(x)) = f \tag{8}$$

The difficulty is constructing the filter such that Equation 7 returns the correct fingerprints for all inserted keys. We consider two construction algorithms: the algorithm by Botelho et al. [10] used in the original Xor filter and a new algorithm by Dietzfelbinger and Walzer based on fuse graphs [19]. The original Xor filter takes a set of keys and determines the order in which to insert them and the segment to use. Then, it sequentially adds the keys to the corresponding segment j. For each key, we compute the value to insert based on the values stored in the other two segments and the fingerprint:

$$S_j[index(hash_j(x))] = fingerprint(x) \oplus \left(\oplus_{i\in\{0,1,2\}\setminus j} S_i[index(hash_i(x))]\right) \tag{9}$$

This construction algorithm ensures that once an item in one of the segments is written or read, it never changes. Thus, the fingerprint retrieved for x (Equation 7) is always identical to the key's fingerprint.

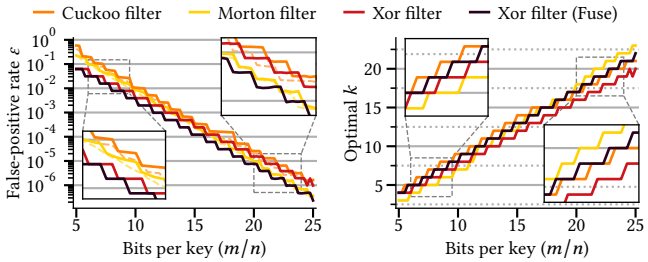
Our fuse graph-based Xor filter uses a construction algorithm similar to the original Xor filter. However, instead of three segments, the filter now consists of l segments. The insert and lookup operations are essentially the same as before, except that we first determine the index $i \in [0, l-3]$ and then access the segments S_i, S_{i+1}, S_{i+2}. The original Xor filter, thus, uses three segments of which only the first one is addressable. We use $l = 130$ segments to facilitate segment addressing.

As for the Cuckoo and Morton filters, the Xor filter's construction can fail if the segments are too small. Majewski et al. [33] found that for $l = 3$, the segments' minimum combined capacity is around 1.23-times the number of elements to insert. Our fuse graph-based Xor filter improves memory efficiency and ideally uses 1.13-times more space than required. If the build nevertheless fails, another hash function can be used. Graf and Lemire [24] accomplish this by combining the keys with a random seed before hashing. Our implementation of the filter supports fingerprint sizes between 1–32 bits. If the fingerprint does not match the registers' size (8-, 16- or 32-bit), the values are loaded unaligned and shifted accordingly. Compared to other filters, Xor imposes two additional restrictions: the keys must be unique, and the filter is immutable after building.

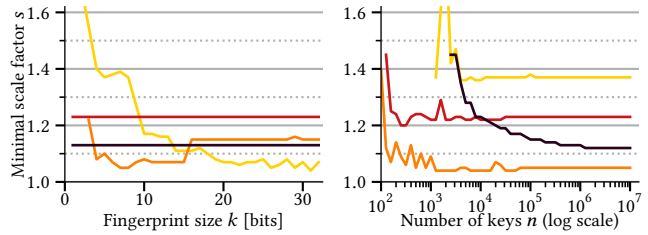
3.2.4 Comparison. As for the Bloom filters, we determine the optimal FPR and the fingerprint size k empirically. We also identify the minimum values of s that avoid unsuccessful builds. We present our results in Figure 6. The hash table-based filters allow a variety of different configurations. For the Cuckoo filter, we can adjust both the fingerprint and the bucket size b. The Morton filter supports scaling the ratio between the logical fingerprints $b \cdot bpb$ and the physical fingerprints fpb per block. We also compare the Xor filter's implementation based on fuse graphs against the original one with three segments.

Figure 6a shows the FPR for the three fingerprint filters. The solid lines represent the rates achieved by our configurations. The lighter dashed lines denote the optimal possible ε by dynamically changing the Cuckoo and Morton filter configuration for every measurement. Both hash table-based filters perform best with three fingerprints per bucket, except at the beginning, where more fingerprints are required to ensure a successful build. Our configuration dynamically chooses the number of fingerprints per bucket based on the fingerprint size. For the Morton filter, we use different configurations based on the fingerprint size k. When possible, we select the number of buckets per block and the OTA size as a power of two, to optimize the addressing within a block. Our experiments show that the ratio of logical and physical fingerprints per bucket is usually between 30% and 60%. When considering the FPR, the fuse graph-based Xor filter outperforms the other filters for the same number of bits per key.

Figure 6b shows the space overhead required to build the filter for the given fingerprint size. For the Morton filter, s decreases with increasing fingerprint size and is smaller than 1.1 when k is large. The Cuckoo filter switches the number of fingerprints per block from 3 to 2 for $k = 16$ and needs roughly 1.15-times more space for a successful build. The Xor filter always uses 1.23-times / 1.13-times more space regardless of the fingerprint size. We also analyze the stability of the factor s in Figure 6c. The parameter converges for $k = 8$ to the values from Figure 6b and remains steady for more than 10^4 elements. Both the Morton filter and the fuse graph-based Xor filter are susceptible to fluctuations with small filters.



(a) Optimal false-positive rate and fingerprint size



(b) Minimal filter size for increasing fingerprint size ($n = 10^6$).

(c) Minimal filter size for increasing number of elements ($k = 8$).

Figure 6: Comparison of different fingerprint filters for configurations that do not fail with 100 runs.

4 IMPLEMENTATION AND OPTIMIZATIONS

Aiming for a fair comparison of the filters presented earlier, we reimplemented them in C++ and exploited compile-time optimization where possible. Parameters like the fingerprint size k, the number of fingerprints per bucket b in Cuckoo filters, or the number of sections and groups in Bloom filters are compile-time constants. The filter's capacity m, determined by s and n, is a runtime parameter.

Prior work optimized their filter implementation using vectorization [11, 28] or faster hash functions [21, 28]. We apply these optimizations and combine them with partitioning — our new optimization improves both build and lookup performance for large filters without affecting either the false-positive rate or space usage.

We evaluate the impact of each optimization for a representative of each filter, namely:

- **Bloom:** 512-bit blocked Bloom filter ($s = 1.5$, cf. Section 3.1.1).
- **Cuckoo:** the Cuckoo filter with 4 fingerprints per bucket ($s = 1.06$, cf. Section 3.2.1).
- **Morton:** the Morton filter with 3 fingerprints per bucket and an 8-bit OTA ($s = 1.38$, cf. Section 3.2.2).
- **Xor:** the original Xor filter ($s = 1.23$, cf. Section 3.2.3).

All filters use $k = 8$ and the optimal values for s from Sections 3.1 and 3.2. We run the microbenchmarks on an Intel i9-9900X CPU (Skylake-X, 3.5-4.4 GHz) with 10 cores and 64 GB of memory, running Ubuntu 20.10 (Kernel 5.8, gcc 10.2), and repeat all measurements five times. Furthermore, we evaluate the scalability on a Xeon Gold 6212U (Kaby Lake, 2.4-3.9 GHz) with 24 cores and an AMD Ryzen 3950X (Zen2, 3.5-4.7 GHz) with 16 cores.

4.1 Block/Bucket Addressing

Fingerprint filters use the *index* function to map hash values to buckets. We apply the same function in the Bloom filters to compute the block indices.[1] The remainder of the division provides a precise and stable mapping: $index(i) = i \bmod C$ (with C denoting the number of blocks/buckets in the filter).

