Performance-Optimal Filtering: Bloom Overtakes Cuckoo at High Throughput

Harald Lang Thomas Neumann Alfons Kemper Peter Boncz*
Technical University of Munich Centrum Wiskunde & Informatica*
firstname.lastname@in.tum.de boncz@cwi.nl

ABSTRACT

We define the concept of performance-optimal filtering to indicate the Bloom or Cuckoo filter configuration that best accelerates a particular task. While the space-precision trade-off of these filters has been well studied, we show how to pick a filter that maximizes the performance for a given workload. This choice might be "suboptimal" relative to traditional space-precision metrics, but it will lead to better performance in practice. In this paper, we focus on high-throughput filter use cases, aimed at avoiding CPU work, e.g., a cache miss, a network message, or a local disk I/O – events that can happen at rates of millions to hundreds per second. Besides the false-positive rate and memory footprint of the filter, performance optimality has to take into account the absolute cost of the filter lookup as well as the saved work per lookup that filtering avoids; while the actual rate of negative lookups in the workload determines whether using a filter improves overall performance at all. In the course of the paper, we introduce new filter variants, namely the register-blocked and cache-sectorized Bloom filters. We present new implementation techniques and perform an extensive evaluation on modern hardware platforms, including the wide-SIMD Skylake-X and Knights Landing. This experimentation shows that in high-throughput situations, the lower lookup cost of blocked Bloom filters allows them to overtake Cuckoo filters.

PVLDB Reference Format:
H. Lang, T. Neumann, A. Kemper and P. Boncz. Performance-Optimal Filtering: Bloom Overtakes Cuckoo at High Throughput. *PVLDB*, 12(5): 502-515, 2019.
DOI: https://doi.org/10.14778/3303753.3303757

1. INTRODUCTION

A Bloom filter [5] represents a collection of n keys with an initially-zeroed array of m bits, setting for each inserted key k bits to 1, using as many hash functions to identify the positions $[0,m)$ where the bits are set in the array. This structure allows for fast true-negative tests, but it can produce false-positives at some probability: the **false-positive rate** f. The more recently introduced **Cuckoo filter** [15] offers similar capabilities. It stores small *signatures* – which approximate keys using a few bits – in *buckets* that can hold a few such signatures. The data structure is a Cuckoo hash table of such buckets. A filter lookup checks all signatures in a maximum of two buckets. An advantage of Cuckoo filters over Bloom filters is that they allow deletes as well as duplicates in the key set (thus: bag). Importantly, Cuckoo filters provide a lower false-positive rate f than Bloom filters, given the same size m. In other words, a Bloom filter needs a larger size to reach the same f, and this larger size may increase its lookup cost.

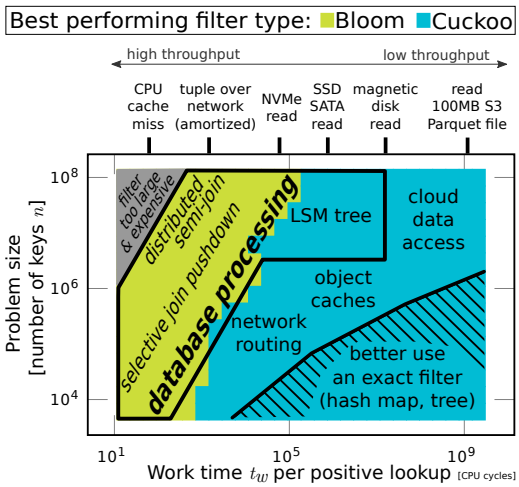


Figure 1: Performance-optimal filter types for different problem sizes n and potential work savings t_w. Example reference points for t_w values are shown above the plot and applications inside the plot.

A popular use case in database systems is **selective join pushdown**. In foreign-key joins between a large (fact) table and a small (dimension) table with a filter predicate that selects a fraction of the dimension tuples, only a fraction σ of the fact table tuples will find a match and contribute to the join result. It can then be beneficial to create a Bloom or Cuckoo filter that contains the n selected dimension keys, and for each fact tuple first test if all k bits of the join key are set in it. If not, the tuple does not join and further work can be eliminated for it, such as a hash table or index lookup, or nested-loop scan. This filtering could be the first step in the join, but the filter test can also be pushed down all the way into the fact table scan, such that the data volume coming out of the scan is reduced, making any intermediate

Proceedings of the VLDB Endowment, Vol. 12, No. 5
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3303753.3303757

operations in between the scan and the join cheaper. Additionally, column stores can skip over data from the non-key columns if whole stretches of tuples test negatively, avoiding (disk or network) I/O and decompression effort.

In selective equi-join pushdowns, Bloom filters are known to significantly enhance query performance in real analytical workloads, as well as in synthetic ones, such as TPC-H and TPC-DS [6]. However, using a Bloom or Cuckoo filter can potentially backfire if it does not eliminate (enough) join candidates, certainly in cases where the selectivity σ=1.0 (no negative lookups), or values of σ close to that. A way to deal with this is to use cardinality estimation to gauge σ and n at query optimization time, in order to decide whether to create a filter at all, and if so, with which parameters. Alternatively, some database systems monitor the join hit-rate σ of hash or index joins at *run-time*, and add a filter if σ is below a given threshold [33]. This has the advantage that by then n is known (e.g., the hash table has already been built – in the case of a hash join), allowing us to better choose the filter size m, as well as parameters such as the k for Bloom filter and the signature and bucket size (l, b) for Cuckoo filters.

A Bloom filter is a well-known data structure also used in many systems outside of databases, notably including network routers and caches of all sorts. Beyond our leading example of selective equi-join pushdown, other uses *inside* database systems include key-value indexes based on multiple structures/runs such as log-structured merge-trees [12] (in order to reduce the number of structures to search), cold storage structures (idem) [2], and distributed-semijoin optimization for exchange operators in MPP systems, which first broadcast a Bloom filter across compute nodes to avoid exchanging unneeded join-probe tuples over the network [21]. Our work applies to all filter usage scenarios.

The common thinking is that if the n is known, space-optimal Bloom filter parameters can be calculated based on theory [25], given a desired false-positive rate f, namely $k = -\log_2 f$ and $m = 1.44kn$. However, our argument is that a minimal size m given an f or vice versa is not a goal in itself, rather, the goal is to optimize performance.

In defining **performance-optimal** filtering, we introduce a model to optimize this *overall* performance and answer the crucial question: *what filtering data structure and parameters best accelerate a particular workload?* To determine what is performance optimal, we have to take into account the additional factors: (i) the actual time t_l a filter lookup takes, (ii) the work time t_w per tuple that a negative lookup identified by the filter saves later on, and (iii) the mentioned fraction 1-σ of real negative lookups (regardless of false positives). The issue is how much t_l to invest (in lookup time) and how much t_w (work time) this pays off and how often this pays off (1-σ).

Systems that incorporate key filtering need to decide on the above question, and its answer is not clear, which is the reason why we performed this research. Our work focuses on filtering techniques that provide both low f but also have low lookup time t_l, and builds on the cache-efficient and hash-efficient blocked Bloom filter work of Putze et al. [31]. In doing so, we introduce two new Bloom filter variants, namely the *cache-sectorized* and *register-blocked* ones.

Further, we take fast **implementation techniques** into account. This includes SIMD `GATHER` instructions as well as the many-core Knights Landing architecture with wide SIMD. Fast implementations only use m values that are powers-of-two, such that modulo can be computed with bitwise `AND`. This means that theoretically optimal values of m typically cannot be chosen, leading to an m that in the worst case is a factor $\sqrt{2} \simeq 1.44$ off (average case 1.22). Instead, we provide fast SIMD implementation techniques that allow almost any size m to be used. Our implementations scale from filters that fit a small cache to filters that are GBs in size. We release all our code and experiments in open-source for reproducibility and re-use.

Comparing the overall performance of two filter configurations, a decrease in false-positive rate (Δ_f) only pays off if the extra work saved ($\Delta_f * t_w$) exceeds the increase in lookup cost (Δ_l), i.e. if $\Delta_f * t_w > \Delta_l$. We will show that in all realistic selective join workloads, blocked Bloom filters with worse false-positive rate outperform Cuckoo filters, due to their lower lookup cost.

Figure 1 summarizes our key findings from our detailed experiments with regard to which filter type, Bloom or Cuckoo, performs best for a given problem size n and the potential savings t_w. In high-throughput situations (left side, low t_w), Bloom filters are to be preferred over Cuckoo filters. Bloom filters offer lower lookup costs but also a higher false-positive rate. In high-throughput scenarios, the costs induced by a false-positive result is relatively low and the fast lookups are the dominant factor. In contrast, low-throughput scenarios (right side) require higher accuracy as the costs induced by a false positive are significantly higher than the actual filter lookup. Database processing use cases for filter structures often involve high-throughput lookups where, e.g., filtering just avoids a CPU cache miss caused by a hash lookup, but also have use cases where filters are used to avoid more expensive accesses (right side: lower-throughput workloads), like magnetic disk seeks, e.g. into log-structured merge-trees on hard disk. These higher t_w use cases are the areas where Cuckoo filters dominate, due to their lower false-positive ratios. In those cases, though, if the problem size is small (low n), then false positives can and should be avoided entirely by using an exact data structure instead.

Our contributions are:

- a formal definition of performance-optimal filtering;
- improved filter variants: *register-blocked* Bloom filters and *cache-sectorization* of blocked Bloom filters;
- consideration of advanced implementation techniques, allowing us to use filter sizes beyond just powers-of-two, and AVX2/AVX-512 SIMD hardware optimizations for Bloom **and** Cuckoo filters;
- extensive experiments that allow us to establish that blocked Bloom filters overtake Cuckoo filters when the work saved t_w by negative lookups is low or moderate: we call this the **high-throughput** use cases;
- open-source implementations for all filter structures[1].

