



TECHNISCHE UNIVERSITÄT MÜNCHEN

TUM School of Computation, Information and Technology

Query Processing on Modern Hardware

Harald Lang



TECHNISCHE UNIVERSITÄT MÜNCHEN

TUM School of Computation, Information and Technology

Query Processing on Modern Hardware

Harald Lang

Vollständiger Abdruck der von der TUM School of Computation, Information and Technology der Technischen Universität München zur Erlangung eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitz: Prof. Dr.-Ing. Georg Carle

Prüfer der Dissertation: 1. Prof. Dr. Thomas Neumann
2. Prof. Alfons Kemper, Ph.D.
3. Prof. Dr. Jens Teubner
(Technische Universität Dortmund)

Die Dissertation wurde am 26.09.2022 bei der Technischen Universität München eingereicht und durch die TUM School of Computation, Information and Technology am 27.02.2023 angenommen.

*To my daughter Lisa, my son David, and my wife Maria.
I love you.*



List of Publications

This cumulative dissertation is based on the following peer-reviewed publications:

- [A] Harald Lang, Thomas Neumann, Alfons Kemper, Peter Boncz, *Performance-Optimal Filtering: Bloom Overtakes Cuckoo at High Throughput*, In *Proceedings of the VLDB Endowment*, 12(5):502–515, Jan. 2019. <https://doi.org/10.14778/3303753.3303757>

This work is licensed under the Creative Commons AttributionNon-Commercial-NoDerivatives 4.0 International License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-nd/4.0/>. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.

Permission to republish the work as part of this thesis was granted by Prof. Dr. Volker Markl (on behalf of the VLDB Endowment) on Dec. 19, 2019.

- [B] Harald Lang, Linnea Passing, Andreas Kipf, Peter Boncz, Thomas Neumann, Alfons Kemper, *Make the Most out of Your SIMD Investments: Counter Control Flow Divergence in Compiled Query Pipelines*, In *The VLDB Journal*, 2019. <https://doi.org/10.1007/s00778-019-00547-y>

This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

- [C] Harald Lang, Alexander Beischl, Viktor Leis, Peter Boncz, Thomas Neumann, Alfons Kemper, *Tree-Encoded Bitmaps*, In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2020. <https://doi.org/10.1145/3318464.3380588>

Reused in accordance with the ACM publication policies (<https://authors.acm.org/author-services/author-rights>):

“Authors can include partial or complete papers of their own (and no fee is expected) in a dissertation as long as citations and DOI pointers to the Versions of Record in the ACM Digital Library are included.”

Abstract

This thesis investigates how query processing in relational database systems could be further improved. In particular, we focus on three key aspects of query processing. First we investigate which approximate filtering data structures are best suited and additionally we present two novel variants of Bloom filters that are optimized for modern hardware and high throughput. We also present algorithms that help to optimally utilize the available compute resources of modern SIMD processors and thereby reduce the runtime of database queries. Last, we investigate compression techniques for bitmaps that are primarily used in secondary indexes and we present a novel bitmap compression format based on binary trees that can, with regard to memory consumption and access latency, compete with the best-known compression techniques in their respective category.

Zusammenfassung

Die vorliegende Arbeit untersucht inwiefern die Anfragebearbeitung in relationalen Datenbanksystemen effizienter gestaltet werden kann. Insbesondere werden die Schwerpunkte auf drei Kernaspekte der Anfragebearbeitung gelegt. Zunächst untersuchen wir welche Datenstrukturen für approximatives Filtern am besten geeignet sind und präsentieren im gleichen Zuge zwei neuartige Varianten des Bloom Filters welche für moderne Hardware und hohe Durchsätze optimiert sind. Des Weiteren präsentieren wir Algorithmen welche die verfügbaren Rechenressourcen in modernen SIMD Prozessoren optimal auslasten um somit die Verarbeitungsdauer von Datenbankanfragen zu reduzieren. Abschließend untersuchen wir Kompressionsverfahren für Bitmaps welche hauptsächlich in Sekundärindexen zum Einsatz kommen und präsentieren ein neuartiges Kompressionsverfahren, basierend auf Binärbäumen, welches hinsichtlich des Speicherbedarfs so wie hinsichtlich der Zugriffslatenzen mit den besten bekannten Verfahren in der jeweiligen Kategorie konkurrieren kann.

Contents

List of Publications	iii
Abstract	v
1 Introduction	3
2 Approximate Membership Query Data Structures at High Throughput	5
2.1 Applications	6
2.2 Data Structures	9
2.3 Performance Optimality	17
2.4 Conclusions	20
3 Efficient Control Flow Handling in Compiled Queries	21
3.1 Data Centric Query Compilation	21
3.2 Data-Parallel Pipelines	22
3.3 Control-Flow Divergence	23
3.4 Countering Underutilization	24
3.5 Conclusions	25
4 Space- and Time-Efficient Bitmap Indexing	27
4.1 Bitmap Index Design Space	28
4.2 Bitmap Compression	31
4.3 Tree-Encoded Bitmaps	40
4.4 Future Work	43
Publications	47
A Performance-Optimal Filtering: Bloom Overtakes Cuckoo at High Throughput	47
A.1 Introduction	50
A.2 Performance-Optimal Filtering	53
A.3 Bloom Filter Variants	55
A.4 Cuckoo Filter	61
A.5 Implementation Techniques	64
A.6 Experimental Analysis	67

A.7	Related Work	75
A.8	Conclusion	78
A.9	Acknowledgements	79
B	Make the Most out of Your SIMD Investments: Counter Control Flow Divergence in Compiled Query Pipelines	81
B.1	Introduction	84
B.2	Background	85
B.3	Vectorized Pipelines	87
B.4	Refill Algorithms	87
B.5	Refill Strategies	96
B.6	Evaluation	100
B.7	Summary and Discussion	117
B.8	Conclusions	118
B.9	Acknowledgements	119
C	Tree-Encoded Bitmaps	121
C.1	Introduction	124
C.2	Tree-Encoded Bitmaps	127
C.3	Operations	134
C.4	Experimental Analysis	141
C.5	Related Work	149
C.6	Conclusion	150
	Bibliography	153
	Appendix	173

1 Introduction

Database system architectures that keep all data in main memory have become a huge success. These systems are orders of magnitude faster than disk-based architectures and allow for *interactive analytics* over huge data sets, which is crucial to many business and scientific applications. The performance of these main-memory database systems is no longer bound by disk I/O, since DRAM offers a significantly higher bandwidth and faster random access. Depending on the complexity of a query, the CPU can quickly become the performance limiting factor [25]. This observation led researchers to revisit existing database operators and algorithms. Many of the proposed algorithms are *hardware conscious* and consider various properties of the underlying hardware such as the memory hierarchy [152, 24], multi-core and system topology [22, 100, 19, 18, 95, 11, 88, 114, 166, 150], as well as specialized CPU instruction sets [148, 141, 142, 140]. In the recent iterations of hardware evolution, the single instruction multiple data (SIMD) capabilities of modern CPUs have made significant advances. In particular, the number of registers increased to 32 and the register size increased to up to 512 bits with the AVX-512 instruction set. Thus, more data can be kept in registers and more data can be processed in data-parallel per issued instruction. Further, we observe that newer CPUs have richer and more complex instructions, which extends the applicability of SIMD to an even larger spectrum of algorithms, thereby offering new potentials to accelerate main memory database systems to analyze big data in near real-time.

In this work we investigate how query execution performance can be further improved. We thereby revisit various aspects of query processing, which includes (i) approximate key filtering, (ii) data-centric query compilation and (iii) secondary index structures.

Chapter 2 focuses on Approximate Membership Query (AMQ) data structures, which are widely used in data management systems to reduce query response times. The various existing AMQ data structures have different properties regarding their lookup performance and their space/precision trade-offs and therefore some data structures suit a specific workload or application better than others. We investigate which type of data structure should be used in which situation and how it should be parameterized to achieve optimal performance. Further, we design two novel variants of Bloom filters that are optimized for high-throughput scenarios that occur in high-performance main-memory database systems.

In Chapter 3, we address a problem that occurs when query execution plans are compiled in a data-centric way. Existing query compilers [85, 93] that fol-

low the produce/consume compilation model, produce tight loops that process a tuple at a time. When this model is extended to process small batches of tuples to utilize SIMD capabilities of modern CPUs, the involved branching logic needs to be adjusted as well. In particular, that means that any branch needs to be taken if *at least one* tuple of the current batch satisfies the corresponding branch condition. In this execution model, we cannot expect that the exact same branches are taken for all tuples of a batch. Thus, in general, the control flow will diverge on a per tuple basis, which consequently leads to underutilization of precious compute resources, because some tuples (or elements thereof) are temporarily set inactive, while still remaining in CPU registers.

In the last chapter, Chapter 4, we revisit compression algorithms for bitmaps (or bit-vectors), a topic that has been around in database research for more than two decades but which has gained some momentum recently. Bitmap compression is an essential part of bitmap indexes, which in turn are highly effective in analytical database management systems, especially in evaluating high-dimensional selection queries. Existing bitmap compression techniques, however, either offer good compression ratios or good performance, but typically not both. Further, to the best of our knowledge, only a single bitmap compression format exists so far, that turned away from the commonly used run-length encoding in favor of allowing for efficient (logarithmic) random access, which in turn significantly improves bitmap operations like computing a bitwise AND. Motivated by the observation that better performance comes at the cost of an up to $3\times$ higher memory consumption, we designed a novel bitmap compression format that is efficient in both dimensions, space and time.

2 Approximate Membership Query Data Structures at High Throughput

Approximate Membership Query (AMQ) data structures allow for space-efficiently representing sets. These data structures, like the well-known Bloom filter [23], store approximations of sets to save space. Most AMQ data structures are designed to have a one-sided error, meaning that with a (small) probability, the data structure erroneously reports that the queried element is part of the set, which is known as a *false positive* result. On the other hand, it is guaranteed that it never incorrectly reports an element not being part of the set, which is known as a *false negative* result. Therefore, the two possible outcomes of a membership query could either be (i) the element is likely part of the set or (ii) the element is definitely not part of the set. The first is called a positive query and the latter a negative query.

The space requirements of an AMQ data structure depend on the error probability, which is often referred to as *false positive probability* or *false positive rate*, as we don't expect false negatives to occur. Typically, the false positive rate can be adjusted to the application's need. A lower false positive rate thereby incurs a higher space consumption, and vice versa.

In database systems, AMQ data structures have found many applications. Such systems use AMQ data structures to reduce costly disk I/O [72, 38, 132, 151, 53, 54, 90], to speed up (distributed) semi-join operations [29, 167, 112, 120], estimate the size of a semi-join [121], accelerate hash table lookups during join processing [158, 100], and more recently AMQ data structures have been employed in approximate indexing techniques [15, 90]. AMQ data structures have also found a lot of applications in networking [32, 165], e.g., in routing [50, 65, 73, 185, 83, 108, 107], network management and monitoring [158, 27, 186], caching proxies [149, 64], as well as in browser/PKI security [97, 111] and anti-virus software [62].

Over the last decades, several extensions and variants of the classic Bloom filter have been proposed and alternative data structures have been developed to address the various different needs of applications. For instance, the classic Bloom filter does not allow for element deletion without reconstructing the entire filter; an issue that has gained a lot of interest in research. Other research directions have been motivated by (i) further reducing the space consumption and/or (ii) reducing the query latencies.