However, the modulo operation is more expensive than other integer arithmetic instructions. Thus, hash tables usually allocate space for power-of-two many elements and replace the modulo operation with a single bitwise and instruction: $index(i) = i$ and $(C-1)$. In the worst case, this approach allocates twice as many bits as needed and is not suitable for space-efficient implementations. Consequently, Lang et al. [28] proposed using so-called magic numbers to replace divisions by a sequence of multiple-shift instructions [49]:

$$index(i) = i - (mulh32(i, magic) \gg shiftAmount) \cdot C \quad (10)$$

The *mulh32* function returns the upper 32 bits of the 64-bit product $i \cdot magic$. The magic number and the shift amount are computed once when instantiating the filter using the libdivide library [1].

The Xor and Morton filter use a slightly different approach to substitute the modulo operation. They map 32-bit hash values to the interval $[0, C-1]$ using the method by Ross [43][2]:

$$index(i) = i \cdot C / 2^{32}, \text{ with } i \in [0, 2^{32} - 1] \quad (11)$$

Compared to the magic number approach, Ross's index function executes fewer instructions and is, therefore, slightly faster.

When using a different index function, we also have to adapt the *alternate_index* function (cf. Section 3.2.1) for Cuckoo and Morton filters. If the number of buckets C is a power of two, we can use the bitwise xor to compute the alternate bucket:

$$alternate\_index(i_1, i_f) = i_1 \oplus i_f \quad (12)$$

However, in combination with Ross's or the magic number addressing scheme, we have to resort to integer arithmetic and take underflows into account:

$$alternate\_index(i_1, i_f) = \begin{cases} i_f - i_1 & i_f \geq i_1 \\ i_f - i_1 + C & i_f < i_1 \end{cases} \quad (13)$$

A similar approach was proposed in [28] and later refined by Neumann and Kipf [35].[3]

While Ross's index function is slower than the power-of-two addressing scheme, the overall performance can improve as smaller filters offer better spatial locality. Figure 7 shows the speedup achieved by Ross's addressing approach, which uses the exact number of blocks/buckets C, relative to the power-of-two addressing scheme, which chooses the next greater power of two for C. We, therefore, compare a space-optimized filter against a baseline that can be up to two times larger. The speedup diminishes when the number of blocks/buckets C approaches a power of two. In these cases, the more expensive computation of Ross's index function is not worthwhile, since the baseline uses nearly the same amount of space as the space-optimized filter.

For construction, only the Bloom and Xor filters benefit from the reduced space usage. Performance doubles around the cache boundaries, as the baseline already exceeds the cache size, resulting in significantly more cache misses. The space-optimized Cuckoo and Morton filter use the slower *alternate_index* function from Equation 13, while the baseline uses Equation 12. Therefore, hash table-based filters cannot benefit from this optimization and are most of the time slower than filters with power-of-two many buckets. Nevertheless, in the following sections, we use the more space-efficient addressing scheme by Ross, which is between 1.1-times and 1.5-times faster than the magic number approach.

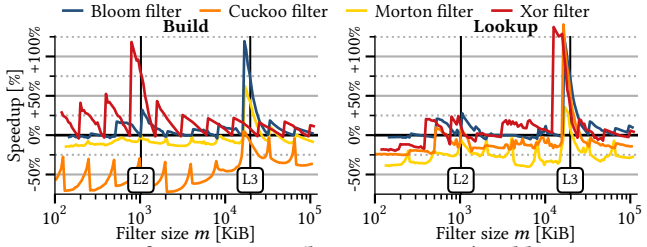


Figure 7: Performance gain/loss using Ross's addressing approach; the baseline is the power-of-two addressing scheme that allocates more blocks/buckets than needed.

4.2 Hashing

Hash functions are an integral part of all filters used to compute block/bucket addresses, bit indices, and fingerprints from 32/64-bit keys[4]. The quality of the function, i.e., producing uniformly

[1] Although the naïve Bloom filter accesses only bits and no blocks, internally, we still need to address a register-sized word, i.e., a block, to load into the processor's registers.

[2] Lemire promoted this idea in a blog post from 2016 [29].

[3] We can omit $(C-1)$ from the original formula as it is not necessary for correctness.

[4] We map larger keys like strings to 32/64-bit values before passing them to the filter.

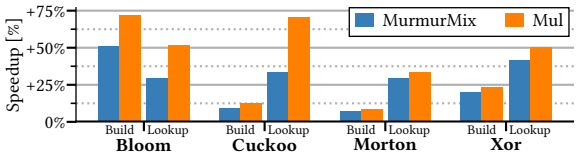


Figure 8: Speedup using the MurmurMix or Mul hash functions; the baseline is the CityHash function (n = 10 M).

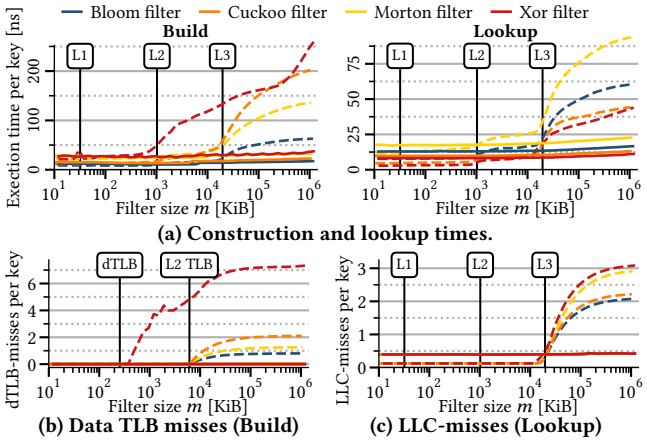
distributed hash values, is crucial to achieving the expected false-positive rates. However, the hash function's execution time is also vital for performance. Therefore, most filter implementations employ fast Multiply-Shift hash functions [42].

To investigate the impact of hash functions on the filter's performance, we compare two hash functions used in existing filter implementations with Google's CityHash [38]: 1) the MurmurHash3 finalizer (MurmurMix) [3], due to its good behavior in practice, and 2) multiplicative hashing (Mul) that reduces computational effort by multiplying the keys with large prime numbers. Figure 8 shows the performance gain for both hash functions relative to the CityHash function. For the Bloom filter, both MurmurMix and Mul improve performance significantly. The speedup increases even further for larger values of k, as the hash function is executed more often.

When building the filter on skewed data, multiplicative hashing can increase the false-positive rate, and the fingerprint filters' construction is more likely to fail. Therefore, we use the MurmurHash3 finalizer to ensure that our analysis is accurate for arbitrary data distributions. Nevertheless, for uniformly distributed data, the multiplicative variant is a good option. RocksDB [20], e.g., first hashes arbitrary large keys with XXHash3 [16] and then uses multiplication to generate more bit indices for its blocked Bloom filter.

4.3 Partitioning

The performance of all filters deteriorates as the number of elements increases. For filters that do not fit into the last-level cache (LLC), lookups are up to one order of magnitude slower than in filters that fit into the L1 cache, due to random memory accesses that miss the cache. Prior implementations reduced the number of accessed cache lines by blocking, but even that requires at least one random memory access per operation. We propose, instead, to partition the filters.



(a) Construction and lookup times.

(b) Data TLB misses (Build)

(c) LLC-misses (Lookup)

Figure 9: Build and lookup performance for partitioned (solid lines) and non-partitioned (dashed lines) filters.