2. PERFORMANCE-OPTIMAL FILTERING

Figure 2 shows the selective join pushdown scenario. The query contains a join between a fact table (probe pipeline) and a dimension table (build pipeline), and may proceed above the join, e.g. with an aggregation.

[1]Source code: https://github.com/peterboncz/bloomfilter-repro

The query cost c can be divided as $c = c_{build}+c_{work}+c_{post}$: the cost to build the hash table, the time to run the pre-join pipeline up until the join lookup itself, and the time to run the rest of the query, including join result generation. Note that $c_{work} = |R|*t_w$, that is, t_w is the *per-tuple* execution time of the probe pipeline.

Installing a Bloom filter in the scan at the bottom of the probe pipeline will reduce the data volume flowing through it to a factor $\sigma+f$. Overall, this can accelerate the query maximally (if $\sigma=f=0$) by a factor $c/(c_{build}+c_{post})$, however we will ignore this in the rest of the paper, focusing on the filter with *most* performance impact – however high it is.

In order for a query optimizer to decide on installing a (Bloom) filter in a join, it needs to estimate t_l, t_w, f, and σ. In order to estimate t_l and f, it would need to estimate the amount of build-side keys n, which can be done using logical cost (cardinality) estimation. There are well-studied methods to do this, but cardinality estimation can still be off. Furthermore, t_l and t_w are physical (per tuple) costs, which are much harder to estimate correctly [23] because they estimate hardware behavior and may even be impacted by external factors (such as concurrent workload or even temperature). The alternative strategy of installing a filter at runtime, after running the probe pipeline for a while, has the advantage that t_w, n and σ are known. The performance-optimal filter F, out of all possible filter configurations $\mathcal{F}$, is the one that minimizes the per-tuple work using filtering $t_{w'}(F)$:

$$t_{w'}(F) = (1-\sigma')*t_l^-(F) + \sigma'*(t_l^+(F)+t_w) \text{ with: } \sigma' = \sigma+f(F)$$

where we use $f(F)$ to denote the false-positive rate f achieved by F. We split up the lookup cost t_l of F for the case of a lookup that finds a hit and when it does not $t_l^-(F)$. This is needed for classic Bloom filters, because they test the k bits one-by-one and break off search as soon as a bit is not set. In classic Bloom filters with a low load factor (i.e., it contains mostly zeros), most negative queries will already test negatively on the first bit, therefore typically only one hash function needs to be computed and only one cache line will be accessed. For positive queries, however, classic Bloom filters need to compute all k hash functions and perform k memory accesses ($t_l^+ \gg t_l^-$), making them expensive if k is significant, and more so if this happens often (largish σ).

Most other filter algorithms that we study – including Cuckoo – are implemented such that they do an equal amount of work for positive and negative queries ($t_l^+ = t_l^- = t_l$) and only access one or two cache lines. Furthermore, classic Bloom filters are hard to SIMDize and are thus computationally more expensive than SIMDizable variants (such as the register-block Bloom filter). A SIMD version of classic Bloom filters was implemented [29], but in the many experiments we performed, it was never performance optimal. As the performance-optimal filtering algorithms *do* exhibit equal performance for positive and negative queries, we can simplify the definition of the performance-optimal filter F^{opt} to the one with least overhead:

$$F^{opt} \in \mathcal{F} : \nexists F \in \mathcal{F} : \rho(F) < \rho(F^{opt}) \quad \text{with: } \rho(F) = t_l(F) + f(F) * t_w \tag{1}$$

Here $\rho(F)$ is the overhead of all filtering (i.e., filter lookup and false-positive work). Parameter σ is still needed, but only to decide if filtering is beneficial at all, namely whether: $\rho(F^{opt}) < (1-\sigma) * t_w$.

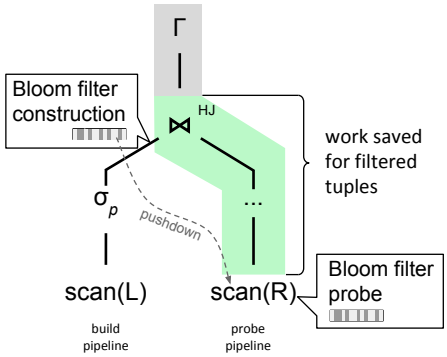


Figure 2: Bloom filters with selective joins. Tuples without a join partner are filtered before entering the pipeline.

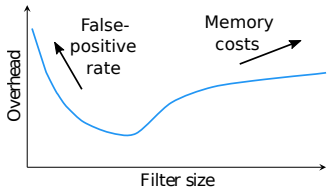


Figure 3: Overhead ρ as a function in m for fixed configuration F, n and t_w.

But how to determine F^{opt}, that is, to find the best combination of lookup cost t_l and f given a t_w and n? For instance, in blocked Bloom filters, the configuration parameters are k (the more hash functions, the higher t_l but typically the lower f) and m (the larger the size, the lower f becomes, but due to more cache and TLB misses, t_l may increase). Figure 3 sketches the overhead ρ as a function in m. If the filter size is set too small, the bitvector of a Bloom filter gets "overly populated" and f increases. On the other hand, if the size chosen is too large, cache miss probability increases, making lookups more expensive.

For the influence of k, m and n on f, a numerical model is available [25] using a Poisson approximation. However, t_l is a physical cost metric and is harder to predict, as it depends on the hardware. We therefore propose to collect the actual filter lookup costs by performing microbenchmarks on the target platform as part of a one-time calibration phase.

3. BLOOM FILTER VARIANTS

As the previous section suggests, there is a very large space of filter variants and possible configurations. To achieve performance optimality it is necessary to understand the individual properties of the filter instances and how these properties affect performance for a given problem setting. To quantify the performance-related aspects, we consider the *precision* given by the false-positive rate f, the *space efficiency* (i.e., the memory footprint) as well as the *computational efficiency*, which refers to the CPU work of lookups and the *memory bandwidth efficiency*. All four dimensions are correlated. For instance, tuning for space efficiency may come at the cost of additional computations and reduced precision, but also with a better bandwidth efficiency.

In the rest of this section, we explore the design space of Bloom filters and their respective positioning within the four dimensions.

3.1 Blocking

A *blocked* Bloom filter as proposed in [31] is a Bloom filter that is split into equally sized blocks. Each block is a small Bloom filter. The size of a block, denoted as B, is proposed to be equal to the size of a cache line, which is $B = 512$ bits on the x86 architecture. All k bits of an inserted key are set within a single block and each insert or lookup results in one single cache miss at most. Because a cache line is the unit of memory transfer and all bits are spread across the entire cache line, it can be considered optimal with regard to memory bandwidth efficiency.

The second advantage of blocked Bloom filters is that fewer hash bits are required per key and they therefore have improved computational efficiency as compared to classic Bloom filters. Blocking reduces the number of required hash bits from $k \cdot \log_2(m)$ down to $k \cdot \log_2(B)$ plus $\log_2(m/B)$ bits to address the corresponding block. Listing 1 shows the lookup function in pseudocode.

The improved bandwidth and computational efficiency comes at the cost of reduced accuracy (higher f). Every block acts as a classic bloom filter of size B, thus the false-positive rate is known to be

$$f_{\text{std}}(m, n, k) = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k, \tag{2}$$

where m is set to the block size B. The important thing to note here is, that not all blocks contain the same amount of keys. In other words, the (block-local) n varies among the different blocks and affects the overall f of the filter. The load that the individual blocks receive is binomially distributed, and [31] provides the following approximation for the false-positive probability:

$$f_{\text{blocked}}(m, n, k, B) = \sum_{i=0}^{\infty} \text{Poisson}\left(i, B\frac{n}{m}\right) * f_{\text{std}}(B, i, k) \tag{3}$$

where f_{std} denotes the false-positive rate of a classic Bloom filter with the arguments m, n and k (see Equation 2).

Register Blocking: For high-throughput scenarios, we consider an extreme case of blocking, where the block size is reduced to the size of native CPU registers, namely 64-bit or 32-bit. These *register-blocked* Bloom filters significantly reduce computational efforts, as all k bits can be tested in a single comparison and only one processor word needs to be loaded (see Listing 2). Thus, since only one processor word is accessed per lookup, a register-blocked filter can no longer be considered as being memory bandwidth efficient. Effectively, only $1/8^{\text{th}}$ or $1/16^{\text{th}}$ of a cache line is accessed for 64-bit and 32-bit blocks, respectively. Therefore, register blocking is a technique to trade memory bandwidth efficiency and precision for further increased computational efficiency, which is particularly important for CPU-cache resident filters.

Impact of blocking: Figure 4a illustrates how blocking affects the false-positive rate f depending on the bits-per-key rate (m/n). We compare a space-optimal classic Bloom filter (blue line) with register-blocked Bloom filters (red and orange lines), and a cache line blocked filter (green line). Figure 4b shows the corresponding values for k, which indicates the computational efforts caused by hashing.