The AMQ filter structures have been well-studied with regard to their space-

precision trade-offs, which means, that a space-efficient filter configuration can be determined to achieve a *desired* false-positive rate. Vice versa, a filter configuration can be determined that offers the highest possible precision (lowest false positive rate) for a given memory constraint. Minimizing the memory consumption of a filter structure for a given desired false positive rate or vice versa is however not a goal in itself, rather, the goal is to optimize the overall performance of a particular workload. In fact, the *desired* false positive rate is often not known in practice, which leads to systems that use a hardcoded false positive rate or even to filter implementations where most parameters are fixed [93]. This modus operandi consequently leads to sub-optimal performance improvements or, in worst-case scenarios, to degradations in end-to-end performance. In particular with fast-paced workloads, where the filter structure is queried thousands or millions of times per second, an improperly chosen filter configuration could have significant impact on overall system performance. System implementers who want to employ AMQ data structures are interested in achieving the best possible system performance, but they are facing questions like which data structure to use (if any) and how it should be configured. And in fact, there are no general answers to these questions, since it is unclear how individual system and workload characteristics map to the filter structure’s configuration parameters, such as the aforementioned desired false-positive rate.

With this work, we provide the necessary tools and formalism to achieve *performance-optimality*. In particular, we study the performance related aspects of filter structures, investigate the raw throughput of filters in terms of queries per seconds, and we model the context in which the filters are installed, which allows for determining the *performance-optimal* filter for a particular workload. Further, we propose novel Bloom filter variants that are optimized for high query rates by leveraging modern hardware capabilities, such as the data-parallelism of Single Instruction Multiple Data (SIMD) instructions.

The rest of this chapter is structured as follows. In Section 2.1, we discuss two example applications for AMQ data structures. The purpose is to illustrate the different requirements on the filter structure in use. In Section 2.2, we give an overview of existing AMQ data structures and in Section 2.3, we present our work on performance-optimal filtering, including our novel Bloom filter variants and an experimental analysis. We draw our conclusions in Section 2.4.

2.1 Applications

In the following, we briefly discuss two use case scenarios for Bloom filters (or AMQ data structures in general) in the context of data management systems. We discuss how Bloom filters are used to improve lookup performance in log-structured merge-trees (LSM trees) [132] and to accelerate relational join processing. These are probably the most common applications in data management systems, and furthermore, they denote two extreme cases regarding

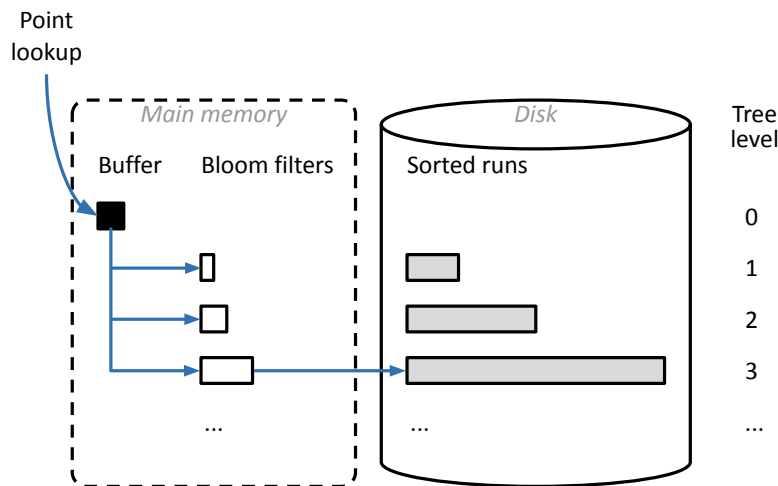


Figure 2.1: LSM trees maintain Bloom filters in main memory to avoid unnecessary disk I/O. ¹

query rates and their expected accuracy.

LSM trees are write-optimized data structures that store key-value pairs in a hierarchy of sorted runs. These sorted runs grow exponentially in size with their depth within the tree. Newly inserted and updated entries are buffered in memory (level 0) and are flushed to persistent storage when the buffer becomes full. When this occurs, the buffered entries are merged with the sorted run of the next level. The merge process cascades down the tree, possibly introducing a new level when all existing levels are at maximum capacity. A key lookup in a LSM tree starts at level 0 and continues at the next deeper level until either the requested key has been found, or all sorted runs have been probed without success. To speed up lookups, each persistent run has a corresponding Bloom filter in main memory that is consulted before the run is accessed. When the outcome of the Bloom filter query is negative, the corresponding run can safely be skipped. Thus, the Bloom filter helps to prevent unnecessary and costly I/O operations, cf. Figure 2.1.

When most data is stored on secondary storage, the lookup performance of an LSM tree is bound by I/O. Lookups can be served at a rate of thousands per second with (rotating) hard disk drives or at a rate of tens of thousands with NVMe SSDs. Thereby, it is important that the Bloom filters are optimally parameterized [53, 54], which could improve the lookup throughput by an order of magnitude.

Accelerating selective (semi-) join operations in relational database systems is another popular use case for Bloom filters. In foreign-key joins between a large fact table and a smaller dimension table with a filter predicate that selects a fraction of the dimension tuples, only a fraction of the tuples from the fact table will find a match and contribute to the join result. In this case, it

¹Illustration is inspired by Figure 2 in [53].

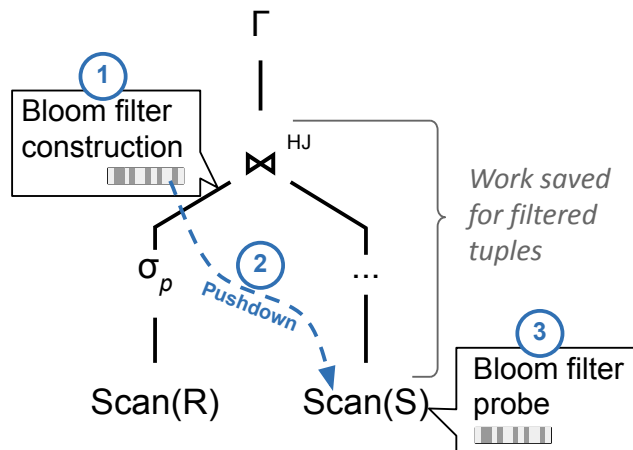


Figure 2.2: Bloom filters with selective joins. Tuples without a join partner are filtered before entering the pipeline.

can be beneficial to create a Bloom filter that contains the selected dimension keys, and for each fact tuple first query the filter. If the filter returns a negative result, the tuple does not join and it is (thereafter) ignored in terms of further work, such as a hash table or index lookup, or nested-loop scan. This filtering could be the first step in the join [100], but the filter test can also be pushed down all the way into the fact table scan, as illustrated in Figure 2.2, such that the data volume coming out of the scan is reduced, making all intermediate operations in between the scan and the join cheaper.

In contrast to the LSM tree use case, join filtering (in a main-memory database system) is either DRAM bandwidth bound or compute bound, depending on the complexity of the query. Thus, in such an in-memory settings, the Bloom filter is probed a rate of millions per second (even billions per second). Another important difference between those two use cases is that the costs incurred by false-positive results differs significantly. In the case of LSM trees, a false positive causes an unnecessary disk I/O, which could waste several milliseconds. Whereas in join processing, the cost of a false positive is typically several orders of magnitude lower, e.g., an LLC miss, which accounts for approximately 50 nanoseconds. We call this a *high-throughput scenario*. In such scenarios, the employed filter needs to be highly throughput optimized as it may otherwise cause performance degradations. In scenarios with lower throughput on the other hand, the filter structure needs to be tuned for high accuracy (low false-positive rates), since fetching a page from disk, or possibly fetching a data block from cloud storage [51], notably increases the lookup latency.

2.2 Data Structures

Since the Bloom filter was introduced in 1970, several other AMQ data structures have been presented and applied in practice. Also, several extensions or modifications to the classic Bloom filter have been proposed. In the following, we give an overview and briefly discuss their properties. In particular, we focus on the performance-related aspects and survey several optimizations that trade off access speed, accuracy, and space efficiency.

2.2.1 Bloom Filter

A Bloom filter [23] consists of an array of bits (a bitmap) B and a set of hash functions h_0, h_1, \dots, h_{k-1} that map arbitrary elements to integers within the range $[0, m)$, where m denotes the number of bits in B . When an element x is inserted, the element is hashed using the k hash functions and the corresponding bits in B are set to 1: $B[h_i(x)] \leftarrow 1$, with $0 \leq i < k$. To check whether an element is (likely) part of the Bloom filter, the element is hashed and the corresponding bits are tested. If all bits at the positions where the k hash functions map to are 1's, then the element is likely part of the set the Bloom filter represents, otherwise the element is definitely not part of the set. The lookup procedure can be aborted when a 0-bit is observed. In that case, the filter can immediately return a negative result without testing the remaining bits. Hence, only positive lookups require that all k bits be tested.

The size m of the bit array is typically chosen based on the number of elements n that will be inserted and a desired false-positive rate f : $m = 1.44 kn$, whereas the number of hash functions is $k = -\log_2(f)$. The resulting Bloom filter instance is considered as *space-optimal*—sometimes also called space-optimized or simply optimal—Bloom filter. The naming might imply that the Bloom filter's space consumption is optimal in the informational theoretic sense, which, however, is not the case. It actually has a 44% space overhead compared to the (asymptotic) lower bound given in [32], when k is chosen as shown above, which implies that Bloom filters still offer compression potentials [116]. The bitmap of a space-optimal Bloom filter is 50% populated with 1-bits. Thus, negative lookups can be answered with 50% chance after testing the first bit, assuming uniformity of the hash functions. Negative lookups can be speed up when the Bloom filter is sparsely populated; at the expense of higher space consumption, i.e., the corresponding Bloom filter instance would then no longer be space optimal. In general, the false-positive rate can be computed as follows:

$$f = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx \left(1 - e^{-kn/m}\right)^k \quad (2.1)$$

Type	Scale
L1 read	≈ 0.5 cycles
L2 read	≈ 2 cycles
L3 read	≈ 10 cycles
DRAM read	≈ 50 ns (200 cycles @ 4 GHz)
TLB L1 miss penalty	≈ 9 cycles
TLB L2 miss penalty	≈ 17 cycles
Branch misprediction penalty	≈ 17 cycles
AND/OR/XOR instruction latency	$\approx 1 - 5$ cycles
DIV instruction latency	$\approx 23 - 88$ cycles

Table 2.1: Latencies and penalties on a Skylake-X platform [1, 67].

Alternatively, we can compute the filter size for given n , k and f :

$$m = -\frac{1}{\sqrt[kn]{1 - \sqrt[k]{f}} - 1} \approx -kn \log^{-1/k}(1 - f) \quad (2.2)$$

Performance Optimization Techniques

Bloom filters are commonly known for being easy to implement and for being fast, as the time complexity of membership queries is constant. In practice, however, implementations differ quite often from the formal definition of the classic Bloom filter. Most of these modifications are motivated by improving the lookup performance, in terms of reduced lookup latency and/or higher lookup throughput, by avoiding common latencies and penalties caused by the underlying hardware. Naturally, the memory hierarchy plays an important role, but so do branch mispredictions as well as the different latencies of various CPU instructions, cf. Table 2.1. In the following, we briefly survey the techniques used to improve the performance of Bloom filters.

Avoid Modulo Operations. The probably most commonly applied optimization is to avoid costly modulo instructions, and substitute them by bitwise-AND instructions, i.e., rather than testing the bit at position $h(x) \bmod m$, the position $h(x) \& \text{mask}$ is tested, with $\text{mask} = (1 \ll \log_2(m)) - 1$. The first involves an integer division instruction (DIV) that is significantly slower than a bitwise operation, cf. Table 2.1. A consequence of this optimization is that the filter size m is restricted to powers of two and the optimal filter size can no longer be chosen, in general. In the worst case, the actual filter size could therefore be almost a factor of two larger than necessary. In other words, avoiding the modulo operation trades improved computational efficiency for higher space consumption.