Inspired by radix joins [46], we use radix partitioning to divide the set of keys before building the filter or performing lookups. For construction, one filter is built for each partition. Lookups first determine which filter to use before testing the keys in the respective partition. If the filter fits in the caches, fewer TLB and cache misses occur. We optimize our single-pass radix partitioning implementation using software write-combine buffers and non-temporal streaming stores [50].

Figure 9a shows the time per key needed to build the filter and perform a lookup. The partitioned filters (solid lines) include the partitioning time. The partitioned variants outperform the baseline (dashed lines) by around 1 MiB. The Xor filter benefits earlier from partitioning due to its memory-consuming construction algorithm. The Cuckoo and Xor filters, in particular, benefit from this optimization. Their build times are nearly 10x faster for filters that exceed the LLC. They perform more random accesses than the Bloom and Morton filters and benefit more from the increased spatial locality.

For all four filters, partitioning reduces the overall number of TLB misses by three orders of magnitude, and the number of last-level cache misses by almost one order. Figure 9b shows that almost no data TLB misses occur for partitioned filters. At first, partitioning increases the number of LLC misses (cf. Figure 9c), but in exchange, the number remains constant even for filters exceeding the L3 cache. Unoptimized filters incur several misses per key as soon as the filter exceeds the LLC.[5]

While applying radix partitioning in Bloom filters is straightforward, we experienced difficulties with fingerprint filters. More specifically, constructing the Cuckoo or the Morton filters tends to fail for more than 512 partitions. We, therefore, also use the Xor filter's seed-based retry technique for hash table-based filters with partitioning. If building the filter fails, we xor the keys with a seed and try again.

In Figure 10, we show the optimal number of partition bits to use. The Bloom filter shows a clear picture. As soon as the filter size exceeds the L2 cache, partitioning pays off for both building and probing the filter. As anticipated, the optimal number of partitions grows with increasing filter size. Partitioning even improves the performance of fingerprint filters with smaller filter sizes. The Xor filter benefits much sooner from a partitioned build process. For lookups, all fingerprint filters perform similarly, thus we only show the Cuckoo filter. Figure 10 also shows that the precise number of partitioning bits does not have a significant impact on the performance. All filters get within 5% of the optimal performance even when choosing a partition size that differs by an order of magnitude.

[5] The hardware prefetcher causes additional cache misses for the blocked Bloom filter and the Morton filter.

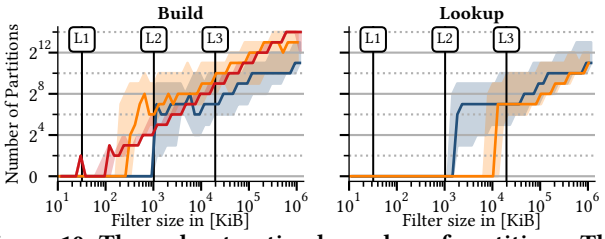


Figure 10: Throughput-optimal number of partitions. The colored area shows throughput deviations of less than 5%.

However, it is important to note that partitioning the filters introduces an additional requirement, namely batching the keys before performing insert or lookup operations. Without partitioning, all filters except the Xor filter support inserting single keys. Although lookups for single elements are still possible, the performance drops by 10%. The reason for this is the additional work needed to determine the correct filter for the key. Nevertheless, partitioning is the most effective technique to guarantee stable performance for filters that exceed the caches, if we can batch the operations. One particular advantage of partitioning is that it does not affect the FPR, unlike other optimizations that minimize the number of cache misses. RocksDB uses partitioning to split its Bloom filters and store them on disk rather than in memory [20]. An additional top-level index loads the correct partition from disk when it is needed.

4.4 Vectorization

As the number of cache lines accessed per lookup cannot be reduced further for blocked Bloom filters, several authors optimize the computations using SIMD instructions [27, 28, 39]. We found two techniques for vectorizing approximate filter structures: parallelizing the computations for one key (vertical vectorization) or performing multiple lookups in parallel by assigning one key to each SIMD lane (horizontal vectorization). While horizontal vectorization can be used with all filters, vertical vectorization only works with the Impala library's sectorized Bloom filter [27]. We implement horizontal vectorized lookups for all four filters and their variants. In contrast to existing vectorized implementations for fingerprint filters, we support arbitrary fingerprint sizes.

Our implementations target processors that support the AVX512F and AVX512VL instruction sets.[6] We use `gather` and `scatter` instructions to implement horizontal vectorization and use masking to avoid branches. Our vectorized filters share most of the code with the scalar implementation. The SIMD instructions are inserted through compiler intrinsics during compilation. In a few cases, such as unaligned memory accesses, the implementations differ: the `gather` instruction only supports aligned accesses, thus, we have to perform two aligned loads instead of one unaligned load.

Although the number of executed instructions decreases almost eightfold, the vectorized filters are at most twice as fast (cf. Figure 11). The vectorized filters spend most of the time fetching data from memory as `gather` scales only modestly compared to scalar loads. As soon as the filters exceed the L2 cache, the performance of vectorized filters deteriorates due to TLB and cache misses. Partitioning mitigates this effect, but the speedup decreases, as both versions use the same radix partitioning implementation. We also vectorized the construction of the Bloom, Cuckoo, and Xor filter using `scatter` instructions. However, only the sectorized Bloom filter using vertical vectorization benefits from this optimization.

[6] We emulate missing instructions on older platforms, but we expect no performance gain in these cases.

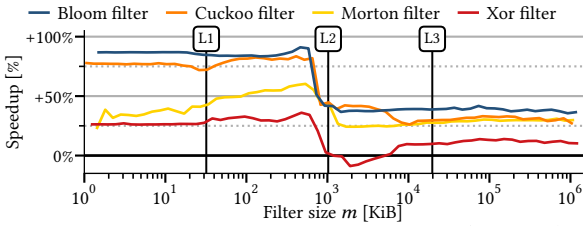


Figure 11: Speedup using vectorized filters (AVX512); the baseline are scalar filters (partitioning is enabled).

4.5 Multi-Threading

An additional benefit of partitioning is that it simplifies the implementation of task-level parallelism: each thread builds the filters for different partitions and avoids synchronization during construction. We parallelize the radix partitioning as proposed by Balkesen et al. [4] and use the single-threaded algorithms to build each filter. This approach is particularly suitable for fingerprint filters, since synchronizing their construction algorithms is non-trivial. Although it is easier to parallelize the Bloom filters using atomic instructions, partitioning results in less overhead.

Lookups, in contrast to insertions, require no synchronization. Once the filter is built, multiple threads can read it simultaneously. In combination with partitioning, two different parallelization schemes are possible: partition the data before assigning one partition to each thread ($\text{Lookup}^{\text{Part+MT}}$) or first split the keys into jobs and then partition them separately ($\text{Lookup}^{\text{MT+Part}}$). The second option has the advantage that no synchronization between the threads is required. However, the spatial locality decreases, since each job accesses the entire filter. The first scheme, in contrast, reads only the part of the filter relevant for the current partition.

Figure 12 shows the speedup when building the filters and performing lookups with multiple threads. For construction and lookups with partitioning, we report the numbers relative to the partitioned filter versions. For non-partitioned lookups ($\text{Lookup}^{\text{MT}}$), we use the filter without partitioning as the baseline. The non-partitioned filters scale almost linearly on all three machines. The partitioned filters, on the other hand, scale sub-linearly due to the radix partitioning. The second partitioned lookup variant ($\text{Lookup}^{\text{MT+Part}}$) scales on all three machines better than the first variant ($\text{Lookup}^{\text{Part+MT}}$).