The smaller the block size, the higher f becomes. This increase in f can be *compensated* to some degree by increasing the filter size m and the bits-per-key rate, respectively. For instance, a classic Bloom filter with $f = 1\%$ requires ≈ 10 bits per key of memory, whereas a register-blocked Bloom filter requires ≈ 12 or 14 bits per key for block sizes of 64 and 32 bits. However, depending on the desired f, the memory footprint of the filter quickly becomes impractical. For instance, with $B = 32$, a false-positive rate of 0.1% requires 32 bits per key, which is significant memory consumption, and an exact data structure, e.g., a hash map might be a better choice.

```
// Block addressing
h = consume log2(m/B) hash bits
block_idx = h mod m/B
found = false
for each k do {
  // Word addressing (W denotes the size of a word)
  h = consume log2(B/W) hash bits
  word = load word from block_idx + h
  h = consume log2(W) hash bits // Bit addressing
  found |= word & (1 ≪ h) // Bit testing
}
return found
```

Listing 1: Lookup function of blocked Bloom filters.

```
// Block addressing
h = consume log2(m/B) hash bits
block = load word at position h mod (m/B)
search_mask = 0;
for each k do {
  // Bit addressing
  h = consume log2(B) hash bits
  search_mask |= 1 ≪ h
}
return block & search_mask == search_mask // Bit testing
```

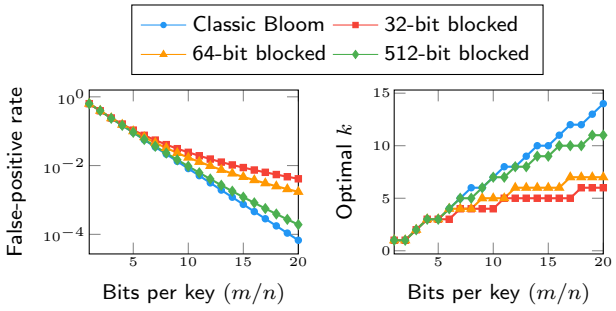
Listing 2: Lookup function of register-blocked Bloom filters.

3.2 Sectorization

With an increasing t_w, the false-positive rate f of a filter becomes increasingly important and it therefore becomes necessary to increase the block size beyond a single processor word. If the bits are distributed over multiple words, we can no longer test multiple bits in one comparison instruction (compare Listing 1 and 2), which significantly reduces CPU efficiency. In this section, we present *sectorization* of blocked Bloom filters to address this inefficiency. For clarity, we first present the key idea of sectorization, followed by an extension which we call *cache-sectorization*. The latter can compete with Cuckoo filters even for large filters that exceed the CPU cache size.

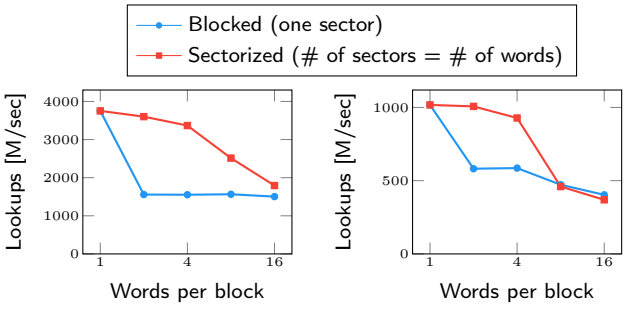
To the best of our knowledge, sectorization (in a primitive form) was first used in the SIMD Bloom filter of the Impala database system [21]. The authors actually combined a *m/k-partitioning* scheme [20] with blocking. However, this technique has not been further investigated with regard to performance and false-positive rate. In the following, we catch up on this by discussing the upsides and downsides of sectorization and further provide a formula for the false-positive rate.

Sectorization is a partitioning scheme that sub-divides blocks into equally sized partitions, which we call *sectors*, and the k bits, set for each key, are equally distributed



(a) (b)

Figure 4: Impact of blocking on the false-positive-rate and the hashing efforts.



(a) Cache (16 KiB filter) **(b)** DRAM (256 MiB filter)

Figure 5: Performance impact of sectorization for varying block sizes.

among all sectors. This partitioning scheme has the following advantages:

1. It reduces the number of required hash bits and therefore reduces computation efforts caused by hashing.
2. It can change the *random access* within a block to a *sequential access* pattern which greatly improves CPU efficiency.

To exemplify sectorization, we set the block size to 512 bits and the sector size to 64 bits. Furthermore, we assume that a native processor word has 64 bits (as in x86_64) and is therefore equal to the size of a sector. Hence, a block is a sequence of s=8 words which can be processed *sequentially* and *independently* by setting the first $k/8$ bits in the first word, the next $k/8$ bits in the second word, and so on. Each word has to be read exactly once and multiple bits can be tested at once, similar to register blocking (see Listing 2).

Formally, we let S denote the *sector size* and the (capitalized) K the number of bits set per key *per sector*, where $1 \leq S \leq B$. As sectorization aims for CPU efficiency in Bloom filter implementations, we restrict the block size as well as the sector size to be a power of two and k needs to be a multiple of the number of sectors B/S.

Sectorization generalizes the (prior) definition of a blocked Bloom filter. I.e., if we set $S := B$, then the individual blocks consist of exactly s=1 sector and it implies that $k = K$. The filter instance is therefore equivalent to a blocked Bloom filter as defined by Putze et al. in [31]. In contrast to m/k-partitioning as proposed by Kirsch et al. [20], sectorization is applied at block level and therefore preserves the locality of a blocked Bloom filter. Further, our scheme is more flexible because it allows us to set multiple bits per partition (sector), which improves CPU efficiency.

Figure 5 shows a performance comparison of a blocked Bloom filter with and without sectorization with $k = 16$ (on a Xeon E5-2680v4 using 28 threads). The leftmost data points refer to register blocking (where one block = one word). We then double the block sizes until we reach the size of a cache line. Immediately, when a block exceeds a single word, the lookup performance drops significantly by $\approx 60\%$ for cache-resident filters and by $\approx 50\%$ for larger filters, because we have to use the first lookup algorithm from Listing 1 (using a random access pattern). On the other hand, with sectorization enabled, the performance degrades gracefully with increasing block sizes. The important thing to note here is that the sector size is set to word size and it remains constant. We only increase the number of sectors with the block size, which is the enabler for using the more efficient lookup algorithm from Listing 2 within each sector (in a sequential order).

Technically, the sector size can be set to any power of two (as long as $S \leq B$), but it must be less than or equal to the word size in order to get a sequential block access pattern. In most cases, setting the sector size to the word size (either 32-bit or 64-bit) is the best option. In rare cases, splitting a word in multiple sectors may improve the false-positive rate, which we describe later. In the remainder of the paper, we set the sector size to the word size, unless stated otherwise.

The downside of sectorization is that the number of ks needs to be a multiple of the number of sectors. In other words, we need at least as many ks as we have sectors. And with regard to CPU efficiency, we would prefer to set/test multiple bits per sector. Thus, it is desirable to have higher ks per sector. For instance, if we use 32-bit words and a block size $B = 512$ bits, we need at least 16 sectors and consequently at least $k = 16$, which is already a very high value for k (not only for high-throughput scenarios). If we also want to set/test multiple bits at once, we have to increase k to unreasonably high values of 32, 48, etc. With these limitations, it is hardly possible to find the right balance between low false-positive rates, high memory bandwidth efficiency, and high CPU efficiency. The Impala implementation, for instance, uses the (hard-coded) configuration $k = 8$, $S = 32$, and $B = 256$ [2], where only the filter size m can be adjusted. It therefore leaves room for optimizations.

To address these limitations, we propose an extension to sectorization that offers more flexible parameterization and is therefore more tunable to a wide variety of problem settings. We call our approach **cache-sectorization**. The design goal is to distribute the bits over entire cache lines but also support lower ks.

Cache-sectorization works as follows: Blocks are partitioned in word sized sectors. Multiple sectors are then logically grouped together. When a key is inserted, we set k/z bits in each group, where z is the number of groups per block. Inside each group, the k/z bits are set in one sector, which is determined by the key's hash value. Figure 6 illustrates the cache-sectorization block partitioning. Per key, z words are accessed, and all words belong to the same cache line. Inside each group, we now have a dependent load

[2]The configuration is ideal for AVX2 SIMD using 256-bit registers.

which makes the access pattern less optimal. However, this is amortized by accessing fewer words per block. Further, across the groups, all operations remain independent and can be performed in parallel. In contrast to sectorization, k can be chosen more flexibly, as a multiple of z instead of s, where $z < s$.

False-positive rate: Cache-sectorization can improve the false-positive rates as compared to sectorization. In Figure 7, we compare both variants: The blue line represents the sectorized variant that spreads the bits across 4 words, resulting in 4 loads per lookup. The cache-sectorized version (red line), also accesses 4 words per lookup but spreads the bits across an entire cache line, which results in a significantly lower false-positive rate. If we further reduce the number of accessed words (orange line), we can improve the lookup performance with f similar to the sectorized variant. For reference, the dashed lines show the false-positive rates of (register-)blocked filters without sectorization, with $B = 32$ and $B = 512$, respectively.

We provide formulas for the false-positive rate for both sectorized variants:

$$f_{\text{sector}}(m,n,k,B,S) = \sum_{i=0}^{\infty} \text{Poi}\left(i, B\frac{n}{m}\right) * \left(f_{\text{std}}\left(S, i, \frac{k}{s}\right)\right)^{s} \tag{4}$$

$$f_{\text{cache}}(m,n,k,B,S,z) = \sum_{i=0}^{\infty} \text{Poi}\left(i, B\frac{n}{m}\right) * \left(\sum_{j=1}^{i} \text{Poi}\left(j, S\frac{i*z}{B}\right) * f_{\text{std}}\left(S, j, \frac{k}{s}\right)\right)^{z} \tag{5}$$

Our experimental analysis discussed in Section 6 shows that a Bloom filter with cache-sectorized blocks can compete with Cuckoo filters and outperforms standard (non-sectorized) blocked Bloom filters.