Blocking. Blocking, as proposed in [144], is a technique where the Bloom filter’s bit array is split into smaller (cache-line sized) blocks. When an element is inserted, only a single block is affected, i.e., all k bits are set within that block. The target block is determined by an additional hash function h_b that maps the input element to the integer range $[0, m/512)$, where 512 is the size of a cache line (in bits) on x64 architectures; assuming m is an integral multiple of 512. Vice versa, only a single block needs to be consulted during lookups. Consequently, a lookup in a so called *blocked Bloom filter* causes at most one cache miss, irrespective of whether the lookup is positive or negative. Serving lookups with at most a single cache miss is a distinctive feature of blocked Bloom filters, and also the most impactful when the filter size exceeds L2 or L3 cache size. Blocking also improves performance with smaller sized filters. It reduces the computational efforts, as less hash bits need to be computed and the modulo operation is executed only once, to determine the target block. Within a block, the faster bitwise-AND is used, as the block size is a power of two. Nevertheless, blocking negatively affect the filter’s space efficiency, as 1-bits are being clustered rather than being uniformly distributed over the entire bit array.

Sectorization. Sectorization is a technique to further improve the CPU efficiency of blocked Bloom filters. The key idea is to sub-divide blocks into smaller partitions, which we call *sectors*. The number of sectors per block thereby is equal to the number of hash functions k , and when an element is inserted, a single bit is set per sector. The main advantage of this performance optimization is that it turns the random access pattern within a block into a sequential access pattern. In the implementation that is used in the Impala database system [93], a sector corresponds to a 32-bit word, and the block size to $32k$ bits, thus, once the target block has been determined, all sectors (words) are processed sequentially and independently.

Branch Elimination. A Bloom filter implementation may always test all k bits, rather than branching out after the first 0-bit is observed. This technique aims for (i) avoiding branch misprediction penalties and (ii) for exploiting out-of-order execution, as all k bits can be tested independently, i.e., the CPU may issue multiple concurrent memory loads to hide memory latencies. Nevertheless, this technique works best with blocked Bloom filters, as it may not cause additional (unnecessary) cache misses.

Exploit Data Parallelism at the Hardware Level. Finally, the simplicity of Bloom filters makes it easy to benefit from SIMD instructions. In principle, SIMD can be used to parallelize filter lookups in two different ways: (i) multiple (batched) lookups are performed in parallel [142] or (ii) the k bits of an element are tested in a data-parallel fashion [144, 93]. The first option might not be applicable in certain situations, whereas the latter might put restrictions on the possible filter configurations, i.e., k should be equal to (or an integral multiple

of) the number of available SIMD lanes, otherwise, the hardware would not be fully utilized.

Variants and Related Work

Besides the performance-related improvements, several other extensions and variants have been proposed. The *scalable Bloom filter* [13], for instance, allow the filter to grow dynamically in size if the number of elements in the set is not known in advance. The scalable Bloom filter thereby internally consists of multiple standard Bloom filters. When the existing filter instance become full, i.e., reaches the desired false-positive rate, a new Bloom filter instance is appended. The newly created instances have a lower false-positive rate than the previous instances, so that the combined false-positive rate does not exceed the pre-defined false-positive rate. Dynamically resizing the filter comes at the cost of more expensive membership tests, as multiple structures need to be consulted. Thus, the more accurately the initial size of the first instance is estimated, the fewer instances are created and the lower the lookup costs are.

Dynamically removing elements from a set is often required in network and streaming applications. Bloom filters have been extended to support deletions by introducing counters [64] on a per-element basis. These counters keep track of insertions and deletions by incrementing or decrementing the corresponding counters. A counting Bloom filter occupies a multiple of the space of a standard Bloom filter; typically each counter requires 4 bits [64], thus a counting Bloom filter is four times larger than a standard Bloom filter.

Spectral Bloom filter [47] generalize Bloom filters to represent multisets (bags), where membership queries return the multiplicity of an element. The returned multiplicity thereby is never smaller than the actual multiplicity of an element, but possible larger within some pre-defined error boundary.

Some Bloom filter variants [60, 56] also allow for false negatives, for instance to improve the false positive rate [60]. For more details, we refer to the comprehensive survey of Tarkoma et al. [165], in particular to Table II, which summarizes the most important properties of 22 Bloom filter variants.

2.2.2 Cuckoo Filter

The cuckoo filter [63] is an AMQ data structure that stores small *signatures* (aka fingerprints) in a cuckoo hash table [136]. A signature of an element is computed using a hash function that maps arbitrary elements to the integer range $[0, 2^l)$, where l denotes the signature length in bits. Similar to cuckoo hashing, each signature has two candidate buckets in the hash table where it can be stored. When both buckets are occupied by other signatures, one of these signatures is picked randomly and is then relocated to its alternative bucket to make room for the newly inserted element. This procedure may continue when the alternative bucket of the evicted signature is occupied as well. To avoid infinite loops during collision resolution, the maximum number

of relocations is limited to a certain threshold. When a collision cannot be resolved within that threshold, the cuckoo filter is considered fully occupied.

A major difference between a cuckoo hash table and a cuckoo filter is that the cuckoo filter has to perform relocations based on the signatures rather than on the original elements. Thus, a cuckoo hash table could use two independent hash functions. For instance, to relocate the element x that is stored in the hash table T in bucket i , the element is (re)hashed to determine the candidate buckets $i_1 = h_1(T[i])$ and $i_2 = h_2(T[i])$, and is then moved either to i_2 if $i = i_1$ or to i_1 if $i = i_2$. This approach is, however, no longer applicable when the hash table contains only signatures. Fan et al. therefore proposed a technique where the alternative bucket index can be determined based on the element's signature and the bucket index the signature is currently stored in. The authors refer to this technique as *partial-key cuckoo hashing*, which works as follows: The indexes of the two candidate buckets i_1 and i_2 of an element x are:

$$\begin{aligned} i_1 &= h_1(x) \\ i_2 &= i_1 \oplus h_2(x\text{'s signature}). \end{aligned} \tag{2.3}$$

There are a few things to note here. First, the index i_1 can also be determined based on i_2 and the element's signature without having the the original element x in hand:

$$i_1 = i_2 \oplus h_2(x\text{'s signature}). \tag{2.4}$$

Further, the hash function h_2 is optional. The purpose of the second hash function is to spread out the candidate buckets over the entire hash table. If the second hash function were omitted, the distance between the buckets would be at most $2^l - 1$, which would lead to clustering and consequently to a higher collision probability.

Since the cuckoo filter's internal data structure is a hash table rather than a bitmap as in Bloom filters, the cuckoo filter could hold only a limited number of elements. When the table is fully occupied, or the collision resolution fails, no more elements can be inserted, naturally. In a cuckoo hash table, it is very unlikely that the table gets fully occupied [136]. In fact, the probability that the collision resolution fails increases with the number of inserted elements. This worsens with the cuckoo filter's partial-key cuckoo hashing, where the alternative bucket index is a combination of the signature and the current bucket index, rather than having two independent hash functions. The authors empirically determined that the maximum table occupancy is only around 50%. To address this issue, the authors propose to increase the bucket size, so that multiple signatures can be stored in a single bucket. This, for instance, allows for a maximum table occupancy of 84%, 95%, or 98%, using a bucket size $b = 2, 4$ or 8 , respectively.

Increasing the number of signatures per bucket also negatively affects the

false-positive rate, which is:

$$f_{\text{cuckoo}} = 1 - \left(1 - \frac{1}{2^l}\right)^{2b\alpha}, \text{ with: } \alpha = \frac{l \cdot n}{m} \quad (2.5)$$

where α refers to the load factor of the table. Nevertheless, setting $b = 4$ makes the cuckoo filter competitive with the Bloom filter in terms of its space/precision trade-offs. I.e., when the cuckoo filter is at maximum load, and the false-positive rate is $f < 0.004$, the cuckoo filter occupies less space than a space-optimal Bloom filter.

To ensure that a filter achieves a desired false-positive rate, the optimal signature size l needs to be determined:

$$l = \left\lceil -\log_2 \left(1 - (1 - f)^{\frac{1}{2b\alpha}}\right) \right\rceil \quad (2.6)$$

Thereby, α is set to maximum capacity for the given bucket size b :

$$\alpha = \begin{cases} 0.5, & b = 1 \\ 0.84, & b = 2 \\ 0.955, & b = 4 \\ 0.98, & b = 8 \end{cases} \quad (2.7)$$

Finally, based on Equation 2.5, the filter size can be computed as follows:

$$m = \frac{l \cdot n}{\alpha} \quad (2.8)$$

Vice versa, when memory is constrained, the optimal signature size is determined as follows:

$$l = \left\lceil \alpha \frac{m}{n} \right\rceil \quad (2.9)$$

Choosing the proper bucket size b is less obvious, as α increases with b , but f decreases with larger b 's. However, as Figure 2.3 shows, a bucket size of 4 is a “good” choice in practice, where the bits per element ratio ($C = m/n$) is typically within the range [8, 20]. $b = 2$ offers only a slightly lower false-positive rate when $8 \leq C < 13$. But for $C \geq 13$, setting the bucket size to 4, improves the accuracy of the filter significantly. The authors Fan et al., therefore use $b = 4$ as the default setting.

As mentioned earlier, the false-positive rate of a filter needs to be less than 0.004 for a cuckoo filter to be more space-efficient than a Bloom filter. This cross-over point can be raised to $f \approx 0.025$ by compressing the individual buckets of the cuckoo filter. The authors adopted the *semi-sorting* technique from [28], which allows for saving 1 bit per stored element (signature).

From the performance perspective, semi-sorting adds significant overhead. For instance, a cuckoo filter with $l = 12$ offers approximately twice the lookup

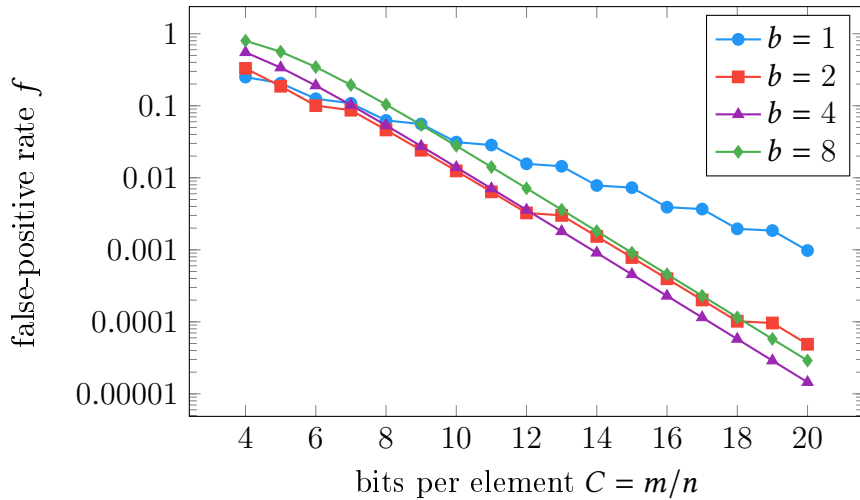


Figure 2.3: False-positive rate of Cuckoo filters for various bucket sizes b .

throughput of a semi-sorted cuckoo filter with $l = 13$, as shown in [71]. Semi-sorting also roughly doubles the filter construction time. Another performance-related issue that cuckoo filter implementations are facing are unaligned memory accesses. For space-efficiency, the buckets are densely packed and depending on the chosen signature size, this circumstance results in unaligned loads. Depending on the signature and the filter size, a lookup can therefore become twice as costly, compared to a lookup that only involves aligned loads. In general, lookup performance can be improved by aligning the buckets (and the signatures) to word boundaries, which does, however, require padding and consequently lowers the overall space-efficiency. In particular, when SIMD optimizations come into play, signatures need to be either 8, 16 or 32 bits in size and aligned accordingly, as current SIMD instruction sets (i.e., AVX2 and AVX-512) do not support unaligned memory gathers. Thus, in summary, the performance highly depends on the chosen signature size, and only for a few combinations of signature and bucket sizes are highly efficient implementations feasible. For instance, a filter that is configured with $l = 8$ and $b = 4$ is highly efficient with AVX-512 SIMD, as buckets are 32-bit aligned and the linear search within the buckets can be performed in a (16-way) data-parallel fashion.