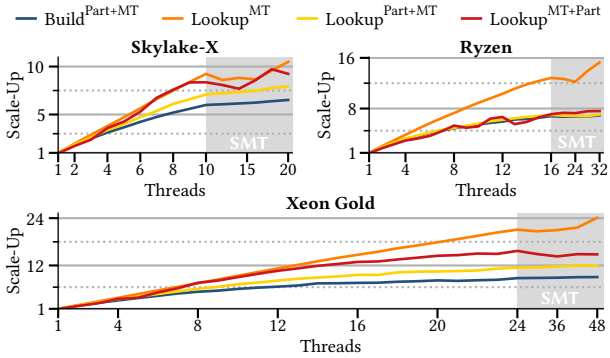


Figure 12: Scalability of the blocked Bloom filter on different machines relative to a partitioned version; $\text{Lookup}^{\text{MT}}$ uses the non-partitioned version as baseline. (n = 100 M)

Table 2: Throughput on Skylake-X [Keys/s (scale-up)].

	Bloom	Cuckoo	Morton	Xor
$\text{Build}^{\text{Part+MT}}$	381 M (6.0x)	341 M (6.9x)	235 M (7.2x)	197 M (6.9x)
$\text{Lookup}^{\text{MT}}$	165 M (9.2x)	255 M (9.5x)	113 M (9.6x)	287 M (9.4x)
$\text{Lookup}^{\text{Part+MT}}$	480 M (7.1x)	574 M (6.6x)	368 M (7.6x)	657 M (6.3x)
$\text{Lookup}^{\text{MT+Part}}$	565 M (8.4x)	662 M (7.7x)	342 M (7.1x)	725 M (7.0x)

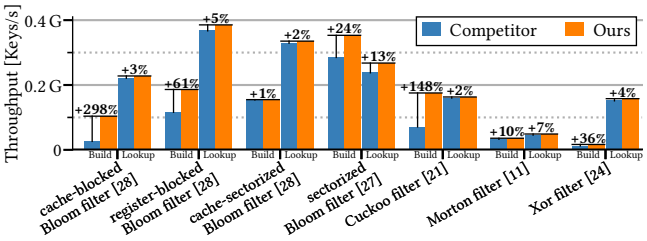


Figure 13: Comparison to related work. Our filters use the same configuration and disable partitioning ($k = 8$, $n = 1\,\text{M}$).

Table 2 lists the throughput for all four filters using the available hardware threads. Although partitioned filters achieve only sublinear speedup, they are still at least twice as fast as the non-partitioned versions and offer a decent scale-up for construction.

4.6 Comparison against State of the Art

Our implementations can enable each of the presented optimizations independently of each other. We use this feature to compare against the state-of-the-art. For a fair comparison, we do not use partitioning. Instead, we ensure that the filter size is less than the LLC and enable only the optimizations supported by related work:

Lang et al. [28]: Register-blocked, sectorized, and cache-sectorized Bloom filters using the magic number-based addressing approach and multiplicative hashing; supports vectorized lookups (AVX512) and operates, unlike other filters, on 32-bit values.

Kornacker et al. [27]: Vectorized implementation of the sectorized Bloom filter using AVX2 instructions and 256-bit blocks.

Fan et al. [21]: Initial Cuckoo filter implementation with power-of-two sized hash tables.

Breslow et al. [11]: Support batched lookups in their Morton filter but provide no handwritten vectorized implementation; instead, they rely on the compiler's auto-vectorization capabilities. Therefore, we disable batching and use the scalar versions.

Graf et al. [24]: Provide the first practical implementation of Xor filters. Their construction algorithm allocates 1.23-times more space and does not use the fuse graph optimization.

As shown in Figure 13, we achieve better performance for all filters and significantly improve construction time for some filters.

5 EVALUATION

We now present an experimental evaluation of our filter implementations, in which we vary all the parameters relevant to the filters. The goal is to identify which filter to use when optimizing for space consumption, throughput, or false-positive rate.

5.1 Experimental Setup

We ran all experiments on the Skylake-X machine and used the following filter configurations.

- ∘ **Bloom:** All experiments evaluate the naïve, register-blocked, cache-blocked, sectorized, and cache-sectorized variants of our Bloom filter implementation.
- ∨ **Cuckoo:** We use a performance-optimized configuration that chooses the fingerprints per bucket b depending on k.
- ∧ **Morton:** The number of buckets per block bpb and the OTA size o are powers of two whenever possible and correspond to the configurations used in Section 3.2.4.
- × **Xor:** We include both the original and the fuse graph-based method to build the Xor filter.

Parameters & Methodology — We first evaluate Lang et al.'s performance-optimal metric that finds the optimal filter which minimizes the per-tuple work: $t_l + \varepsilon \cdot t_w$ [28]. It combines the lookup time t_l with the false-positive rate ε (FPR) to find the best-performing filter for a certain workload[7]. The parameter t_w describes the workload's estimated extra work time for a false positive. We evaluated this metric for exponentially growing datasets between 10 K and a 100 M million keys on random data generated by the Mersenne Twister engine from the C++ STL. We also varied the memory scale factor s from 4–26 and the fingerprint size k from 1–25[8].

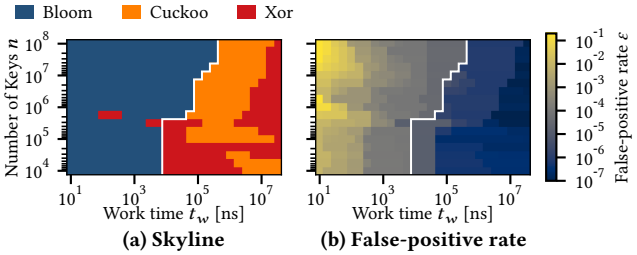
Next, we performed an in-depth analysis of the filters' lookup and build performance for 10 K, 1 M, and 100 M keys by clustering the data by filter size m and false-positive rate. This benchmark finds the filter with the highest throughput for a given FPR and filter size and examines the effects of tailoring one parameter. Besides scaling the parameters s and k, we used all valid combinations of vectorization and partitioning enabled or disabled and varied the number of partitions (2^i partitions, $5 \le i \le 12$). For every cluster, we report the filter with the maximum throughput.

In both experiments, we measure the performance on ten hardware threads and report the average of five repetitions. Unless stated otherwise, we show the best-performing results from all combinations of vectorization and partitioning enabled or disabled.

5.2 Lookup Performance

The skyline plot in Figure 14a shows the optimal filter according to the performance-optimal metric. The white line divides the measurements into two parts where either Bloom filter variants or fingerprint filters dominate. Our results show strong similarities to those of Lang et al. [28]: for small work times t_w, Bloom filters are optimal. As soon as t_w, the extra work time for a false-positive, is larger than 10^4 ns, low false-positive rates are more critical than the lookup time (cf. Figure 14b), and the fingerprint filters perform better. For roughly half a million keys, the Xor filter briefly outperforms the Bloom filters for low t_w (left of the white line). While all Bloom variants with comparable FPRs exceed the L2 cache, the Xor filter still fits into the cache, as it uses the space more efficiently.