4. CUCKOO FILTER

With the Cuckoo filter [15], Fan et al. presented an alternative to Bloom filters which claims to be "practically better" in terms of lookup performance and space consumption. A Cuckoo filter is a variation of a cuckoo hash table [27] with two major differences:

1. It stores small *signatures* (aka fingerprints) instead of the entire keys, while every hash bucket can hold multiple of such signatures.

2. For collision resolution, the alternative bucket of an entry is determined based on the key's signature and its current bucket index, instead of using two independent hash functions.

The signature of a key is computed using a hash function. Typically, the low-order bits of the hash value are used as the signature. Inside the cuckoo hash table, each signature has two candidate buckets in which they can be stored. The indexes of the two buckets i_1 and i_2 of a key x are calculated as follows:

$$\begin{aligned} i_1 &= \text{hash}(x) \\ i_2 &= i_1 \oplus \text{hash}(x\text{'s signature}) \end{aligned} \tag{6}$$

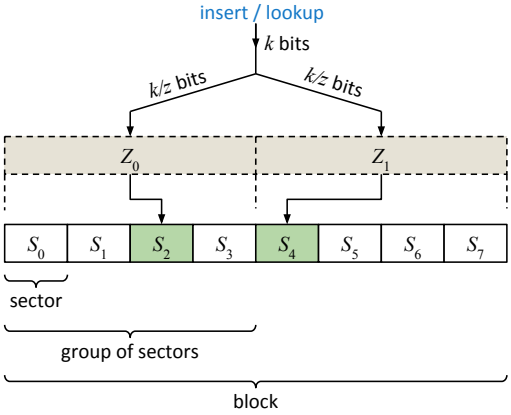


Figure 6: Block partitioning scheme of cache-sectorized Bloom filters: hashing sets bits concentrated in z words spread over a cache line.

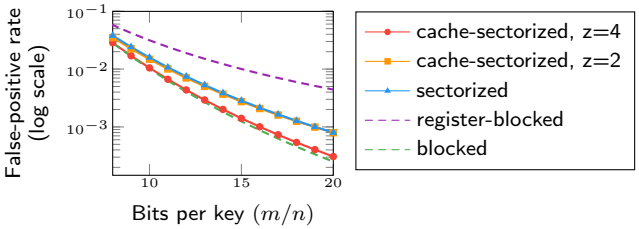


Figure 7: Comparison of the false-positive rate of sectorized and cache-sectorized Bloom filters, with $k = 8$.

The noteworthy property is that the index i_1 can also be computed using the key's signature and the index i_2:

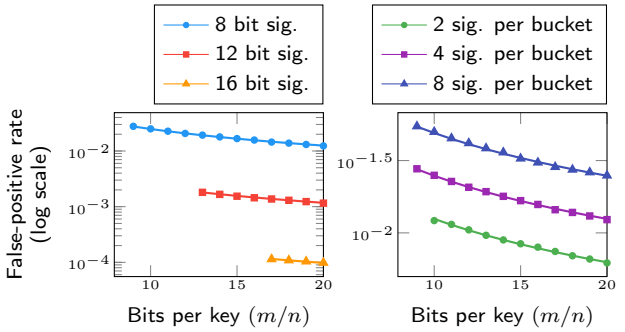
$$i_1 = i_2 \oplus \text{hash}(x\text{'s signature}) \tag{7}$$

This property allows to determine the alternative bucket of a signature without having access to the actual (unhashed) key value, which is not stored in the cuckoo hash table. This technique, which the authors refer to as *partial-key cuckoo hashing*, is necessary to relocate signatures inside the table in case of collisions. Whenever a signature cannot be stored, due to fully occupied buckets, a signature in one of the two target buckets is randomly chosen and relocated to its alternative bucket. The fact that the alternative bucket index is a combination of the signature and the first bucket index (and vice versa) results in a lower table occupancy (~50%) compared to a cuckoo hash table that is using two independent hash functions. The authors address this issue by storing multiple signatures per bucket. For instance, using a bucket size $b = 2$, 4, or 8 increases the table occupancy to 84%, 95%, or 98%, respectively. However, this approach also negatively affects the accuracy, as we describe in the following paragraph. During a lookup, the key is hashed to compute the signature and to determine both candidate buckets. Afterwards, both buckets are searched for that signature.

The false-positive probability of a Cuckoo filter is

$$f_{\text{cuckoo}}(\alpha, l, b) = 1 - \left(1 - \frac{1}{2^l}\right)^{2b\alpha}, \text{ with: } \alpha = \frac{l*n}{m} \tag{8}$$

where l is the signature length in bits and α the load factor of the table. Thus, the false-positive rate primarily depends



(a) Four sig. per bucket. **(b)** Sig. size fixed to 8 bits.

Figure 8: The false-positive rate of Cuckoo filters for different signature lengths and bucket sizes.

on the signature size l. The longer the signatures, the lower the false-positive rates. Typically, l is between 8 and 16 bits. Increasing the filter's size (i.e., lowering the load factor α) only gradually improves the false-positive rate, as shown in Figure 8a. On the other hand, reducing the numbers of signatures per bucket b significantly improves the false-positive rate (see Figure 8b), while coincidentally impairing memory efficiency due to lower load factors.

A noteworthy property of a Cuckoo filter is that an insertion may fail if the target buckets are fully occupied and the signatures cannot be relocated. This is in contrast to Bloom filters, where insertions always succeed.

5. IMPLEMENTATION TECHNIQUES

For our evaluation we implemented a blocked Bloom filter that is optimized for high-throughput scenarios where lookups are performed in batches. Our implementation is generic in the sense that it allows us to vary the block size, sector size, and naturally the number of hash functions as well. Even though our implementation is generic, the genericity does not induce any runtime costs as it is mostly written in C++ template language. All parameters are compile-time static except the size of the filter m.

Further, we revised and extended the original Cuckoo filter implementation and unified the interface of all filters under test with regard to *batched* lookups. I.e., the contains functions take an entire *list of keys* at once and produce a *position list* (also called a selection vector) consisting of 32-bit integers. As this work focuses on high-throughput scenarios, we use multiplicative hashing for both, Bloom and Cuckoo filters.

5.1 Data Parallelism

The performance-critical *contains* functions make extensive use of SIMD instructions (i.e., from the AVX2 and the AVX-512 instruction set). SIMD is primarily used to execute multiple lookups in parallel which allows the average number of CPU cycles per lookup to be reduced to less then two cycles (for low ks). Our actual C++ implementations of the Bloom filter contains functions are very similar to the scalar pseudocode in Listings 1 and 2. This also applies for the SIMDized versions, as we used an abstraction layer for the SIMD vector types and vector instructions. This allows us to perform one lookup per SIMD lane, whereas each SIMD lane operates on 32-bit words. It also allows us to easily scale to broader SIMD registers. For instance, the C++ implementations for AVX2, which performs eight simultaneous lookups, is the same as for AVX-512, which performs 16 lookups in parallel. Please note that this technique relies on the `gather` instruction and we therefore do not support pre-AVX2 architectures. It is also noteworthy that the SIMD abstractions do not incur any runtime costs. Each contains function (one per filter configuration) is compiled into a branch-free instruction sequence.

Similarly, we optimized the Cuckoo filter implementations to perform parallel lookups. In contrast to the Bloom filter, the Cuckoo filter implementation is less generic. It requires a separate code path for each signature length, and not all signature lengths are "SIMD friendly". Some may result in unaligned memory accesses. – Please note that this also applies for non-SIMD implementations. – We therefore optimized only the (SIMD-friendly) instances with 8-, 16- and 32-bit signatures.

Modern processors differ greatly in their SIMD capabilities and their out-of-order execution capabilities [1, 18]. We observed significant performance differences across various platforms (Xeon, Knights Landing, Skylake-X and Ryzen) with a single SIMD implementation. To address this issue, we instantiate the filter templates with multiple parameters with respect to the vector lengths and unrolling factors and perform a short *calibration phase* at library installation time which allows us to select the best performing instantiation at runtime. The calibration is done only once per platform and in the worst case, or if the underlying platform is of a pre-AVX2 generation, the scalar (non-SIMD) code is used as a fallback.

5.2 Magic Modulo

A common optimization technique is to size data structures to powers of two to avoid costly modulo operations and substitute them by bitwise `AND`s (this applies, for example, to the Impala Bloom filter, the SIMD Bloom filter from [29], and to the reference implementation of the Cuckoo filter). I.e., the operation `hash(key) mod m`, which involves an integer division, is several times slower than using `hash(key) & mask` (with `mask := (1≪log2(m))-1`). However, our experimental analysis shows, that this approach is very inflexible and leaves large potential for optimizations.

Especially for SIMD, there is no satisfactory solution to sizing data structures more flexibly. Even an inefficient modulo operation is not possible, because modern SIMD instruction sets do not support integer division. Putze et al. [31] therefore proposed to perform the division with floating-point arithmetic. Even though, the floating-point division on Intel vector processing units is still an expensive operation, i.e., 13 cycles on Haswell, the operation is applied to eight elements in parallel. If we take the necessary type conversions into account, a division of eight elements takes 15 cycles, which is an improvement of approximately 6× over scalar code. For our evaluation, we implemented an approach known from the field of compiler construction, which performs the same operation in approximately 10 cycles.