In terms of random memory accesses, the cuckoo filter shows a different behavior than the Bloom filter. A Bloom filter could definitely answer a negative lookup when the first 0-bit is observed. Only in case of positive lookups do all k bits need to be tested. Cuckoo filters on the other hand need to inspect *both* candidate buckets of an element to answer negative lookups. Only positive lookups can be answered with 50% change by inspecting only a single bucket. However, as the event of a negative lookup is more likely than a positive lookup in practice, cuckoo filter implementations typically eliminate conditional branches and always test both candidate buckets to improve per-

formance, as described in the earlier Section 2.2.1. Thus, a lookup causes at most two cache misses, which however, is still twice as high as with blocked Bloom filters.

Variants and Related Work

While the cuckoo filter is a compact variant of a cuckoo hash table, it shares similarities with the (counting) *quotient filter* [20, 137]. Both use open addressing hash tables in which signatures (or fingerprints) are stored. A major difference is that the quotient filter relies on linear probing for resolving collisions, rather than on cuckoo hashing. Linear probing is considered *cache-friendly* in case of collisions as it preserves spatial locality. It is, however, also known that it leads to clustering under high load, which makes lookups more costly as the number of comparisons increase as well as the number of (potential) cache misses. Thus, decoupling the lookup costs from the hash table’s load factor is one of the most notably achievements of the cuckoo filter.

More recently, Breslow et al. proposed the *morton filter* [30, 31], a cache-friendly variant of the cuckoo filter. The main motivation behind morton filters was to eliminate the second (random) memory access, which could significantly improve performance when the filter instance exceeds the last-level cache in size. This is achieved by introducing a compressed block format which offers better locality and hence reduces the number of relocation. In consequence, more lookups can be served without consulting the alternative candidate bucket. Probing a compressed block is computationally more intense compared to an uncompressed bucket. However, the increased computational costs are significantly lower than an additional cache miss.

The *vacuum filter* [172] is another variant of the cuckoo filter, which also addresses the issue of always having two random memory accesses during lookups. The authors propose to use a modified function to determine the alternate bucket, where the maximum distance between the current and the alternate bucket is reduced to improve spatial locality. Spatial locality comes at the cost of higher collision probability, which may result in poor space-efficiency. To compensate for this, vacuum filters use multiple ranges (at least two) in which the alternate bucket can be found. For instance, 75 % of the elements use a narrow range and 25 % are spread out over the entire filter. The actual range to use is thereby determined by the least significant bits of the signature.

Besides preserving spatial locality of signatures, the vacuum filter’s primary goal was to increase the space-efficiency of cuckoo filters. The standard cuckoo filter requires the number of buckets in the hash table to be a power of two, which results in space-inefficiencies as up to 50 % of the hash table might not be populated. Vacuum filters have overcome this limitation. However, this issue has been successfully addressed before in [A] and [127], by replacing the XOR operation in Equations 2.3 and 2.4 by different involutory functions.

The vacuum filter as well as the *dynamic cuckoo filter* [43] extend the standard cuckoo filter to support dynamic resizing. Both make internal use of mul-

tuple (linked) filter instances, similarly to the scalable Bloom filter [13]. The approach naturally negatively affects lookup performance, as multiple filter instances need to be probed. Further, the approach also impairs the false-positive rate, which is in contrast to the scalable Bloom filter. The authors of the vacuum filter therefore propose to periodically reconstruct the filter to avoid these issues.

The Bloomier filter [42, 39] is another signature-based AMQ data structure, which is not only suitable for approximate membership queries but also for approximating arbitrary functions. A variant of the Bloomier filter [59] builds the foundation of the *xor filter* [71], which is the most recent data structure mentioned in this thesis. Without going into details, the xor filter is the most space-efficient AMQ data structure currently known. The space consumption per element is $1.23 \cdot \log_2(1/f)$ or even $1.0824 \cdot \log_2(1/f) + 0.5125$ when the optional compression is enabled; the authors refer to the latter as xor+ filter. In terms of lookup performance, the xor filter can compete with cache resident cuckoo filters, as it issues slightly less CPU instructions. When the filter exceeds the size of the last-level cache, xor filters perform worse than cuckoo, as they use three random memory accesses, whereas the cuckoo only requires two. A noteworthy difference of the xor filter is that it does not support dynamic inserts, or any other manipulating operations. Instead, it is built holistically for the entire input set. Thus, the xor filter is a static data structure, and can therefore not be considered as a drop-in replacement for Bloom and cuckoo filters, it however fits well in database applications, such as semi-join filtering and for LSM trees, cf., Section 2.1.

2.3 Performance Optimality

With our research on performance-optimal filtering, we shed light on the questions, which filter structure to use in certain situations and on how the filter structure should be parameterized to accelerate a given workload the most. In contrast to prior (mostly theoretical) work, which mostly focused on minimizing the space consumption of filter structures, we aim for maximizing the end-to-end performance.

To determine what is performance optimal, we take additional factors into account: (i) the actual time spent in filter lookups, denoted as t_l , (ii) the work time t_w that is saved later on, when an element is filtered out, and (iii) the fraction of negative lookups $1 - \sigma$. The question is how much lookup time (t_l) to invest and how much work time (t_w) could be saved and how often this pays off ($1 - \sigma$). We formulate this problem as determining the filter configuration with the least *overhead*, where the overhead consists of (i) the unnecessary work the system has to perform when false positives occur, and (ii) the work involved by querying the filter data structure.

We formally define the filtering overhead denoted as ρ for a given filter

configuration F as:

$$\rho(F) = t_l(F) + f(F) \cdot t_w. \quad (2.10)$$

Here, $t_l(F)$ and $f(F)$ denote the lookup time and the false positive rate of the filter configuration F . The performance-optimal filter configuration F^{opt} among all possible configurations \mathcal{F} is then:

$$F^{opt} \in \mathcal{F} : \nexists F \in \mathcal{F} : \rho(F) < \rho(F^{opt}) \quad (2.11)$$

Note, we thereby assume that all considered filter structures are branch-free (cf., Section 2.2.1), i.e., the costs for positive and negative lookups are the same. This assumption allows for defining ρ independently from the parameter σ . However, even though the equation is simplified, it is valid for high-throughput situations. Our experimental analysis revealed that branch-free implementations offer the best performance, when filters are queried at very high rates (and t_w is low). On the other hand, in situations with the query rates are lower (typically in conjunction with high t_w), t_l becomes negligible small compared to the work caused by false positives ($f(F) \cdot t_w$), such that Equation 2.10 is still a good approximation, irrespective of whether the filter is branch-free or not. Nevertheless, parameter σ is still needed to determine whether using a filter is beneficial at all. Namely, when the following holds:

$$\rho(F^{opt}) < (1 - \sigma) \cdot t_w \quad (2.12)$$

To determine F^{opt} , we assume that the (context-)parameters n , and t_w are known, e.g., through selectivity estimations or runtime profiling. For the filter parameters, e.g., the filter size m , the number of hash functions of a Bloom filter k , or the signature size of a Cuckoo filter l , analytical models are available to estimate (or approximate) the false positive rates. However, t_l is a physical cost metric, which is harder to model, as it depends on the underlying hardware and on implementation details. We therefore propose to collect the actual filter lookup costs by performing microbenchmarks on the target platform as part of a one-time calibration phase².

2.3.1 Bloom Filter Variants for High Throughput

As our research focus on high-throughput scenarios, we developed two novel Bloom filter variants, which offer significantly faster lookups (lower t_l) and are highly optimized for modern hardware.

Register-Blocked Bloom Filter. Register-blocking is an extreme case of blocking [144], where the block size is reduced to the size of a CPU register, i.e.,

²More recently, the calibration approach has been extended and successfully applied in the context of GPU-accelerated filtering [75]. The source code is freely available on GitHub: <https://github.com/peterboncz/bloomfilter-bsd/tree/gpu/amsfilter>

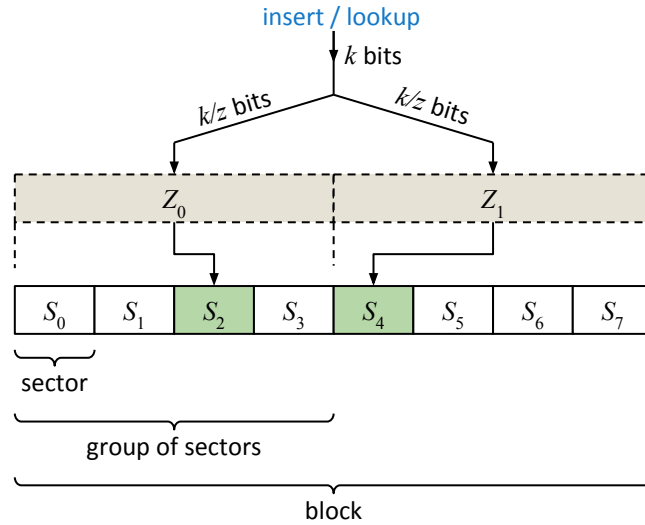


Figure 2.4: Partitioning scheme of cache-sectorized Bloom filters: The bits are concentrated in z words spread over a cache line. (Figure from [A])

32- or 64-bit. The approach significantly reduce computational efforts, as all k bits can be tested in a single comparison and only a single processor word needs to be loaded.

Cache-Sectorized Bloom Filter. Our second Bloom filter variant combines the advantages of cache-line blocking [144] and sectorization, and thereby removes the aforementioned parameterization constraints of the latter (cf., Section 2.2.1). With cache-sectorization, blocks are partitioned in word sized sectors. Multiple sectors are then logically grouped together. When an element is inserted, k/z bits are set in each group, where z denotes the number of groups per block. Inside each group, the k/z bits are set within a single sector, as illustrated in Figure 2.4. Since a sector corresponds to a processor word, multiple bits can be tested in a single comparison, similarly to register-blocking. Across the groups, all operations remain independent and can be performed in parallel. The main advantages over (plain) sectorization are (i) k can be tuned more flexibly, as a integral multiple of z rather than as a multiple of the number of sectors, (ii) the k bits are spread over the entire cache-line, and therefore our approach can be considered being *memory bandwidth efficient*.