[7] We assume that positive and negative lookups in filters take the same time on average.
[8] The sectorized and cache-sectorized Bloom filter variants restrict the parameter k and thus have respectively fewer data points.



(a) Skyline (b) False-positive rate

Figure 14: Performance-optimal filters for increasing number of keys n and t_w (vectorized, partitioning disabled).

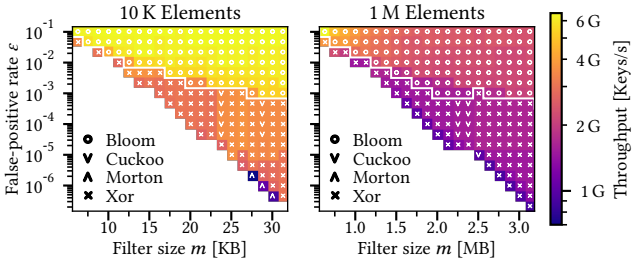


Figure 15: Lookup performance and best-performing filter. The white line separates Bloom from fingerprint variants.

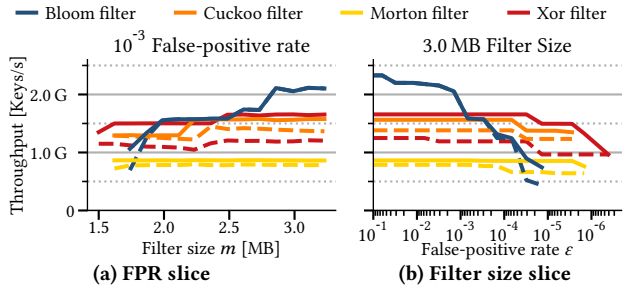
The Xor filter dominates most of the fingerprint filters' area for less than 10^5 elements. For more elements, the filters exceed the L1 cache and the Cuckoo filter takes over as it accesses at most two cache lines. With three random memory accesses, the Xor filter can outperform the Cuckoo filter only for small datasets or high work times t_w that require very low FPRs. In the remainder of this section, we analyze the filters' false-positive rate and throughput in more detail for 10 K, 1 M, and 100 M elements and evaluate the impact of partitioning. We choose the filter sizes to see how big the difference in performance is when the filter resides in different levels of the cache hierarchy.

5.2.1 Optimal filter variant. Figure 15 and Figure 17 show the filter with the highest throughput for the given memory budget and the measured false-positive rate. The white line separates Bloom and fingerprint filters with both partitioning and vectorization enabled. Since no filter can reach arbitrary low FPRs, there are no data points in the graphs' lower-left halves.

The filters with *10 K elements* are smaller than the L1 cache, so we obtain the highest throughput. For *1 M elements*, the filters only fit into the last-level cache, except for the smallest ones in the top left corner, which still operate in the L2 cache.

We investigate the filters in more detail in a horizontal and vertical slice from the lookup measurements in Figure 15 (1 M elements). In the FPR slice (Figure 16a), the Bloom filter dominates the performance except for small sizes, where the (partitioned) Xor filter performs best. Only the naïve Bloom filter can attain the false-positive rate using the given space but suffers substantial L1 cache misses even for small sizes due to high k. For a higher memory budget, first the sectorized and then the register-blocked variants take over. In conclusion, larger filters can improve the throughput to some degree using optimized variants that increase locality.

When looking at a constant filter size (e.g., 3.0 MB in Figure 16b), we see that the fingerprint and Bloom filters perform differently.



(a) FPR slice **(b) Filter size slice**

Figure 16: Slice from 1 M elements for constant size or FPR (dashed lines show the throughput without partitioning).

While Bloom filters can trade a worse FPR for performance, the fingerprint filters' throughput does not improve for high FPRs. Bloom filters offer more possibilities for performance optimization by clustering the memory loads or reducing the number of hash functions k and, thereby, the computational effort at the cost of significantly increasing the FPR. Fingerprint filters, in contrast, can only scale the fingerprint size, which slightly improves performance if a higher FPR is acceptable. Nonetheless, fingerprint filters achieve lower false-positive rates for the given memory budget than the Bloom filters at a modest throughput reduction. This is an attractive trade-off if false positives are expensive, like accessing data on disk or over the network.

Figure 16 also shows the maximum throughput without partitioning. For the Bloom filter, the dashed line is most of the time not visible, i.e., partitioning does not pay off. The Xor filter's throughput significantly decreases if partitioning is not available, and it falls behind the Cuckoo filter. This matches the results from Figure 14: non-partitioned Xor filters are only optimal once very low FPRs are required that the Cuckoo filter cannot attain.

Recap — Bloom filters offer the highest throughputs but cannot reach low FPRs on a size budget. Fingerprint filters, most notably Cuckoo and Xor, can reach very low FPRs while being a bit slower.

5.2.2 Optimal Bloom filter variant. If opting for performance, we next break down which Bloom filter variant performs best in Figure 18. Register-blocked filters dominate most of the areas where the Bloom filter performs better than fingerprint filters. Register-blocking trades FPR and size for maximum throughput by reducing memory accesses to a minimum. Consequently, they are the filter of choice for high throughput scenarios like semi-join reducers where false-positives are inexpensive.

The other variants do not achieve such high throughputs but offer a better trade-off between filter size and false-positive rate. Sectorized and cache-sectorized filters perform second best as they use the entire cache line to reduce the FPR and specialized access patterns to improve performance. The cache-blocked filter offers a slightly better FPR since it does not restrict the set bits' placement. Naïve bloom filters are not competitive in performance but offer by far the lowest possible FPRs at the cost of having a complete random access pattern.

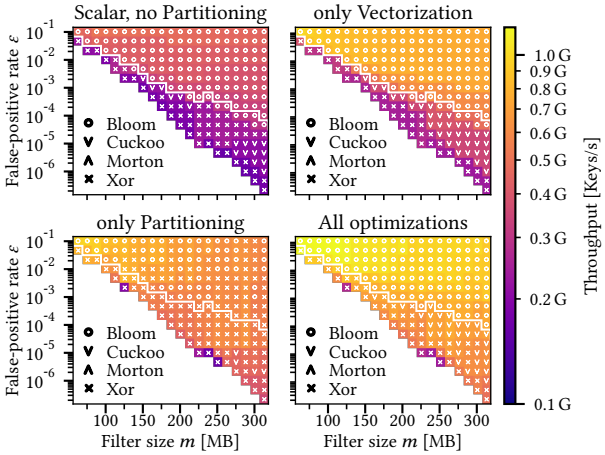


Figure 17: Lookup performance for 100 M elements.

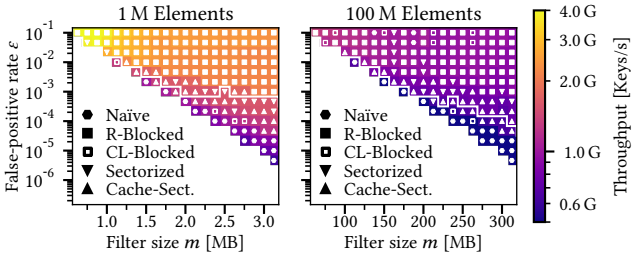


Figure 18: Best-performing Bloom filter variant for lookup.

Recap — Bloom filters gradually trade high throughput for better FPRs by decreasing the spatial locality. The range goes from the register-blocked filters that offer the highest throughput to the naïve Bloom filter with the lowest FPR.