Modern compilers substitute the costly modulo operations (more precisely, the involved integer division) with a cheaper instruction sequence consisting of a multiply, a shift, and an addition. Based on the divisor, a compiler determines a *magic number* [19] to multiply with, a shift amount and a summand. On most platforms, the multiply-shift-add se-

quence is faster than an integer division. Naturally, the compiler can only optimize if the divisor is known at compile time, which is not the case with dynamically-sized data structures such as, in our case, the Bloom or Cuckoo filters. We therefore re-implemented this approach manually to support (almost) arbitrary filter sizes. We further optimized the magic number approach to substitute the integer division with a multiply-shift instruction sequence, without the trailing addition, saving one additional instruction. The enabler for this optimization is, that the magic numbers for (unsigned) division can be categorized into two classes: (i) those which require multiply-shift-add instructions and (ii) those which only require a multiply and a shift. Which instruction sequence to use depends on the divisor. In our context, we can (slightly) vary the size of the data structure. This additional degree of freedom allows us to choose a divisor that belongs to the second class and to save the trailing addition. We refer to this approach as *magic modulo*. A modulo operation $i = \text{hash}(key) \bmod C$ is thereby replaced by

$$\begin{aligned} h &= \text{hash}(key) \\ i &= h - (\text{mulhi\_u32}(h, magicNo) \gg shiftAmount) * h, \end{aligned} \tag{9}$$

whereas the function mulhi_u32 multiplies two 32-bit integers, producing a 64-bit intermediate, and returns the upper 32 bits of the product.

Magic modulo is used to determine a block[3] of the Bloom filter and a bucket in the Cuckoo filter, respectively. The *actual* filter size is therefore

$$m_{\text{actual}} = x * \text{nextMagicNo}\left(\left\lceil \frac{m_{\text{desired}}}{x} \right\rceil\right) \tag{10}$$

with $x := B$ for Bloom filters and $x := l*b$ for Cuckoo filters. In our implementation, which supports up to 2^{32} blocks, the actual number of blocks is at most 0.0134% higher than the desired number of blocks or buckets, respectively. Naturally, magic modulo is more expensive than a single bitwise `AND` which is not in favor of Cuckoo filters, because the indexes of *two* buckets need to be computed. Further, the `XOR` operation of the partial-key cuckoo hashing (see Equation 6) needs to be replaced by a different and slightly more expensive self-inverse function. In our implementation, the bucket indexes of a key x are computed as follows:

$$\begin{aligned} i_1 &= \text{hash}(x) \text{ magicMod } C \\ i_2 &= -(i_1 + \text{hash}(x\text{'s signature})) \text{ magicMod } C \end{aligned} \tag{11}$$

where C denotes the number of buckets.

Figure 9 illustrates the benefits of magic modulo by using an example of a cache-sectorized Bloom filter ($k = 8, B = 512, z = 2$). Magic modulo allows to vary the filter size in very small steps (purple line) compared to the power-of-two sizes (blue dots). At cache boundaries, there is a wide range where this flexibility improves lookup performance ①. The range, where the performance degrades over power-of-two modulo is relatively small ②, because magic modulo has only a modest overhead ③. With an increasing filter size (e.g., a multiple of the last-level cache size), magic modulo becomes less beneficial with regard to performance ④, but still gives better control over the memory consumption. The same applies for very small L1- or L2-resident filters.

[3]The Bloom filter block sizes are powers of two and therefore, inside a block, bitwise `AND` instructions are used to determine the bit positions.

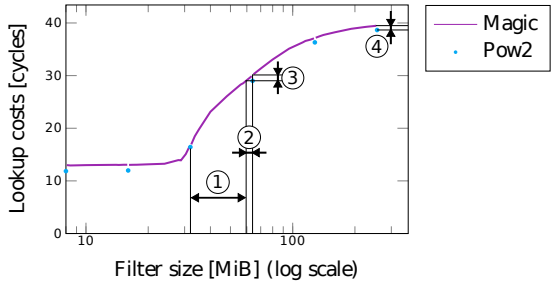


Figure 9: Lookup performance of a cache-sectorized Bloom filter for varying filter sizes.

Table 1: Hardware platforms

	Intel Xeon	Intel Knights Landing	Intel Skylake-X	AMD Ryzen
model	E5-2680v4	Phi 7210	i9-7900X	1950X
cores (SMT)	14 (x2)	64 (x4)	10 (x2)	16 (x2)
SIMD instr.	AVX2	AVX-512[1]	AVX-512[2]	AVX2
SIMD [bit]	2×256	2×512	2×512	256
freq. [GHz]	2.4 – 3.3	1.3 – 1.5	3.3 – 4.5	3.4 – 4.0
L1 cache	32 KiB	64 KiB	32 KiB	32 KiB
L2 cache	256 KiB	1 MiB	1 MiB	512 KiB
L3 cache	35 MiB	-	14 MiB	32 MiB
launch	Q1'16	Q4'16	Q2'17	Q3'17

[1] AVX-512{F,CD,ER,PF}
[2] AVX-512{F,DQ,CD,BW,VL}

6. EXPERIMENTAL ANALYSIS

In this section, we present the results of our experiments, conducted on four different hardware platforms (see Table 1). We tested many different problem sizes (n) and ran experiments on the different hardware platforms, varying all relevant parameters for each filter data structure. For Bloom filters, we considered values for k in $[1, 16]$, B in $\{4, 8, 16, 32, 64\}$ bytes, S in $\{1, 2, 4, 8, 16, 32, 64\}$ bytes, W in $\{32, 64\}$ bits and z in $\{2, 4, 8\}$. For Cuckoo filters, we varied l in $\{4, 8, 12, 16\}$ bits, and b in $\{1, 2, 4\}$. As the data set we used random 32-bit integers (uniformly distributed) generated with the Mersenne Twister engine from the C++ Standard Template Library. To get stable results, we repeated each measurement five times and report the average. This resulted in more than 15 million experiments that we performed on all these possible filter configurations. Throughout all experiments we used the GCC compiler (version 5.4.0) with optimization level set to `-O3`.

Unless stated otherwise, we present multi-threaded results, using one thread per core; except for KNL with 4-way hyper-threading, we ran two threads per core. Even though, all experiments ran on a single processor, we had to take NUMA effects into account. KNL and Ryzen are NUMA architectures, with four and two nodes, respectively. On these platforms, we replicated the filter data to all NUMA nodes and let all threads query the NUMA-local filter. The probe data (256 MiB), on the other hand, was distributed across all nodes in a round-robin fashion.

Skylines. For each valid[4] filter configuration $F \in \mathcal{F}$ we scaled the problem size n from 2^{10} to 2^{28} keys. More precisely, we used the values $n_{i,j} = \lfloor 2^{i+j*0.0625} \rfloor$ with i in

[4]Please note that not all configurations are valid. For instance, setting $B := 64$ and $S := 512$ is illegal, as the sector size may not exceed the block size.

$[10, 27]$ and j in $[0, 15]$. For each $\langle F, n\rangle$ pair, we scaled the filter size m between $4n$ and $20n$, thus limiting the bits-per-key rate to 20. The values of m are also scaled exponentially, containing all powers of two and nine intermediates in between. For each experimentally collected data point, we *compute* the overhead $\rho(F)$ for 28 different t_w values. For the t_w values, we use 2^i with i in $[4, 31]$. From the resulting data set, we determined for each $\langle n, t_w\rangle$ point the performance-optimal filter configuration F^{opt} with the smallest overhead. By doing this for all these points, we obtain a **skyline** of performance-optimal filter configurations.

Performance-optimal filter type. Figure 10 summarizes the results from the four hardware platforms. For each $\langle n, t_w\rangle$ point, we report whether the performance-optimal filter is a Bloom filter (green area) or a Cuckoo filter (blue area). On all platforms, the Bloom filter is the filter of choice for high-throughput scenarios and Cuckoo for moderate and low-throughput scenarios. On AVX-512 platforms (KNL and SKX), the more SIMD-friendly Bloom filter covers a larger space than on AVX2. Please note that the unit of time for t_w are CPU cycles.

On all platforms we observe a similar shape of the skylines. The left-hand side is dominated by Bloom filters due to their lower lookup costs, whereas the right-hand side, Cuckoo filters dominate due to their lower false-positive-rate. But we also observe that the t_w-range in which the Bloom filters dominate increases with the problem size. For instance, for large problem sizes, Bloom filters perform better than Cuckoo filters for t_ws up to approximately 10^5 cycles. Whereas for small n values, the Bloom filter only performs best up to a t_w of $\sim 10^3$ cycles. This is caused by the higher cache miss probability of the Cuckoo filter which significantly increases the lookup costs once the filter spills to L3 or DRAM. Figure 14 shows a comparison of the lookup costs for three different filter instances. The fact that Cuckoo filters access two cache lines almost doubles their lookup costs compared to Bloom filters. Thus, in terms of filter overhead ρ, it takes "longer" for the Cuckoo filter to compensate the higher lookup costs with its lower false-positive rate.

Performance comparison. In Figure 11a, we compare the performance of Bloom and Cuckoo filters on our default evaluation platform SKX. For each $\langle n, t_w\rangle$ point, we show the performance improvement of the best performing filter, either a Bloom or a Cuckoo filter, over its counterpart. Depending on n and t_w, we observe relative speedups of up to $3\times$ for Bloom filters in high-throughput scenarios ①. The Cuckoo filter, on the other hand, outperforms Bloom filters in low-throughput scenarios by factors ②. Naturally, for arbitrarily large t_ws, the speedup of Cuckoo filters becomes arbitrarily large, as the lower false-positive rate outweighs the higher lookup costs. However, in practical scenarios ($t_w \leq 10^9$ cycles), we observe speedups of up to $5\times$.