2.3.2 Experimental Analysis

For our experimental analysis, we considered various hardware platforms, where we measured the end-to-end filter performance for varying n and t_w . We evaluated highly optimized blocked Bloom filters and Cuckoo filters and determined which filter type and which filter parameterization performs best for each individual combination of $\langle n, t_w \rangle$. We found that:

- on all platforms blocked Bloom filters outperform Cuckoo filters in high-throughput scenarios (by up to $3\times$), due to their lower lookup costs.
- a false positive rate of 0.0001 to 0.01, as provided by blocked Bloom filters, is sufficient for fast moving workloads.
- in low-throughput scenarios, e.g., where filtering is used to avoid disk I/O, Cuckoo filters perform better, due to their lower false positive rate.
- our register-blocked filter extends the spectrum where key filtering is beneficial. Since the average lookup time is reduced to ≈ 1 CPU cycle, a register-blocked Bloom could be installed, even when t_w is low, e.g., in the order of a cache miss, or even lower.
- our cache-sectorized filter dominates the classic blocked Bloom filter in high-throughput situations. In the other cases, it slightly falls behind, as cache-sectorization negatively affects the filter’s accuracy.
- SIMD-optimizations improve the (average) lookup throughput by up to $10\times$ for Bloom filters and by up to $7\times$ for Cuckoo filters on AVX-512 platforms; or $6\times$ and $4\times$ on AVX2 platforms.

2.4 Conclusions

Our research has shown that carefully implemented filters could improve the lookup performance by several factors, and thereby significantly extend the spectrum (towards lower t_w s) in which filtering is beneficial. It also showed, that the end-to-end performance can be improved by factors, when filters are configured properly for the given context. Our formal model considers the filter performance in a larger end-to-end context and combined with the proposed one-time calibration approach (through micro-benchmarking) it forms a general and practical framework for determining the performance optimal filter configuration for arbitrary workloads. The only two parameters that need to be provided are the number of elements in the filter (n) and the amount of work saved (t_w) when an element is filtered. We expect that both parameters can be easily determined as they are closely related to the actual application. The calibrated framework then maps to an actual filter configuration and thereby removes the burden of finding an answer to the question, what the desired false-positive rate is.

3 Efficient Control Flow Handling in Compiled Queries

In this chapter we investigate the performance and the CPU efficiency of compiled queries. In particular, we focus on the case where multiple tuples are processed in a data-parallel (SIMD) fashion. In such situations, where SIMD vectorization is combined with data-centric query compilation, *control flow divergence* may result in sub-optimal utilization of the available vector-processing units of modern CPUs. In particular query pipelines that involve many code branches are highly affected and therefore offer significant potentials to improve query performance.

Before we present our novel algorithms and strategies that address the negative performance impacts of control-flow divergence, we quickly recap the foundations of data centric query compilation and data-parallel pipelines.

3.1 Data Centric Query Compilation

Data centric query compilation, as proposed by Neumann [128], translates sequences of relational algebra operators into highly efficient low-level code, which subsequently is compiled to native machine code using an optimizing compiler framework such as LLVM [98]. The generated code thereby consists of a tight loop that iterates over the input tuples. The loop is generated by the operator at the pipeline source, e.g. a table scan. The loop body is populated by the other operators of the pipeline. Figure 3.1 depicts a simple query plan and the corresponding generated code; for illustration purposes, the algebraic operators emit simple pseudo code and the different colors connect the relational operator on the left-hand side with their generated code on the right-hand side. When such code fragments are compiled and executed, the currently processed tuple can be kept in CPU registers as long as necessary, i.e., in the ideal case each individual tuple (or each component thereof) is loaded and evicted only once. Thus, this *data centric* query execution model eliminates costly memory materializations along the query pipeline. Conceptually, memory materializations happen only the end of the pipeline, i.e., at the pipeline sink, which is in contrast to *operator centric* execution models, as used for instance in MonetDB [6] and MonetDB/X100 [26, 192] (also known as VectorWise and Actian Vector), where the individual operators transfer tuples via memory. In the data centric model, the code of the operators is fused together, which basically allows for transferring tuples across operator

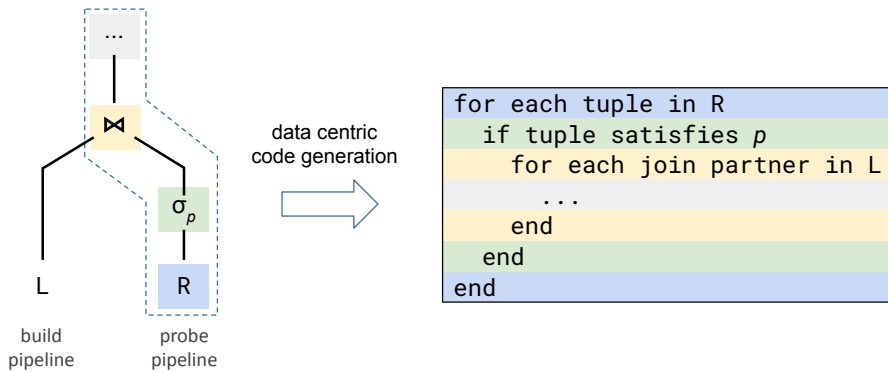


Figure 3.1: A simple example for data centric query compilation.

boundaries with zero overhead, i.e., no memory accesses nor (virtual) function calls are involved. Another major difference between these two models is the *amount* of tuples that are transferred between the operators at once. In the data centric model, the unit of transfer (UoT, as defined in [57]) is a single tuple, whereas VectorWise transfers a batch of tuples (or a vector) at once. Thereby, the batch size is chosen large enough to amortize the interpretation overhead (i.e., function calls) and small enough to fit in fast cache memory to avoid costly round trips to DRAM. The tuple-at-a-time processing on the other hand seems to be inefficient at the first glance but in fact offers very high throughput [86] as it results in much better code and data locality.

3.2 Data-Parallel Pipelines

A down side of data centric tuple-at-a-time processing is that it does not fully utilize modern CPU capabilities. In particular, the instruction-level data-parallelism offered by modern SIMD instruction sets is not utilized. Even though most optimizing compilers support auto vectorization, the generated loop bodies are however structurally too complex for compilers to analyze and therefore the resulting native code consists of scalar (non-SIMD) instructions. A solution to overcome this limitation is to make the code-generating relational operators aware of the data-parallelism offered by the underlying hardware. When the *degree of parallelism* is known to the operators, the operators can increase the number of tuples that are processed at once accordingly. In other words, the generated code is explicitly vectorized, rather than relying on the (implicit) auto vectorization feature of the compiler. The degree of parallelism, or the number of components in a vector, is thereby determined by the bitwidth of corresponding attributes and the bitwidth of SIMD registers. For instance, on AVX-512 platforms, eight 64-bit values are processed in parallel.

```

for each vector in R
  if at least one vector element satisfies  $p$ 
    if at least one vector element has a join partner
      ...
    end
  end
end
end

```

Figure 3.2: The explicitly vectorized code of the query shown in Figure 3.1.

3.3 Control-Flow Divergence

With vectorized pipelines we can utilize the SIMD capabilities of modern CPUs. Figure 3.2 shows the vectorized version of the query pipeline from Figure 3.1. The most important differences to the scalar version are the adjusted branching conditions. For instance, the predicate p is evaluated against an entire vector of attribute values rather than against a single scalar value. Thus, there might be some elements in the vector that satisfy p , whereas other elements don't. In this (likely) situation, the code of the subsequent operator(s) still needs to be executed as otherwise qualifying elements would be discarded. This applies in general, if at least one element satisfies p . Only if none of the elements qualify, the if-branch can be skipped. When the if-branch is taken and non-qualifying elements are present, then the non-qualifying elements need to be excluded from further processing.

In general, when various elements of a vector would take different branches, a data parallel program has to execute all of these branches, whereas in each branch only the corresponding SIMD lanes are set to active. This situation is known as *branch divergence* or *control-flow divergence*. The major problem of control-flow divergence is that non-qualifying elements consequently waste precious compute resources as not all available SIMD lanes are utilized. In the simple case of evaluating a selection predicate, all SIMD lanes containing non-qualifying elements are set *inactive* within all subsequent operator(s).

Basically any conditional jump in a program may cause control-flow divergence. In the above example, elements are either discarded entirely (filtered out) or further processed. Thus, it is an extreme case of control-flow divergence. In more general cases, SIMD lanes may only be temporarily inactive, which in particular happens during join processing. The probe side query pipeline thereby involves traversing a pointer-based data structures, like a hash table, to find join partners. A hash table lookup, for instance, might require several comparisons and to follow multiple bucket pointers until a join partner for the current tuple is found. In doing so, some SIMD lanes terminate their search earlier than others. The corresponding lanes are then temporarily set inactive, until all remaining search instances terminated. The resulting underutilization of the SIMD processing units was the motivation for our re-

search, in which we investigate on the performance penalties of control-flow divergence and how this problem can be efficiently solved in the context of compiling query engines. Our approach is briefly presented in the following section. The full publication can be found in Section B.

3.4 Countering Underutilization

With our research on divergence handling in modern database systems we contribute two novel algorithms for the AVX-512 architecture, which allow for fine-grained assignment of new tuples to idle SIMD lanes. We refer to these algorithms as *refill algorithms*. Further, we present two strategies for integrating the refill algorithms with compiled query pipelines. These *refill strategies* determine (i) where the current utilization of the SIMD lanes takes place and (ii) how new tuples are assigned to the idle lanes. In all cases, refilling does not incur costly memory materializations of the in-flight attribute values, i.e., all active values remain in SIMD registers. Thus, our approach is considered non-pipeline breaking, even when the more restrictive definition of a pipeline breaker of Neumann [128] is applied.

Refilling is essentially about copying new attribute values to desired positions in a destination vector register, whereas the desired positions are typically the idle SIMD lanes. Thereby, the source values are either fetched from a memory location or from another vector register. Further, we can distinguish between the cases where the source values are stored consecutively or at random positions. We provide efficient AVX-512 algorithms for all combinations.¹

We identified two base strategies for integrating our refill algorithms with compiled query pipelines. Our first strategy performs refills exclusively at the pipeline source, e.g., as part of the table scan operator. The strategy introduces branching code in the upper part of the pipeline, and when the lane utilization falls below a certain threshold, the control-flow is returned to (the generated code of) the operator at the pipeline source. At the pipeline source, the idle SIMD lanes are then assigned new values. When the control-flow is returned to the pipeline source, the active elements remain in vector register, and therefore their corresponding lanes need to be *protected* from being modified or overwritten. Lane protection requires just a bit of bookkeeping on a per operator basis, and introduces very little overhead (a few bitwise operations). Lane protection, however, inherently causes an underutilization of SIMD lanes in the code path between the pipeline source and the operator that branched out in the first place. Thus, the strategy should be only applied to operators that are close to the pipeline source. Our second refill strategy makes use of small buffers to counter underutilization. When an operator encounters an underutilization, it either refills with values from this buffer, or in case when the buffers does not contain enough values, it evicts the remaining active values

¹For details, we refer the reader to [B]. The source code is available at https://github.com/harald-lang/simd_divergence.

from the pipeline to the buffer and returns the control flow, i.e., the operator defers the processing of these values and uses them later on, when the lane utilization falls below threshold. In contrast to our first strategy, all SIMD lanes are empty when the control flow returns. Therefore, the other operators are not affected. Our implementation uses additional vector registers as tiny buffers and our refill algorithms are used in both directions, to refill the pipeline as well as to flush the active elements to the buffer register.

We evaluated our approach in a main-memory setting with several database operators that are subject to control-flow divergence, and we compare to scalar (non-vectorized) pipelines, to vectorized pipelines without divergence handling as well as to the approach of Menon et al. [115], where the intermediate results are densely materialized in memory at operator boundaries. We found that our approach can reduce the end-to-end query runtime by more than 30%, compared to a vectorized pipeline without divergence handling. In particular with more involved query pipelines, which perform geospatial point-polygon joins, our approach shows an up to $3\times$ higher throughput than the materialization approach. In contrast, the materialization approach shows better results with simple query pipelines that are memory bound, rather than compute bound. However, its performance quickly degrades when divergence handling is performed at multiple operator boundaries along the pipeline, or when the number of attributes (of the processed tuples) increases. In both cases, our approach scales significantly better due to its lower overhead.