5.2.3 Optimal k. Although all filters support arbitrary k for the fingerprint size or the number of hash functions, this flexibility mostly benefits the Bloom filter variants. It directly improves the performance as fewer memory loads occur and the hash function is evaluated less often. The register-blocked filter typically uses very low k (≤ 4) for maximum performance throughout most of our measurements. For the naïve Bloom filter, this number can increase to 20 when aiming for very low FPRs, resulting in approximately 20 cache misses (without partitioning). When using the Xor or Cuckoo filters, powers of two for k perform best and 16-bit fingerprints, in particular, offer a good trade-off. These sizes simplify fingerprint comparison in the vectorized implementations and allow for aligned loads, reducing LLC misses.

5.2.4 Vectorization. The vectorized lookup implementations improve the performance of all filters, as shown in Figure 19, which compares the best-performing vectorized and non-vectorized filters. We included the same dividing line as in Figure 17 for reference. As expected, we observe the biggest performance boost for small filter sizes that fit into the L2 cache (cf. Section 4.4).

In general, the register-blocked Bloom filter (cf. Figure 18) benefits the most from vectorized lookups since it performs only one memory load while the other variants require multiple loads. The Xor filter performs the most random memory accesses and thus, is at most 50% faster. The performance of the Cuckoo filter, which accesses at most two random memory words, almost doubles. Without vectorization, the Xor filters dominate large areas for 100M (cf. Figure 17). However, the difference in throughput is mostly less than 20%, and vectorized Bloom and Cuckoo filters take over.

Recap — Vectorization pays off without impacting the FPR. It boosts all filter implementations and should be applied in any setting that allows concurrent probing of multiple elements.

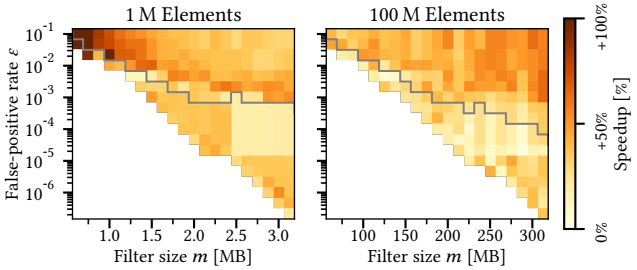


Figure 19: Vectorized vs. non-vectorized filters.

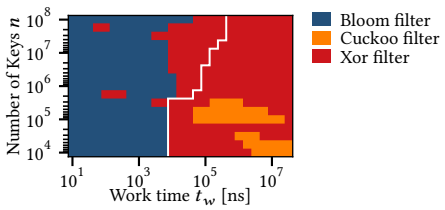


Figure 20: Partitioned performance-optimal filters.

5.2.5 Partitioning. Since we observed substantial performance improvements with partitioning in Section 4.3, we now compare the lookup performance of the fastest partitioned and non-partitioned filters, as shown in Figure 21. However, this time the baseline has vectorization and multi-threading enabled, which partly amortizes the achieved speed-up. The two right-hand plots in Figure 17 show the fastest filter implementation for 100 M keys. We included the same dividing line for reference.

The tipping point at which partitioning begins to provide benefits is at 1 M elements. The filter that dominates performance in most cases — the register-blocked Bloom filter (cf. Figure 18) — benefits the least from partitioning since it already minimizes memory accesses. Furthermore, we hide the cache miss latencies by building a mask for set bits in the (SIMD) registers while loading the block from memory. When fingerprint filters dominate the performance, they can get up to a third faster, and in a sense, partitioning narrows the gap between the two filter families.[9]

For 100 M elements, partitioning can almost triple the maximum performance of some fingerprint filters, while the peak Bloom performance remains about the same. Even though not visible in the figure, the naïve Bloom filter's throughput improves, as partitioning reduces the number of cache misses, closing the performance gap to register-blocked and sectorized variants. Overall, partitioning closes the performance gap between the fingerprint filters and the (register-blocked) Bloom filter. In particular, the Xor filter benefits from the optimization and even closes in on the Cuckoo filter (cf. Figure 17). This is also evident in the performance-optimal analysis with partitioning in Figure 20: the Xor filter takes over large parts formerly dominated by the Cuckoo and the Bloom filters (left of the white line).

Recap — The larger the number of elements gets, the more partitioning pays off. Most notably, the optimization boosts the Xor filter, narrowing the gap to the Bloom and Cuckoo filters.

[9] In some cases, small partitions hinder the Xor filter from being built, which is the reason for the missing data points in the lower right corner.

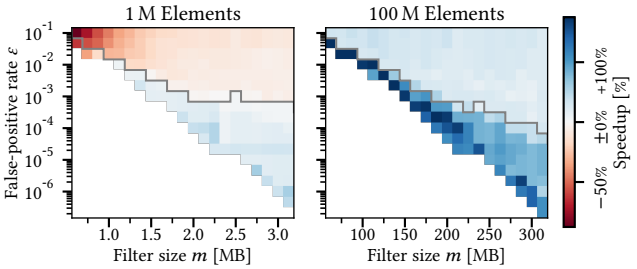


Figure 21: Partitioned vs. Non-Partitioned variant.

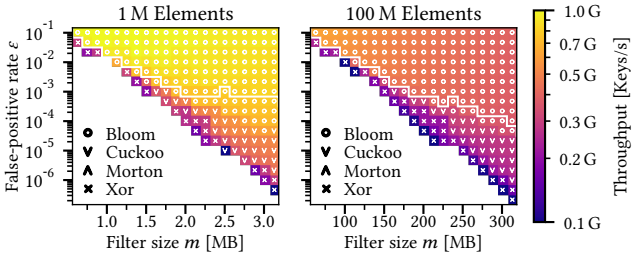


Figure 22: Best-performing filter for construction.

5.3 Build Performance

The only effective way to parallelize the filter's construction process is first to partition and then to build one filter per partition. Hence, we always use partitioning, which scales well, as shown in Table 2.

Figure 22 shows the filter with the best construction time. We include the white line from Figure 15 to compare how the divider changes from lookup to build performance. While, once again, register-blocking dominates most of the upper area, the sectorized Bloom filter benefits from vectorized inserts and is now the optimal choice for FPRs smaller than 1%. For small FPRs, even the naïve Bloom filter overtakes some of the fingerprint filters.

The Cuckoo filter builds slightly more slowly than the Bloom filters. For small load factors, relocations are rare and insert operations ideally access only the primary and alternate bucket. The Xor filter achieves higher load factors and lower FPRs, but its construction is more involved and thus takes longer. It only outperforms the Cuckoo filter for very low FPRs that hash table-based filters cannot attain or for high load factors.

Recap — If the focus lies on performance, the register-blocked Bloom filter builds the fastest. Other Bloom variants or the Cuckoo filter also offer lower FPRs for the same space consumption at the cost of slightly lower build and lookup throughputs. If the filter is not rebuilt regularly, the Xor filter is an option. It builds the slowest and does not support updates but offers even higher lookup performance and lower FPRs.

6 LESSONS LEARNED

Each of the filter variants and optimizations offers different performance characteristics that lead to different use-cases.

Optimizations For Bloom and Xor filters, specialized addressing schemes improve throughput and reduce space overhead. Hash table-based filters, however, should rather use the power-of-two sized tables if the available space is not limited. When solely optimizing for build or lookup performance, multiplicative hashing is the fastest option. However, for skewed data, the computed hash values are not uniformly distributed, and we recommend using MurmurHash3's finalizer.