False-positive rate. The lowest possible false-positive rate f in our experimental setup is 0.0002 for Bloom (using $k = 11$, and $B = S = 512$) and 0.00005 for Cuckoo (using $l = 16$ and $b = 2$). Note that f for Cuckoo could theoretically be even lower. For instance, with b set to 1, the false-positive probability would be 0.000024. However, construction would most likely fail, as the load factor of the cuckoo hash table would be significantly higher than 50%. Further, if an implementation that supports 19-bit signatures were be available, f could be lowered to 0.000015. Nevertheless, the considered Cuckoo implementations have false-positives rates that are up to an order of magnitude lower than the Bloom filter implementations. Figure 11b shows the dominance of Cuckoo filters in terms of low f (green area). For faster moving workloads (left side), the top performers are Bloom filters with f in $[0.0001, 0.01)$. Higher fs are mostly observed in the area where filtering is not beneficial (top left corner).

Best performing Bloom filter variants. In Figure 12a, we only consider Bloom filters and show which variant performs best. We differentiate between register-blocked, sectorized, cache-sectorized and blocked, whereas the latter refers to blocked Bloom filters without sectorization. The results prove that, due to their low lookup costs, our newly developed Bloom filter variants, register-blocking and cache-sectorization, are well suited for a wide range of problem sizes in high-throughput scenarios. We observed an up to 48% reduced overhead with cache-sectorization as compared to plain sectorization (15% on average). In very few scenarios, plain sectorization performs slightly better (yellow outliers). We attribute this to the dependent load in cache-sectorization (see Section 3.2). However, the increase in overhead was at most 0.5% throughout our experiments.

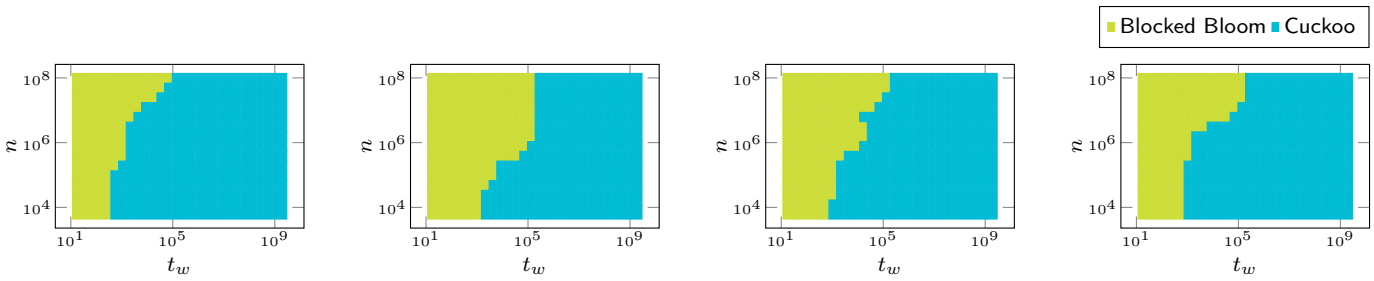
In low-throughput scenarios, higher precision is more important than CPU efficiency and refraining from using sectorization lowers the false-positive rate. However, if we take the space dominated by Cuckoo into account, only a small window of opportunity remains for (standard) blocked Bloom filters ③. Please see Figure 1 for example use cases.

So far, we have only distinguished between the filter types and the different Bloom filter variants. In the following, we examine the parameterization of the individual filters.

Bloom filter configurations. In Figures 12b-12g we resolve the parameters of the performance-optimal Bloom filter configurations. We start with the block sizes. Larger block sizes generally trade CPU efficiency for improved accuracy. However, our cache-sectorization approach allows for efficiently spreading the bits across an entire cache line, with just a minor impact on the lookup costs. Thus, block sizes larger than 4 bytes (single word) and smaller than 64 bytes (cache line) play a minor role ④. Nevertheless, register-blocked Bloom filters with a block size of 4 bytes still outperform cache-sectorization and are therefore the best choice for very low t_ws.

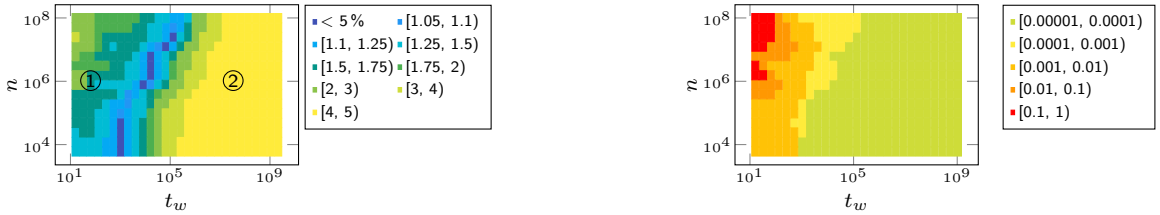
Figure 12c shows the number of sectors used in the performance-optimal Bloom filters. In almost all high-throughput cases, the number of sectors is equal to the number of words per block. A rare exception is ⑤, where the sector size is smaller than the word size. In our implementation, the smallest possible sector size is one byte. Which means, that even a register-blocked filter can be sectorized. This sectorization on the sub-word level has no impact on the lookup performance, but it negatively affects the false-positive rate. However, for very low ks, there is almost no difference in f, and the outlier can therefore be considered noise. In that particular case, the second-best filter instance, which is not sectorized, has only a 0.2% higher overhead. We therefore conclude that sub dividing words into multiple sectors is not beneficial in practice.

On the right-hand side of the skyline, the low-throughput cases, the sector count drops to one (standard blocked Bloom filter), as non-sectorized filters offer a lower f.



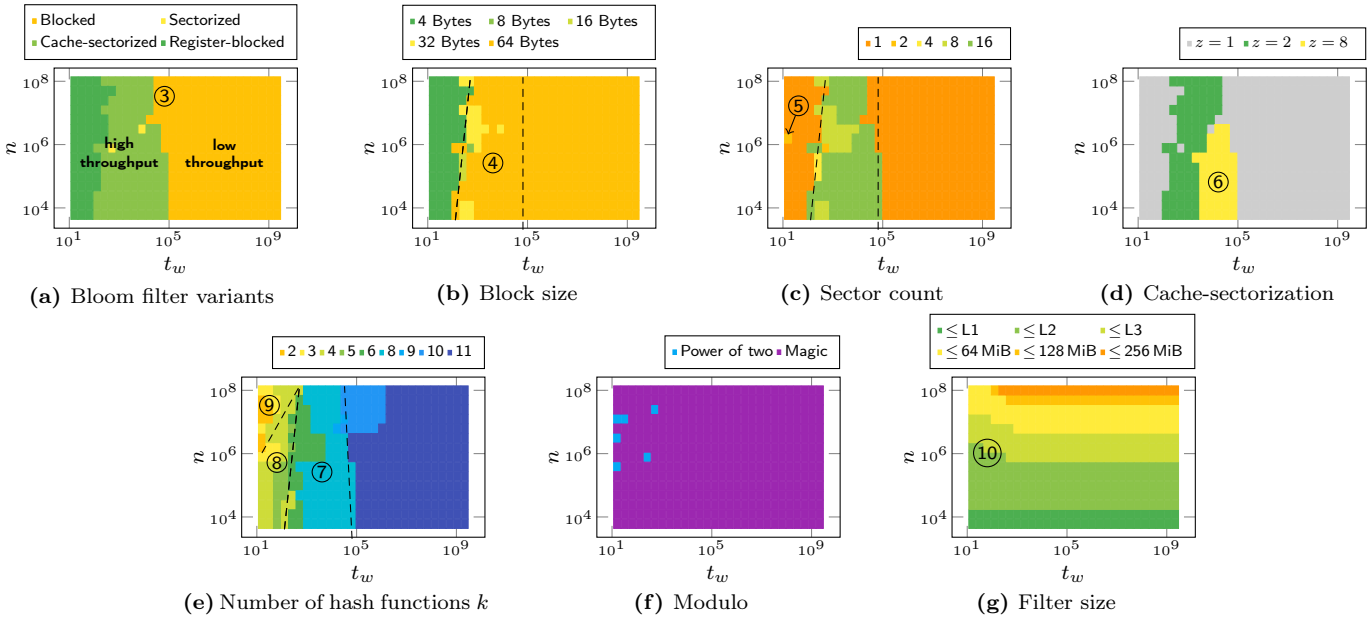
(a) Xeon (14 threads) **(b)** KNL (128 threads) **(c)** SKX (10 threads) **(d)** Ryzen (16 threads)

Figure 10: Skyline of performance-optimal filters for varying n and t_w.



(a) Speedups of the best filter over its counterpart. **(b)** False-positive rates of the best performing filters.

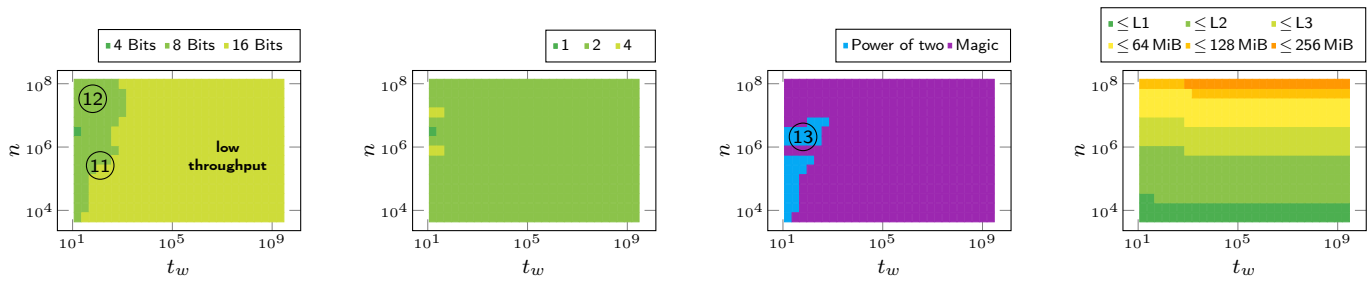
Figure 11: Performance comparison of Bloom and Cuckoo filters (a) and the corresponding false-positive rates (b).