3.5 Conclusions

Our refill strategies and algorithms show that data-centric query pipelines can still be efficient even when they are vectorized via SIMD. In particular queries that involve selective predicates or traversing irregular pointer based data structures like hash tables or radix trees highly benefit from our efficient refill algorithms because our approach allows for fully utilizing the available vector processing units without the need for costly memory materializations.

4 Space- and Time-Efficient Bitmap Indexing

In this chapter we present a novel method for compressing bitmaps. The work is closely related to bitmap indexing. While bitmap indexes and bitmap compression are in general orthogonal topics, the work on bitmap compression is heavily motivated by reducing the space consumption and the query time of bitmap indexes. In fact, compression became an essential part of bitmap indexes and the term bitmap index mostly refers to a *compressed* bitmap index.

Bitmap indexes have been used in database systems since the late 80's [131] and have been adopted by many commercial and open source database systems [33, 134, 113, 34]. Especially in read-optimized systems that are designed for data warehousing [41, 113, 161, 164, 183], bitmaps indexes have gained in popularity as they can accelerate predicate evaluation [131, 133, 125], as well as join [130] and aggregation queries [133, 34]. They have also been successfully used in scientific applications to analyze large data sets [160, 176, 78, 138, 155, 178, 163, 187], e.g., from astronomy and high-energy physics. The great success of bitmap indexes is based on their good performance with *high-dimensionality queries*, even when the individual attributes are not selective.

One major issue with bitmap indexes is that the available bitmap compression methods either provide high compression ratios or high performance, but not both. Performance-optimized state-of-the-art methods are up to two orders of magnitude faster than their space-optimized counterparts. The higher performance comes at the cost of an up to $3 \times$ higher space consumption. In particular, when bitmaps are more densely populated, state-of-the-art approaches show weaknesses regarding compression ratios, which makes them less suitable for indexing low cardinality attributes or for indexes that are designed to answer range queries. To address these issues, we invented a novel compression method that is efficient in both dimensions, space *and* time. Further, in contrast to most existing bitmap compression methods, our approach is robust regarding compression ratios within a wide spectrum of bitmap characteristics.

The rest of this chapter is structured as follows. In Section 4.1 we give an overview of the design space of bitmap indexes and how the different designs affect the characteristics of the individual bitmaps. In Section 4.2 we give an overview on existing bitmap compression methods, including discussions on their strengths, weaknesses and similarities. In Section 4.3 we then present our novel compression approach and we outline our next steps and possible future research directions in Section 4.4.

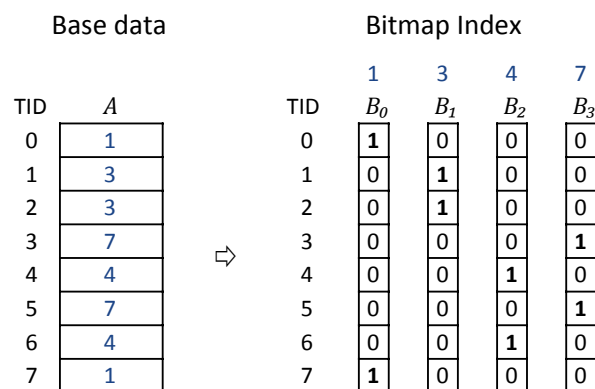


Figure 4.1: A basic bitmap index.

4.1 Bitmap Index Design Space

In this section we discuss bitmap indexes in general. We give an overview of the large design space and describe the individual aspects and methods involved when constructing these index structures.

We start with the most common configuration, which is also referred to as *basic* bitmap index [162]. Figure 4.1 shows such an index, created on a column *A* with four distinct values ($|A| = 4$). For each distinct attribute value of the column an individual bitmap is created. The positions of the set bits correspond to the tuple identifiers (TID), thus the index allows for answering selection queries of the kind $A = c$ without further computations, where c denotes a constant value. Conjunctive and disjunctive predicates can be evaluated by combining bitmaps using the corresponding bitwise operations.

This example also shows that the space consumption of a basic bitmap index is in $O(C \cdot n)$, where C refers to the attribute’s cardinality and n to the total number of indexed values (or tuples). On average, each of the C bitmaps receive a payload of $\frac{n}{C}$ 1-bits. Thus, the *density* decreases with an increasing number of distinct values. Formally, the density d of a bitmap B is defined as $d(B) = \frac{|B|}{n}$, where $|B|$ denotes the number of set bits, also known as the *population count*. The lower the density, the more likely the bitmap contains long *runs* of 0’s, and vice versa. Having such long runs (multiple consecutive identical bits) offers great compression potentials, which basically all bitmap compression techniques try to exploit. Several effective bitmap compression techniques have been proposed, which have been the enabler for indexing columns with higher cardinalities. — We discuss bitmap compression in detail in Section 4.2. — Besides the high space consumption, a basic bitmap index as shown above is in general not efficient in answering range queries, as multiple bitmaps need to be unioned. In the worst case, half of the bitmaps need to be accessed during query evaluation. To address these issues, several techniques have been proposed to reduce the size of an index and to efficiently support arbitrary range queries. These techniques are categorized [162] into *decomposition*, *binning*, *encoding*, and *compression*, which we briefly discuss in

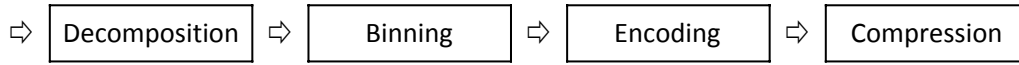


Figure 4.2: The techniques involved during bitmap index construction.

TID	{1,3} {4,7}		≤ 1 ≤ 3 ≤ 4 ≤ 7				b100 b010 b001		
	B_0	B_1	B_0	B_1	B_2	B_3	B_0	B_1	B_2
0	1	0	1	1	1	1	0	0	1
1	1	0	0	1	1	1	0	1	1
2	1	0	0	1	1	1	0	1	1
3	0	1	0	0	0	1	1	1	1
4	0	1	0	0	1	1	1	0	0
5	0	1	0	0	0	1	1	1	1
6	0	1	0	0	1	1	1	0	0
7	1	0	1	1	1	1	0	0	1

(a) binned

(b) range-encoded

(c) bit-sliced

Figure 4.3: Example bitmap index instances for the same base data as in Figure 4.1.

the following:

- *Binning* [94] is a simple yet powerful technique for reducing the number of bitmaps in an index. Instead of constructing a bitmap per distinct value, multiple values are put together and share the same bitmap. The resulting bitmap then represents a set of values, as illustrated in Figure 4.3a. Consequently, the index can no longer distinguish between the individual elements of the corresponding set. Selection queries therefore produce *false positive* results, and the raw data column needs to be consulted to *refine* the results. — Note that with multi-dimensional queries, the refinement may happen after the intermediate results from each dimension have been combined, as it could reduce the amount of raw data accesses during refinement [163]. — Nevertheless, depending on how the values are binned, the resulting index may still exactly represent a subset of the indexed values [155]; for instance to avoid accessing the raw data for frequently occurring selection predicates.
- *Encoding* defines how the contents of the bins are translated into a set of bitmaps. In the simplest case, for each bin a separate bitmap is constructed, which results in a basic bitmap index as depicted in Figure 4.1. Note that in that case, we conceptually have as many bins as there are distinct values, i.e., a 1:1 mapping. As mentioned earlier, the basic bitmap index is efficient for evaluating equality predicates, therefore this encoding scheme is also called *equality encoding*.

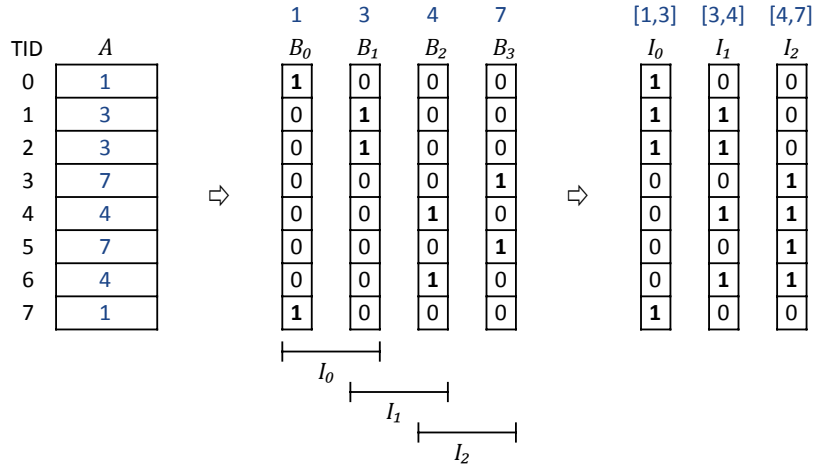


Figure 4.4: An interval-encoded bitmap index.

Other popular encoding schemes are *range encoding* [36] and *interval encoding* [37]. Both are applicable for ordinal attributes and require the bins to be sorted. In a range-encoded bitmap index, a single bitmap identifies the values that is less than or equal to the value the current bin maps to. Figure 4.3b illustrates a range-encoded index for the base data from Figure 4.1. For instance, the bitmap B_2 covers all tuples where $1 \leq A \leq 4$. This encoding allows arbitrary range queries and equality queries to be evaluated by accessing at most two bitmaps, e.g., to search for tuples with $3 \leq A \leq 4$, the bitmaps B_0 and B_2 need to be XORed. Range encoding however, does not reduce the size of the index, it actually introduces redundancies, which could negatively affect the compressibility of the individual bitmaps.

The more involved interval encoding on the other hand almost halves the number of bitmaps and introduces less redundant bits. The construction of an interval-encoded index is illustrated in Figure 4.4. Given a basic index with C bitmaps, the bitmaps are translated into $k = \lfloor \frac{C}{2} \rfloor + 1$ intervals I_0 to I_{k-1} . Each interval I_j spans the bitmaps (or bins) $[B_j, B_{j+\lceil \frac{C}{2} \rceil}]$, with $0 \leq j < k$. Similarly to range encoding, range and equality queries require at most two bitmaps to be accessed. For instance, the query $1 \leq A \leq 4$ is rewritten as $I_0 \vee I_1$ and $A = 3$ to $I_0 \wedge I_1$.

- *Decomposition* refers to decomposing the indexed values into multiple components before they are assigned to the available bins. A single value may therefore map to multiple bins, rather than just a single one. For instance, values may be decomposed into their multiples of 42 ($\lfloor \frac{v}{42} \rfloor$) and the residuals ($v \bmod 42$). Optionally, the bins associated with the individual components may be encoded differently.

An extreme case for such a *multi component* bitmap index is the *bit-sliced* index [133, 147], in which the values are decomposed bit by bit and are

assigned to the bins based on their significance. Figure 4.3c illustrates a bit-sliced index for the example data set from Figure 4.1. The index consists of three bitmaps, since the maximum value 7 could be represented with 3 bits. Bit-slicing, and decomposition in general, can therefore significantly reduce the overall index size to $\lceil \log_2(C) \rceil$ bitmaps (assuming the values are densely packed). However, it typically requires all bitmaps to be read during query evaluation, which harms performance.

The discussed techniques can be combined almost arbitrarily and optionally be arranged as a hierarchy of bitmap indexes [177, 180, 181, 155], which offers a more fine-grained trade-off between space requirements and query response times [156, 181]. Binning, encoding, as well as decomposing the indexed values have significant impact on the characteristics of the individual bitmaps [180]. In particular, binning, range and interval encoding tend to generate denser bitmaps, which are harder to compress [181].