Vectorization improves the lookup performance of all four filters irrespective of their size when batching the operations, most notably the (blocked) Bloom filter and the Cuckoo filter. Although it is also possible to vectorize the inserts, we only noticed a stable performance boost for the sectorized Bloom filter. Partitioned filters need to compensate for the radix partitioning and are thus of benefit when the filters exceed the LLC. Even though unpartitioned filters scale better in multi-threaded lookups, partitioning still provides a significant performance boost and decent scalability. Furthermore, it allows for trivial multi-threaded construction of all variants by building a filter per partition. The Xor filter profits the most from partitioning, which narrows the gap between Bloom and fingerprinting filters. Where batching is possible, both optimizations improve the throughput manifold (cf. Figure 1).

Filter Variants As expected, Bloom filters dominate both build and lookup performance when high FPRs are acceptable. Most variants, like register-blocking, optimize for better performance at the cost of higher FPRs. Other variants are not as fast but offer better FPRs for the same memory budget.

When focusing on space consumption and low false-positive rates, the Xor filter using our new construction algorithm achieves the lowest FPR for the given space and still provides decent lookup throughput. However, the filter has the longest construction time and is immutable. The Cuckoo filter cannot compete with the Xor filter's FPR but instead offers more functionality with similar lookup performance. This filter should be considered particularly with workloads that insert or delete keys after construction. Morton filters effectively improve the Cuckoo filters' space usage and false-positive rate at the cost of decreased lookup and build throughput.

Ultimately, tuning the performance of each filter is still a trade-off. By spending more time on the filter's construction, we can use the space more efficiently and reach lower FPRs for the same memory budget. Similarly, when granting slightly more time for lookups, fingerprint filters like the Xor filter pay off because their FPR is lower. When increasing the filter size, preference can be given to either lower false-positive rates or higher performance, using locality-optimized Bloom filter variants or register-friendly fingerprint sizes.

7 CONCLUSION

Our work focuses on optimizing the four most promising approximate filters (Bloom, Cuckoo, Morton, and Xor) and identifying the optimal filter for the four key dimensions: false-positive rate, memory footprint, build, and lookup throughput. To allow a fair comparison, we reimplemented the filters, applied all existing optimizations, and moreover, evaluated *radix partitioning*, which significantly boosts the performance for large filters without affecting the false-positive rate.

Each of the optimizations and filter variants has performance characteristics (outlined in Section 6) that are beneficial in different use-cases. Bloom filters are the most reasonable choice for in-memory query processing, as high throughput is crucial and high FPRs are acceptable. Fingerprint filters offer a better trade-off when focusing on space consumption and achieving low FPRs. Most notably, Xor filters are attractive for applications like LSM trees, which currently rely on sub-optimal Bloom filters [17]. However, if applications need more functionality with comparable FPR, the Cuckoo filter is the next best choice, as the Xor filter is immutable.

Acknowledgments

Partially funded by German Research Foundation (DFG) – 361477420. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 725286).

REFERENCES

[1] 2019. *Libdivide v3.0.* Retrieved December 31, 2020 from https://github.com/ridiculousfish/libdivide

[2] Paulo Sérgio Almeida, Carlos Baquero, Nuno Preguiça, and David Hutchison. 2007. Scalable Bloom Filters. *Inform. Process. Lett.* 101, 6 (2007), 255–261. https://doi.org/10.1016/j.ipl.2006.10.007

[3] Austin Appleby. 2011. *MurmurHash & SMhasher.* Retrieved December 31, 2020 from https://github.com/aappleby/smhasher

[4] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Özsu. 2013. Main-Memory Hash Joins on Multi-Core CPUs: Tuning to the Underlying Hardware. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013) (ICDE '13)*. IEEE Computer Society, USA, 362–373. https://doi.org/10.1109/ICDE.2013.6544839

[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. 2012. Don't Thrash: How to Cache Your Hash on Flash. *Proc. VLDB Endow.* 5, 11 (July 2012), 1627–1637. https://doi.org/10.14778/2350229.2350275

[6] Burton H. Bloom. 1970. Space/Time Trade-Offs in Hash Coding with Allowable Errors. *Commun. ACM* 13, 7 (July 1970), 422–426. https://doi.org/10.1145/362686.362692

[7] Peter A. Boncz, Stefan Manegold, and Martin L. Kersten. 1999. Database Architecture Optimized for the New Bottleneck: Memory Access. In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*. 54–65. http://www.vldb.org/conf/1999/P5.pdf

[8] 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* (Zurich, Switzerland) *(ESA'06)*. Springer-Verlag, Berlin, Heidelberg, 684–695. https://doi.org/10.1007/11841036_61

[9] Prosenjit Bose, Hua Guo, Evangelos Kranakis, Anil Maheshwari, Pat Morin, Jason Morrison, Michiel Smid, and Yihui Tang. 2008. On the False-Positive Rate of Bloom Filters. *Inf. Process. Lett.* 108, 4 (Oct. 2008), 210–213. https://doi.org/10.1016/j.ipl.2008.05.018

[10] 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* (Halifax, Canada) *(WADS'07)*. Springer-Verlag, Berlin, Heidelberg, 139–150.

[11] 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. https://doi.org/10.14778/3213880.3213884

[12] 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* (Denver, CO, USA) *(USENIX ATC '16)*. USENIX Association, USA, 281–294.

[13] A. Broder and M. Mitzenmacher. 2003. Network Applications of Bloom Filters: A Survey. *Internet Mathematics* 1 (2003), 485 – 509.

[14] 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* (New Orleans, Louisiana) *(SODA '04)*. Society for Industrial and Applied Mathematics, USA, 30–39.

[15] Saar Cohen and Yossi Matias. 2003. Spectral Bloom Filters. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data* (San Diego, California) *(SIGMOD '03)*. Association for Computing Machinery, New York, NY, USA, 241–252. https://doi.org/10.1145/872757.872787

[16] Yann Collet. 2012. *xxHash.* Retrieved April 31, 2021 from https://github.com/Cyan4973/xxHash

[17] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2017. Monkey: Optimal Navigable Key-Value Store. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 79–94. https://doi.org/10.1145/3035918.3064054

[18] Martin Dietzfelbinger and Rasmus Pagh. 2008. Succinct Data Structures for Retrieval and Approximate Membership (Extended Abstract). 385–396. https://doi.org/10.1007/978-3-540-70575-8_32

[19] Martin Dietzfelbinger and Stefan Walzer. 2019. Dense Peelable Random Uniform Hypergraphs. *CoRR* abs/1907.04749 (2019). arXiv:1907.04749 http://arxiv.org/abs/1907.04749

[20] Facebook. 2012. *RocksDB.* Retrieved May 31, 2021 from https://github.com/facebook/rocksdb

[21] Bin Fan, Dave 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* (Sydney, Australia) *(CoNEXT '14)*. Association for Computing Machinery, New York, NY, USA, 75–88. https://doi.org/10.1145/2674005.2674994

[22] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. 2000. Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol. *IEEE/ACM Trans. Netw.* 8, 3 (June 2000), 281–293. https://doi.org/10.1109/90.851975

[23] D. Ficara, S. Giordano, G. Procissi, and F. Vitucci. 2008. MultiLayer Compressed Counting Bloom Filters. In *IEEE INFOCOM 2008 - The 27th Conference on Computer Communications*. 311–315. https://doi.org/10.1109/INFOCOM.2008.71

[24] Thomas Mueller Graf and Daniel Lemire. 2020. Xor Filters: Faster and Smaller Than Bloom and Cuckoo Filters. *ACM J. Exp. Algorithmics* 25, Article 1.5 (March 2020), 16 pages. https://doi.org/10.1145/3376122

[25] D. Guo, J. Wu, H. Chen, Y. Yuan, and X. Luo. 2010. The Dynamic Bloom Filters. *IEEE Transactions on Knowledge and Data Engineering* 22, 1 (Jan 2010), 120–133. https://doi.org/10.1109/TKDE.2009.57

[26] Adam Kirsch and Michael Mitzenmacher. 2008. Less Hashing, Same Performance: Building a Better Bloom Filter. *Random Struct. Algorithms* 33, 2 (Sept. 2008), 187–218.