(a) Bloom filter variants **(b)** Block size **(c)** Sector count **(d)** Cache-sectorization

(e) Number of hash functions k **(f)** Modulo **(g)** Filter size

Figure 12: Skyline of configurations of the best performing blocked Bloom filters (on SKX).



(a) Signature length **(b)** Bucket size **(c)** Modulo **(d)** Filter size

Figure 13: Skyline of configurations of the best performing Cuckoo filters (on SKX).

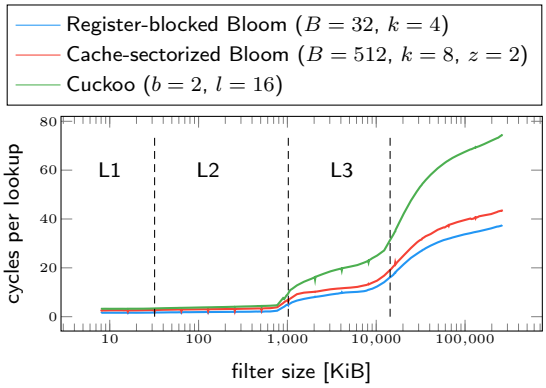
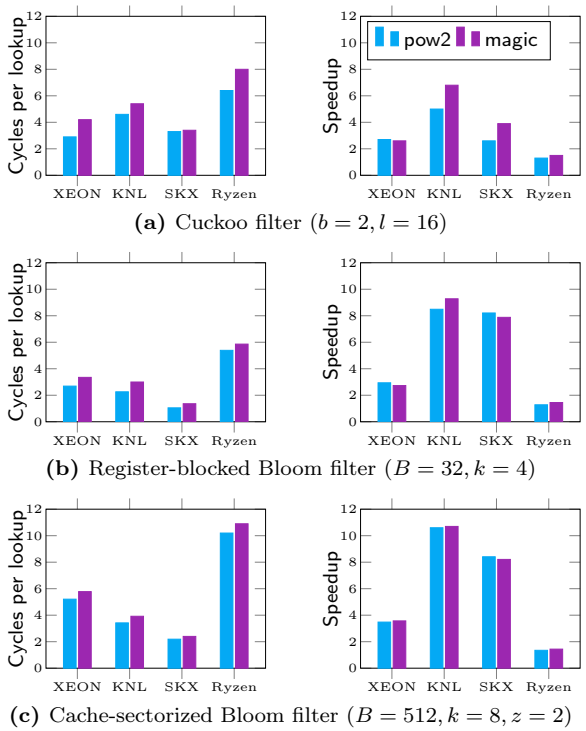


Figure 14: Lookup time for varying filter sizes (on SKX).



(a) Cuckoo filter ($b = 2, l = 16$)

(b) Register-blocked Bloom filter ($B = 32, k = 4$)

(c) Cache-sectorized Bloom filter ($B = 512, k = 8, z = 2$)

Figure 15: Performance of our SIMD-optimized filter implementations.

As mentioned earlier, cache-sectorization covers the largest space in high-throughput scenarios (see Figure 12d). However, the space where $z = 8$ ⑥ is dominated by the Cuckoo filter. Therefore, the most interesting configuration is where two words of a cache line are accessed ($z = 2$).

With regard to the number of hash functions (k), which are shown in Figure 12e, we found that in high-throughput scenarios, a $k \leq 8$ is sufficient. In particular $k = 6$ and $k = 8$ are the sweet spots for cache-sectorized filters ⑦. For register blocking, ks between 3 and 5 offer the best performance ⑧. Filters with a k less than 3 are not practical altogether, as they fall into the area where filtering is not beneficial ⑨. For low-throughput scenarios, we found that ks larger than 11 are never performance optimal.

Figure 12f shows that almost all top-performing Bloom filters make use of magic modulo to optimally utilize the available memory budget (20 bits per key). Magic modulo also helps in cases, where it is better to reduce the k and increase f instead of going to L3 or DRAM ⑩, by adjusting the filter size in small steps.

Cuckoo filter configurations. In Figure 13, we shed light on the parameters of the best performing Cuckoo filters.

Throughout our experiments, the Cuckoo filter tends to use the largest possible signature length l for the given memory budget. – Note that the largest signature length is 16 bit in this case. – Only in very high-throughput scenarios do smaller signatures become beneficial, due to a higher degree of SIMD parallelism. However, in that area, either Bloom filter dominates ⑪ or filtering is not practical altogether ⑫.

An interesting insight regarding the Cuckoo filter is that a bucket size of $b = 2$ is to be favored over $b = 4$, which was chosen for the evaluation in [15] (and hard-coded in their implementation). Choosing a bucket size of 4 was the key feature for Cuckoo filters to achieve better space efficiency than Bloom filters. The fact, that our experimental results show that two signatures per bucket perform better in almost all cases (see Figure 13b) substantiates our general finding, that optimal filter space efficiency does not equate to optimal performance.

Similar to Bloom filters, magic modulo is used to exploit memory constraints as well as possible. However, in comparison to Bloom filters, power-of-two modulo covers a larger space (see Figure 13c ⑬), which is due to the higher costs involved with magic modulo, as described in Section 5.2.

In general, we observed similar memory consumptions among the two filters under test. The claim that Cuckoo filters have better space efficiency [15] no longer holds when performance optimality is the objective.

6.1 SIMD Optimizations

We present the performance impact of our SIMD optimizations. Figure 15 shows the query performance and the speedup over the scalar (non-SIMD) implementation of three representative filters: a Cuckoo filter, a register-blocked, and a cache-sectorized Bloom filter (L1 cache-resident filters, 1 thread). The blue bars represent the filter instances using sizes of powers of two and the purple bars represent the filter instances using magic modulo.

SIMD optimizations offer speedups of up to $10\times$ and therefore make filtering applicable for a larger spectrum in high-throughput scenarios (small t_ws). On AVX2 platforms, the (bare L1) performance of Cuckoo filters is very similar to register-blocked Bloom filters. If the filter size exceeds L2, blocked Bloom filters perform better with regard to CPU cycles per lookup due to better memory bandwidth efficiency (see Figure 14). On AVX-512 platforms, Bloom performs significantly better than Cuckoo. In particular on the Knights Landing (KNL) platform, the Cuckoo filter suffers from mixing AVX2 and AVX-512 instructions due to the missing AVX-512**BW** (**B**yte **W**ord) instruction set. In contrast to the Intel platforms, we barely observed any significant speedups on the AMD Ryzen platform (mostly less than 50% improvement over scalar), which we attribute to the poorly performing `gather` instruction. Compared to SKX, Ryzen is $\approx 2\times$ to $5\times$ slower in absolute numbers (wall clock time, per thread).

7. RELATED WORK

The survey [9] describes many of the application areas of Bloom filters [5]: databases, dictionaries, (P2P) networking and routing. In databases, log-structured merge-trees (LSM) have become important in write-optimized (cloud or cluster) storage, splitting up a structure into multiple layers that are generated sequentially and periodically merged. Queries need to check all layers, and in that respect Bloom filters help to avoid accessing layers that do not contain a key. Monkey [12] observes that different layers need differently tuned Bloom filters. That paper navigates correlated parameter spaces in a data structure and identifies an optimal tuning method. Our insights can be useful for LSMs: we find that Cuckoo filters are a better match than Bloom filters for workloads where filtering avoids I/O.

There have been many extensions of the original Bloom filter [5]. Scalable Bloom filters [3] allow the filter to grow dynamically if n is not known in advance, at the cost of more expensive membership tests (lookups into multiple structures). Spectral Bloom filters [11] and counting Bloom filters [28, 7] can represent bags (duplicate keys) rather than sets. The Bloomier filter [10] can associate a value (rather than a bit) with a key. Retouched Bloom filters [13] allow the suppression of certain selected false positives (that are particularly harmful for the performance of an application).

Our adapted cache-sectorized and register-blocked Bloom filters owe in spirit much to the work by Putze et al. [31] in its search for more CPU-efficient and cache-efficient filters. That study introduced multi-blocked Bloom filters and described SIMD implementations for insert and test, and showed that reducing k and increasing m with regard to their information-theoretic optima can significantly improve performance. Our research into performance-optimal filtering delves deeply into that realm of possibilities. Their SIMD approach is different, as it spreads the bits of a single key throughout the full SIMD register, and the lookup instruction sequence tests just for one key. Rather than setting k bits one-by-one, these bits are generated using pseudo-random, pre-generated bit patterns stored in a table. How these bits are generated is not described, and the Putze et al. source code was not available on request, so a performance comparison was not possible. Our method looks up multiple keys in parallel, one key per SIMD lane, profiting from ever-wider SIMD widths in hardware. For instance, cache-sectorized lookup uses `GATHER-AND-CMP` computation sequences that resolve 16 keys at once using AVX-512.

A SIMD implementation of classic Bloom filters is described in [29]: at every iteration, one bit for multiple tuples is tested (one key per `GATHER` lane). Keys that have been resolved are retired and the SIMD lanes they leave empty get refilled with new tuple data. This approach still suffers from the original Bloom problem that a negative query needs k cache line accesses. In addition, the refill mechanism requires significant CPU work.