4.2 Bitmap Compression

One of the earliest bitmap compression technique used and commercialized in a database management system is the *Byte-aligned Bitmap Code* [14] (BBC). It partitions the input bitmap in bytes and categorizes them either as *fill* bytes or *literal* bytes. A byte that either contains only 0-bits or 1-bits is thereby considered a fill byte. Bytes that contain both 0- and 1-bits are considered literal bytes. Once the input bytes are categorized, BBC inspects sequences of fill and literal bytes and tries to compress these sequences, for instance through merging multiple consecutive fill bytes of the same kind (either 0-fills or 1-fills), which is know as *run-length encoding*. More precisely, the BBC algorithms distinguishes among four different cases:

- 1) 0 to 3 fill bytes followed by 0 to 15 literal bytes (aka tail) are encoded as a header byte followed by the literal bytes. The header contains the fill bit, the length of the fill in number of bytes, and the number of subsequent literal bytes. The most significant bit of the header byte is set to 1, which is used to distinguish it from the other cases.
- 2) 0 to 3 fill bytes followed by a literal byte that contains a single bit that is different from the fill bit. BBC stores this case as a single header byte. The header contains the fill bit, the fill length, as well as the position of the *odd* (or *dirty*) bit within the last byte.
- 3,4) Cases 3 and 4 are similar to the first two cases, but they relax the length restriction of the fills. In the first two cases, the length was encoded in the header byte, using two bits. Thus, the maximum fill length was three bytes. In cases 3 and 4, the fill length is not encoded in the header byte. Instead, the header byte is followed by a *variable byte integer* [49] that allows for fills of arbitrary lengths.

Due to the fact that BBC uses two different run-length encodings (case 1 and 3), it is suitable for bitmaps that contain both, short *and* long runs. Further, BBC is also optimized for the case where a single random *dirty bit* disrupts a run (case 2 and 4). Instead of storing a literal byte with one bit set, the position of that dirty bit is encoded in the header byte. This approach is often called *piggybacking*.

The disadvantage of BBC is that decoding a compressed bitmap requires a lot of branching, which harms performance. The later proposed *PackBits* [58] (PAC) format addressed this issue by reducing the number of cases to distinguish. In contrast to BCC, PAC is a very simple run-length encoding scheme. It partitions the input bitmap into bytes and distinguishes literal runs and fill runs. Each run is preceded by a header byte that encodes the length of the run in the first seven bits and the type of the run in the most significant bit. Note that the header byte does not contain the fill bit (as in BBC). In fact, PAC uses a *fill-byte* that is replicated during decoding. Further, odd bits are not piggybacked in the header bytes, which impairs compression ratios. Thus, PAC trades space for decompression performance.

Both BBC and PAC are byte-based compression schemes and are considered not to fully utilize today’s hardware capabilities, as modern processors operate on 32 bit and/or 64 bit *words*, rather than on bytes. In 2006, Wu et al. presented the *Word-Aligned Hybrid* [179] (WAH), which is one of the most popular bitmap compression techniques nowadays. WAH addressed the issue of slow decompression speeds of its predecessors. Driven by the observation that byte-wise decoding does not fully utilize modern processors, the authors proposed to align the *encoded* runs to either 32-bit or 64-bit word boundaries, depending on the underlying hardware.

WAH partitions the input bitmap into groups of $w - 1$ bits, where w is either 32 or 64, depending on the underlying hardware. A group that only consists of 0’s or 1’s is called a fill group. A group that contains both, 0 and 1 bits, is called a literal group. Groups are then encoded either as fill or literal words. The most significant bit is used to distinguish between both. Similarly to the aforementioned compression schemes, WAH compresses only fills, by merging two or more consecutive fill words into a single word. Thereby, all fill words need to carry the same fill bit, which is stored in the second most significant bit position. The remaining $w - 2$ bits are used to store the run length as a multiple of $w - 1$. A literal word contains $w - 1$ bits of the plain bitmap as is. The most significant bit is used to identify the word as a literal word.

While word alignment increases the CPU efficiency during decompression and bitwise operations, it also negatively affects the compression ratios [175]. Follow-up work proposed several extensions to WAH or variants thereof to improve compression ratios:

- *Compressed N Composable Integer Set* [48] (CONCISE): The CONCISE approach is motivated by the observation, that real-world bitmaps often contain so called *dirty* bits or *odd* bits. These bits disrupt long runs and

force the WAH approach to encode the corresponding parts of the bitmap using three words. For instance, a bitmap like ‘000...0001000...000’ would be encoded as a fill word, a literal word, followed by another fill word. CONCISE therefore introduced *mixed fill groups* which in essence is a fill word that additionally stores the position of a single dirty bit. Given the bitmap from above, the CONCISE approach requires only two words to encode it, rather than three words as with WAH.

- *Position List WAH* [55] (PLWAH): PLWAH and CONCISE are motivated by the same observation and are therefore very similar. Both have been developed independently and were published in 2010. We therefore omit further details here.
- *COMPRESSED Adaptive index format* [68] (COMPAX): The COMPAX format as well addresses the issue of random dirty bits, but in contrast to CONCISE and PLWAH, COMPAX is able to deal with multiple (clustered) dirty bits. With this approach, up to two dirty *bytes* can be merged with a fill word, i.e., a fill word (F) that is preceded by a literal word (L) and also followed by another literal word, is merged into a single LFL code-word.
- *Variable Aligned Length WAH* [76] (VAL-WAH): The authors of VAL-WAH made the observation that in practice, many bits within a fill words remain unused. Within a fill word $w - 2$ bits are reserved to store the length of the run (in number of words), where $w \in \{32, 64\}$ refers to the bit-width of the an encoded word. Guzun et al. observed that with real-world data sets, runs are significantly shorter and therefore many bits are wasted. The proposed solution is to decouple the segment size, that is the unit of compression, from the size of an encoded word w which is determined by the underlying hardware, i.e., a segment size of b bits could be chosen and therefore up to $32/(b + 1)$ segments can be encoded in a single 32-bit word. Note that one bit is still required to distinguish between literal words and fill word, thus $b = 7$ refers to byte-alignment. VAL-WAH further allows for having multiple different alignments within a single bitmaps and thus improves compression ratios also when bitmaps have inhomogeneous characteristics with respect to the distribution of set bits.
- *Super Byte-aligned Hybrid* [89] (SBH): The SBH approach has more in common with BBC than with WAH, as it operates on bytes rather than on words. It however adopts the distinction between literal and fill words, which in this case are restricted to bytes. A literal byte represents 7 bits of the input bitmap as is. A fill byte either encodes a 0-run or a 1-run, whereas the run lengths are integral multiples of 7. The maximum run length that can be encoded using a single byte is 63×7 bits, which is

rather short and would make the approach ineffective with highly clustered or sparsely populated bitmaps. SBH therefore encodes longer runs using two bytes, which increases the maximum run length to 4095×7 bits.

- *Partitioned Word Aligned Hybrid* [168] (PWAH): The partitioned WAH sub-divides the 32- or 64-bit words of WAH into smaller equally sized partitions. Each partition either represents a literal part of the bitmap or a run. As the partition size, and therefore the alignment, can be chosen almost arbitrarily, PWAH can efficiently store shorter 0- and 1-runs, similarly to VAL-WAH. Additionally, PWAH introduces so called *extended fills*, where multiple fill partitions are fused together to space-efficiently encode longer runs; similarly to the two byte encoding used in SBH. The PWAH compression was a “side product” while developing a data structure for answering reachability queries on graphs. The compression technique gained very little visibility, and has not been further evaluated in other contexts nor has it been compared to other bitmap compression techniques, except for WAH.
- *Byte Aligned Hybrid* [106] (BAH): The more recently proposed Byte Aligned Hybrid approach is probably the most complicated compression scheme for bitmaps. BAH compresses the bitmap in segments and thereby it distinguishes between three different segment types: sequences of zero bytes (ZERO), sequences of literal bytes (LITERAL), and a frequent bit-pattern of length 32 encoded in either one or two bytes (ENCODABLE). The latter requires an additional dictionary with frequently occurring bit patterns. Unlike the compression schemes discussed so far, a BAH compressed bitmap does *not* consist of a single sequence of encoded words, it instead consists of a *main* byte array and three auxiliary arrays. Depending on the encoded byte read from the main array, the other arrays are consulted to reconstruct/decompress the bitmap. The decoder therefore has to distinguish among five different cases, whereas in three cases one of the other arrays need to be read. The BAH approach is clearly motivated by reducing space consumption rather than improving query performance.
- *Run-Length Huffman* [159] (RLH): The RLH compression approach was published around one year after WAH was proposed. The approach sets itself apart from the compression schemes discussed so far, since it (i) gives up on word (or byte) alignment and (ii) it does not rely on a hybrid encoding, namely storing runs and literal parts of the bitmap interleaved. Instead, a RLH compressed bitmap is represented as a sequence of variable length Huffman codes [79]. The encoding works as follows: A given bitmap is translated into a list of integers. Each integer represents a 1-bit of the bitmap. The *value* of the integer denotes the *distance* to the next 1-bit. These integer values are the *symbols* that are assigned Huffman codes to. Frequent symbols (distance values in this case) are

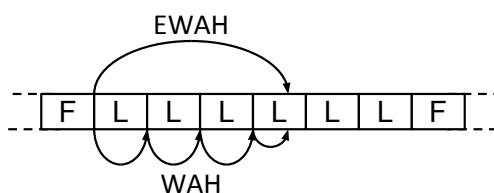


Figure 4.5: EWAH encodes the number of literal words (L) that are following a fill word (F) as part of the preceding fill word. The additional information allows for fast seeks within the literal part of the bitmap, whereas WAH needs to read all words in between.

represented with shorter codes, and vice versa. The compressed bitmap then consists of the encoded distances serialized into a bit string and a Huffman tree that is required for decoding.

The development of WAH was clearly motivated by the objective of achieving higher performance. In comparison to its predecessor BBC, it requires less branching, as WAH has to distinguish only between two types of code words, and since code words are 32- or 64-bit wide, it is more CPU efficient. Naturally, the design decisions made, compromised the compression capabilities. In general, WAH compressed bitmaps occupy more space than bitmaps compressed with BBC [175]. Follow up research identified this as an issue and proposed the aforementioned extensions to WAH or proposed new approaches based on run-length encoding. The compression techniques CONCISE, PLWAH, COMPAX, VAL-WAH, SBH, PWAH, BAH, and RLH were primarily motivated by saving space rather than improving access latencies. With regard to the space/time trade-offs, most of the research efforts happened into the opposite direction compared to WAH and PAC. To the best of our knowledge, there was only a single extension to WAH that was aiming for even higher performance:

- *Enhanced Word-Aligned Hybrid* [103] (EWAH): The Enhanced WAH is very similar to WAH as it also uses two different word types to encode a bitmap. The major difference is that EWAH additionally stores the number of literal words that are following the current fill word¹, similarly to the BBC case 1, as described on Page 31. This allows for faster random access within the literal parts of the compressed bitmap or allows for efficiently skipping over it to the next fill word, cf. Figure 4.5. To store the additional information, EWAH slightly compromises compression ratios. In particular, when runs are very long (larger than 2^{15} words), plain WAH is more effective. EWAH can therefore be considered a performance improved variant of WAH. Another advantage of EWAH

¹ The authors of EWAH use the term *marker word* rather than fill word. We however refer to it a fill word to emphasize the conceptual similarities among the different compression techniques we are discussing, and for the sake of better readability.

is that it occupies less space with poorly compressible bitmaps. A literal word in EWAH can carry a payload of w bits, where w denotes the word size, which is in contrast to a literal word in WAH, where one bit per word is required to distinguish literal words from fill words; thus the payload is $w - 1$ bits, which could increase the space of the bitmap by 3.125% with $w = 32$, or by 1.5625% with $w = 64$ respectively. EWAH on the other hand adds at most a 0.1% overhead when the bitmap is not compressible. It should be noted, that fast random accesses (and skips) within the literal parts of a compressed bitmap were already possible with the earlier BBC and PAC formats. This capability, however, got lost with WAH when the header bytes got removed and replaced by the indicator bits on a per-word basis.