[27] Marcel Kornacker, Alexander Behm, Victor Bittorf, Taras Bobrovytsky, Alan Choi, Justin Erickson, Martin Grund, Daniel Hecht, Matthew Jacobs, Ishaan Joshi, Lenni Kuff, Dileep Kumar, Alex Leblang, Nong Li, Henry Robinson, David Rorke, Silvius Rus, John Russell, Dimitris Tsirogiannis, Skye Wanderman-milne, and Michael Yoder. 2015. Impala: A modern, open-source sql engine for hadoop. In *In Proc. CIDR'15*.

[28] Harald Lang, Thomas Neumann, Alfons Kemper, and Peter Boncz. 2019. Performance-Optimal Filtering: Bloom Overtakes Cuckoo at High Throughput. *Proc. VLDB Endow.* 12, 5 (Jan. 2019), 502–515. https://doi.org/10.14778/3303753.3303757

[29] Daniel Lemire. 2016. *A fast alternative to the modulo reduction.* University of Quebec (TELUQ). Retrieved December 31, 2020 from https://lemire.me/blog/2016/06/27/a-fast-alternative-to-the-modulo-reduction/

[30] J. Lu, Ying Wan, Yang Li, Chuwen Zhang, H. Dai, Y. Wang, Gong Zhang, and B. Liu. 2017. Ultra-Fast Bloom Filters using SIMD techniques. In *2017 IEEE/ACM 25th International Symposium on Quality of Service (IWQoS)*. 1–6. https://doi.org/10.1109/IWQoS.2017.7969125

[31] Chen Luo and Michael J. Carey. 2019. LSM-based storage techniques: a survey. *The VLDB Journal* 29, 1 (Jul 2019), 393–418. https://doi.org/10.1007/s00778-019-00555-y

[32] Lailong Luo, Deke Guo, Richard T. B. Ma, Ori Rottenstreich, and Xueshan Luo. 2018. Optimizing Bloom Filter: Challenges, Solutions, and Comparisons. *CoRR* abs/1804.04777 (2018). arXiv:1804.04777 http://arxiv.org/abs/1804.04777

[33] B. S. Majewski, N. C. Wormald, G. Havas, and Z. J. Czech. 1996. A Family of Perfect Hashing Methods. *Comput. J.* 39, 6 (01 1996), 547–554. https://doi.org/10.1093/comjnl/39.6.547 arXiv:https://academic.oup.com/comjnl/article-pdf/39/6/547/1103380/390547.pdf

[34] Michael Mitzenmacher. 2001. Compressed Bloom Filters. In *Proceedings of the Twentieth Annual ACM Symposium on Principles of Distributed Computing* (Newport, Rhode Island, USA) *(PODC '01)*. Association for Computing Machinery, New York, NY, USA, 144–150. https://doi.org/10.1145/383962.384004

[35] Thomas Neumann and Andreas Kipf. 2019. *Cuckoo Filters with arbitrarily sized tables.* Technical University of Munich. Retrieved December 31, 2020 from http://databasearchitects.blogspot.com/2019/07/cuckoo-filters-with-arbitrarily-sized.html

[36] Rasmus Pagh and Flemming Friche Rodler. 2004. Cuckoo Hashing. *J. Algorithms* 51, 2 (May 2004), 122–144. https://doi.org/10.1016/j.jalgor.2003.12.002

[37] 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* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 775–787. https://doi.org/10.1145/3035918.3035963

[38] Geoff Pike and Jyrki Alakuijala. 2013. *CityHash v1.1.1.* Retrieved December 31, 2020 from https://github.com/google/cityhash

[39] 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* (Snowbird, Utah) *(DaMoN '14)*. Association for Computing Machinery, New York, NY, USA, Article 6, 6 pages. https://doi.org/10.1145/2619228.2619234

[40] Felix Putze, Peter Sanders, and Johannes Singler. 2010. Cache-, Hash-, and Space-Efficient Bloom Filters. *ACM J. Exp. Algorithmics* 14, Article 4 (Jan. 2010), 18 pages. https://doi.org/10.1145/1498698.1594230

[41] David Reinsel, John Gantz, and John Rydning. 2018. The Digitization of the World — From Edge to Core. IDC White paper. https://resources.moredirect.com/white-papers/idc-report-the-digitization-of-the-world-from-edge-to-core

[42] Stefan Richter, Victor Alvarez, and Jens Dittrich. 2015. A Seven-Dimensional Analysis of Hashing Methods and Its Implications on Query Processing. *Proc. VLDB Endow.* 9, 3 (Nov. 2015), 96–107. https://doi.org/10.14778/2850583.2850585

[43] Kenneth A. Ross. 2006. Efficient Hash Probes on Modern Processors. IBM Research Report RC24100 (Nov. 2006). https://dominoweb.draco.res.ibm.com/reports/rc24100.pdf

[44] Christian Esteve Rothenberg, Carlos A. B. Macapuna, Fábio L. Verdi, and Maurício F. Magalhães. 2010. The Deletable Bloom Filter: A New Member of the Bloom Family. *Comm. Letters.* 14, 6 (June 2010), 557–559. https://doi.org/10.1109/LCOMM.2010.06.100344

[45] Christian Esteve Rothenberg, Carlos Alberto Braz Macapuna, and Alexander Wiesmaier. 2009. In-packet Bloom filters: Design and networking applications. *CoRR* abs/0908.3574 (2009). arXiv:0908.3574 http://arxiv.org/abs/0908.3574

[46] Stefan Schuh, Xiao Chen, and Jens Dittrich. 2016. An Experimental Comparison of Thirteen Relational Equi-Joins in Main Memory. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1961–1976. https://doi.org/10.1145/2882903.2882917

[47] Berthold Vöcking. 2003. How Asymmetry Helps Load Balancing. *J. ACM* 50, 4 (July 2003), 568–589. https://doi.org/10.1145/792538.792546

[48] Minmei Wang, Mingxun Zhou, Shouqian Shi, and Chen Qian. 2019. Vacuum Filters: More Space-Efficient and Faster Replacement for Bloom and Cuckoo Filters. *Proc. VLDB Endow.* 13, 2 (Oct. 2019), 197–210. https://doi.org/10.14778/3364324.3364333

[49] Henry S. Warren. 2012. *Hacker's Delight* (2nd ed.). Addison-Wesley Professional.

[50] Jan Wassenberg and Peter Sanders. 2011. Engineering a Multi-Core Radix Sort. In *Proceedings of the 17th International Conference on Parallel Processing - Volume Part II* (Bordeaux, France) *(Euro-Par'11)*. Springer-Verlag, Berlin, Heidelberg, 160–169.