The Cuckoo filter [15] achieves better precision than Bloom filters, can represent bags, and allows deletions. However, the CPU and memory cost of Cuckoo filters make membership tests slower. Our work puts Bloom and Cuckoo filters in perspective, and our open-source software release provides highly efficient SIMD implementations for Cuckoo filters, making them more performance competitive. Another optimized Cuckoo filter named the Morton filter is presented in [8]. It reduces the number of accessed cache lines from two down to one in most cases. This is achieved by introducing a new SIMD-friendly data layout, an overflow logic, and compression. We compared our implementation with the reported numbers[5] on similar hardware (Ryzen Threadripper 1950X), showing that our implementation provides the same query performance with large filters ($\approx$ 200MB); we expect it to outperform Morton filters significantly with smaller (cache-resident) filters, which is not the sweet spot of Morton filters.

A few alternative and approximate non-Cuckoo hash tables have been proposed. Both the Quotient filter [4] and TinySet [14] store signatures in a mini-chained hash table. Their advantage over Cuckoo filters is a single cache-miss, as the entire chain fits in a cache line. Their disadvantage is a more CPU-intensive and SIMD-unfriendly lookup, since a loop is needed to walk the chain and determine membership.

Space-efficient index structures, in general, have attracted a lot of interest in database research. Many lightweight data structures have been proposed to accelerate table scans by (i) skipping blocks of tuples, e.g., Column Imprints [32] or MinMax indexes using Small Materialized Aggregates (SMAs) [26], (ii) skipping scan ranges within blocks, e.g., Positional SMAs [22] and Adaptive Range Filters [2], or (iii) by skipping (parts of) individual tuples, e.g., BitWeaving [24, 30] and ByteSlice [16]. The more recent Column Sketches [17] are more heavy weight, as they store approximations of columns using lossy compression, but are also applicable to a wide range of workloads (see Table 1 in [17]). However, it is an open question, whether Bloom filter pushdowns can be combined with Column Sketches (or with any of the aforementioned index structures). A Bloom filter could be populated with the compressed values from the sketch column, but this would require the Column Sketches to have a low false-positive rate and the compressed values need to be known at build time.

8. CONCLUSION

While the space-precision trade-offs of Bloom filters are clearly understood, choosing a performance-optimal configuration is less obvious – in fact it was already known that space-optimal Bloom filters are typically not the most effective configurations. The emergence of new filter types, and specifically the Cuckoo filter, created yet another question for practitioners with regard to what filter type and configuration to use for their problems. Our work sheds light on the issue of which filter structure to choose, and with which parameters, by formally defining performance-optimal filtering and measuring it in our exhaustive experimentation. Our overall finding is that the amount of work saved (t_w) primarily determines the choice between Bloom and Cuckoo: high-throughput workloads (small t_w) should use a (cache-sectorized) Bloom filter, whereas slower moving workloads (high t_w), where precision is absolutely essential, should use a (SIMD) Cuckoo filter.

9. ACKNOWLEDGEMENTS

This work was partially supported by the German Federal Ministry of Education and Research (BMBF) grant `01IS12057` (FASTDATA and MIRIN), and the DFG projects `NE1677/1-2` and `KE401/22`. We would like to thank Abe Wits for his suggestions regarding this research.

[5]At the time of writing, the source code of Morton filters was not available for reproducibility.

10. REFERENCES

[1] Advanced Micro Devices, Inc. *Software Optimization Guide for AMD Family 17h Processors (rev. 3.00)*. 2017.

[2] K. Alexiou, D. Kossmann, and P. Larson. Adaptive range filters for cold data: Avoiding trips to siberia. *PVLDB*, 6(14):1714–1725, 2013.

[3] P. S. Almeida, C. Baquero, N. M. Preguiça, and D. Hutchison. Scalable bloom filters. *Inf. Process. Lett.*, 101(6):255–261, 2007.

[4] M. A. Bender, M. Farach-Colton, R. Johnson, R. Kraner, B. C. Kuszmaul, D. Medjedovic, P. Montes, P. Shetty, R. P. Spillane, and E. Zadok. Don't thrash: How to cache your hash on flash. *PVLDB*, 5(11):1627–1637, 2012.

[5] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[6] P. A. Boncz, T. Neumann, and O. Erling. TPC-H analyzed: Hidden messages and lessons learned from an influential benchmark. In *Performance Characterization and Benchmarking - 5th TPC Technology Conference, TPCTC 2013, Trento, Italy, August 26, 2013, Revised Selected Papers*, pages 61–76, 2013.

[7] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese. An improved construction for counting bloom filters. In *Algorithms - ESA 2006, 14th Annual European Symposium, Zurich, Switzerland, September 11-13, 2006, Proceedings*, pages 684–695, 2006.

[8] A. Breslow and N. Jayasena. Morton filters: Faster, space-efficient cuckoo filters via biasing, compression, and decoupled logical sparsity. *PVLDB*, 11(9):1041–1055, 2018.

[9] A. Z. Broder and M. Mitzenmacher. Network applications of bloom filters: A survey. *Internet Mathematics*, 1(4):485–509, 2003.

[10] B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal. 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 2004, New Orleans, Louisiana, USA, January 11-14, 2004*, pages 30–39, 2004.

[11] S. Cohen and Y. Matias. Spectral bloom filters. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003*, pages 241–252, 2003.

[12] N. Dayan, M. Athanassoulis, and S. Idreos. Monkey: Optimal navigable key-value store. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 79–94, 2017.

[13] B. Donnet, B. Baynat, and T. Friedman. Retouched bloom filters: allowing networked applications to trade off selected false positives against false negatives. In *Proceedings of the 2006 ACM Conference on Emerging Network Experiment and Technology, CoNEXT 2006, Lisboa, Portugal, December 4-7, 2006*, page 13, 2006.

[14] G. Einziger and R. Friedman. TinySet - An access efficient self adjusting bloom filter construction. *IEEE/ACM Trans. Netw.*, 25(4):2295–2307, 2017.

[15] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher. Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, CoNEXT '14, pages 75–88, New York, NY, USA, 2014. ACM.

[16] Z. Feng, E. Lo, B. Kao, and W. Xu. ByteSlice: Pushing the envelop of main memory data processing with a new storage layout. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 31–46, 2015.

[17] B. Hentschel, M. S. Kester, and S. Idreos. Column Sketches: A scan accelerator for rapid and robust predicate evaluation. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 857–872, 2018.

[18] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. 2018.

[19] H. S. W. Jr. *Hacker's Delight, Second Edition*. Pearson Education, 2013.

[20] A. Kirsch and M. Mitzenmacher. Less hashing, same performance: Building a better bloom filter. *Random Struct. Algorithms*, 33(2):187–218, 2008.

[21] M. Kornacker, A. Behm, V. Bittorf, T. Bobrovytsky, C. Ching, A. Choi, J. Erickson, M. Grund, D. Hecht, M. Jacobs, I. Joshi, L. Kuff, D. Kumar, A. Leblang, N. Li, I. Pandis, H. Robinson, D. Rorke, S. Rus, J. Russell, D. Tsirogiannis, S. Wanderman-Milne, and M. Yoder. Impala: A modern, open-source SQL engine for hadoop. In *CIDR 2015, Seventh Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*, 2015.

[22] H. Lang, T. Mühlbauer, F. Funke, P. A. Boncz, T. Neumann, and A. Kemper. Data Blocks: Hybrid OLTP and OLAP on compressed storage using both vectorization and compilation. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 311–326, 2016.

[23] V. Leis, A. Gubichev, A. Mirchev, P. A. Boncz, A. Kemper, and T. Neumann. How good are query optimizers, really? *PVLDB*, 9(3):204–215, 2015.

[24] Y. Li and J. M. Patel. BitWeaving: Fast scans for main memory data processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 289–300, 2013.

[25] M. Mitzenmacher and E. Upfal. *Probability and computing - randomized algorithms and probabilistic analysis*. Cambridge University Press, 2005.

[26] G. Moerkotte. Small Materialized Aggregates: A light weight index structure for data warehousing. In *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA*, pages 476–487, 1998.

[27] R. Pagh and F. F. Rodler. Cuckoo hashing. *J. Algorithms*, 51(2):122–144, 2004.

[28] P. Pandey, M. A. Bender, R. Johnson, and R. Patro. A general-purpose counting filter: Making every bit count. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 775–787, 2017.

[29] O. Polychroniou and K. A. Ross. Vectorized bloom filters for advanced SIMD processors. In *Tenth International Workshop on Data Management on New Hardware, DaMoN 2014, Snowbird, UT, USA, June 23, 2014*, pages 6:1–6:6, 2014.

[30] O. Polychroniou and K. A. Ross. Efficient lightweight compression alongside fast scans. In *Proceedings of the 11th International Workshop on Data Management on New Hardware, DaMoN 2015, Melbourne, VIC, Australia, May 31 - June 04, 2015*, pages 9:1–9:6, 2015.

[31] F. Putze, P. Sanders, and J. Singler. Cache-, hash-, and space-efficient bloom filters. *J. Exp. Algorithmics*, 14:4:4.4–4:4.18, Jan. 2010.

[32] L. Sidirourgos and M. L. Kersten. Column imprints: A secondary index structure. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 893–904, 2013.

[33] M. Zukowski, M. van de Wiel, and P. A. Boncz. Vectorwise: A vectorized analytical DBMS. In *IEEE 28th International Conference on Data Engineering (ICDE 2012), Washington, DC, USA (Arlington, Virginia), 1-5 April, 2012*, pages 1349–1350, 2012.