All bitmap compression techniques discussed so far are based on run-length encoding or at least employed run-length encoding alongside other techniques. The major disadvantage of this approach is that random accesses have *linear time complexity*. In particular, to test a random bit at position k , all preceding bits within the range $[0, k)$ need to be decoded (or decompressed) beforehand.

Efficient random access is crucial with respect to performance when two or more bitmaps are *intersected*. In particular, when the bitmaps have different bit densities, we can avoid accessing all of the data of the denser (larger) bitmap. It allows the accesses to the relevant positions to be restricted, i.e., the positions where bits are set within the sparser (smaller) bitmap. In database systems, this could significantly accelerate the evaluation of conjunctive predicates [35]. Chambi et al. identified this opportunity and developed the *Roaring Bitmap* format [35]. Further, Athanassoulis et al. proposed an extension to WAH, called *Fence Pointers* [17], to overcome the linear access time. In the following, we briefly introduce and discuss these two approaches.

- *Roaring Bitmaps* [35, 102]: The Roaring Bitmap format sets itself apart from all aforementioned bitmap compression techniques, as it does *not* rely on run-length encoding. Instead, Roaring uses a different approach that relies on partitioning and using multiple representations on a per partition basis. More precisely, the plain bitmap is partitioned into chunks of 2^{16} bits. Each chunk is physically stored in a separate container. Roaring implements three different container types, where each container uses a different representation for the bitmap. Depending on the characteristics of the chunk, Roaring chooses the best suitable container type, i.e. the one that results in the lowest memory consumption. At the time of writing, Roaring supported the following container types:
 - Array Container: Within an array container, the bitmap is represented as a list of integers. Each integer represents a single 1-bit. The value of the integer refers to the index (position) of the set bit. Since the partitions of the bitmaps are limited in size to 2^{16} bits and the integer values represent the position within the partition, rather

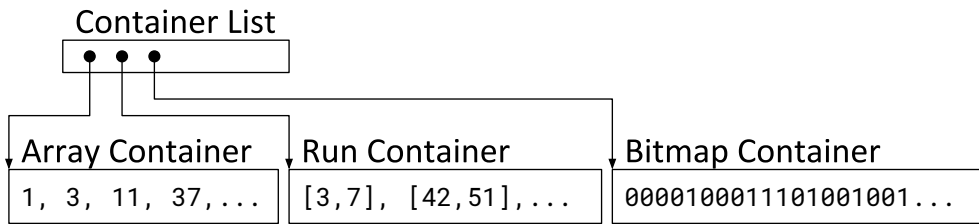


Figure 4.6: Roaring partitions the bitmap and stores each partition using the container type that occupies the smallest amount of memory. (Figure from [C])

than within the (typically larger) original bitmap, the integers are stored as 16-bit values. Each set bit in a partition therefore occupies 16 bits of memory. When the number of set bits exceeds 4096, this representation would require more space than a plain bitmap representation. To avoid this case, Roaring only makes use of an array container iff the number of set bits is less than or equal to 4096.

- **Bitmap Container:** The bitmap container stores the partition of plain bitmap as is. It serves as a fallback option when other container types are not able to represent the bitmap space-efficiently. For instance, when the number of set bits in a partition exceeds 4096 bits, Roaring (most likely) stores the partition as a (verbatim) bitmap container.
- **Run Container:** A run container [102] is employed when the input bitmap partition contains long 1-runs. Inside the container, these runs are stored as a list of integer pairs $\langle b, e \rangle$, where $[b, e]$ is the range spanned by the 1-run. Similarly to the array container, the integers are stored as 16-bit values. Thus, up to 2048 runs can be stored within the container without exceeding the size of the plain bitmap partition.

Figure 4.6 illustrates how an example bitmap is represented with Roaring. A Roaring bitmap can be seen as a two-level tree, whereas the first level contains the pointers to the individual containers, and the containers represent the second tree level. One of the most important properties of the Roaring bitmap format is that the bit positions stored in an array container, the runs stored in a run container, as well as the containers itself are *sorted*. Random accesses can therefore be performed in logarithmic time complexity. It takes at most two binary searches to determine the value of a random bit; only in cases where the partition is stored as a bitmap container, the second binary search is superfluous, since each bit is directly addressable. It should be noted that the asymptotical logarithmic time complexity is caused by the binary search within the container list rather than the search within the container, since the con-

tainers are bound in size. Within the container list, empty partitions are omitted and the pointers to the physically existing partitions are densely packed, which is the reason why a direct access via the partition offset is not possible and a binary search is required instead.

- *Fence Pointers* [17]: The fence pointers approach extends WAH (or similar formats) by an additional index structure. This index provides a mapping from the words of the plain bitmap to the corresponding words of the encoded bitmap. Naturally, indexing all words in the plain bitmap would result in a significant space-overhead. Therefore, fence pointers are supposed to index the plain bitmap more coarse-grained; e.g., one index entry per 10^3 to 10^5 words offered a good compromise between space consumption and access latency with the workloads tested in [17]. The indexing granularity in general offers a space/time trade-off that needs to be tuned within the application context. The space consumption of a fence pointer index is linear in the size of the plain bitmap, more precisely: $(\lceil n/(w-1) \rceil / g) \cdot s$, where n denotes the number of bits in the plain bitmap, w the word size, g the granularity, and s the size of a fence pointer in bits. Even though fence pointers have a high space consumption, they reduce the time complexity of random accesses. Independently from how g is chosen, random accesses on the (run-length) compressed bitmap can now be performed in constant time. In worst case g encoded words need to be read to test an arbitrary bit. To the best of our knowledge, this is a unique feature among all currently existing bitmap compression methods.

Both approaches provide efficient random access, whereas the asymptotic cost for the Fence Pointer approach is lower. It should, however, be noted that both approaches could be either in constant or in logarithmic time complexity, with some minor changes. In Roaring, the binary search in the container list could be avoided if the pointers to the containers would not be densely packed and therefore would be directly addressable by the partition offset. I.e., when accessing the k^{th} bit, the pointer to the corresponding container would then be stored at position $k \gg 16$. On the other hand, the Fence Pointer approach could index the compressed bitmap rather than the plain bitmap. The size of the index would then be linear in the size of the compressed bitmap, rather than in the size of the plain bitmap. It would however increase the access latency, since a binary search would be required.

In any case, the Fence Pointer approach adds significant space overhead to WAH and even without this additional index structure, WAH has a significantly higher space consumption compared to Roaring. For instance, the experimental evaluation in [102] and [35] with real-world data sets showed that Roaring requires up to 50 % less space than WAH. Our experimental evaluation with a large variety of bitmap characteristics in [C] has further shown, that only in very rare cases, WAH is able to compress slightly better than Roaring.

Even though, the linear access time of WAH has been overcome with Fence Pointers, the high memory consumption makes this approach less attractive for practical applications. It also has not been further evaluated and compared to other bitmap compression techniques. In particular, it was not considered in the experimental study of Wang et al. [171]² Roaring, on the other hand, gained enormous popularity since its initial release. At the time of writing, Roaring was available in 11 programming languages and was widely adopted in database systems, e.g., in Druid³, Hive⁴, Kylin⁵, and Procella [40]; in full-text search engines like Solr⁶, and many other systems⁷.

The huge success of Roaring is credited to its high performance in database and information retrieval workloads, which in most cases boils down to compute the intersection of bitmaps [35, 102, 171]. Roaring can be up to two orders of magnitude faster than WAH for computing intersections [102]. The experimental study of Wang et al. [171] confirmed the superiority of Roaring over other bitmap compression techniques with regard to intersection performance with several benchmarks, including the relevant queries from the Star Schema Benchmark [135] (SSB) and TPC-H Decision Support Benchmark⁸.

With regard to space consumption, Roaring proves that it is possible to compress bitmaps significantly better than *word-based* compression schemes, e.g., like WAH and Concise, while offering fast access times. The compression ratios can however not compete with *byte-based* compression techniques, such as BAH or BBC, as shown in the following.

To experimentally compare the compression capabilities we make use of a benchmark setting that was defined in [35] and was used earlier to evaluate Roaring bitmaps [35, 102]. The benchmark consists of bitmaps that were created when indexing four different real-world data sets, namely CENSUS 1881, CENSUS INCOME, WEATHER, and WIKILEAKS. For each data set 200 individual bitmaps need to be compressed. Further, each data set comes in two flavors: *as is* and *sorted*. In the latter case, the raw input data is sorted before indexing, which is known to reduce the overall size of compressed bitmap indexes [103]. The reduced size is due to indexing sorted data results in bitmaps with longer 0- and 1-runs, as the 1-bits are more *clustered*. For the experimental evaluation we chose two word-based compression schemes, WAH and Concise, and two byte-based compression schemes, BAH and BBC. Thereby, we use BBC as the baseline, since BBC offers the lowest space consumption in almost all cases (except for the unsorted WIKILEAKS data). Figure 4.7 shows the space consumption of the compressed bitmaps relative to BBC. As mentioned earlier, Roaring significantly reduces the space consumption over

²Same applies for PAC, COMPAX, PWAH, BAH, and RLH.

³<https://druid.apache.org/>

⁴<https://hive.apache.org/>

⁵<https://kylin.apache.org/>

⁶<https://lucene.apache.org/solr/>

⁷We refer to the official Roaring web site for more details: <https://roaringbitmap.org/>

⁸<http://www.tpc.org/tpch/>

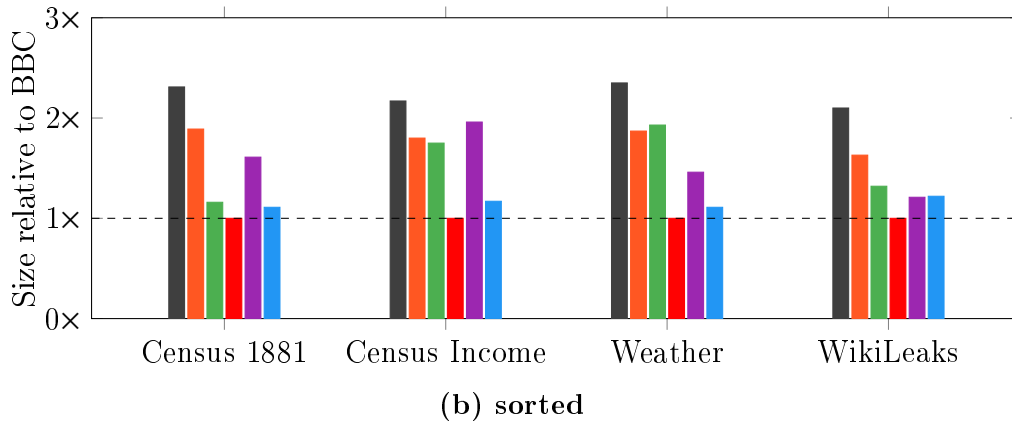
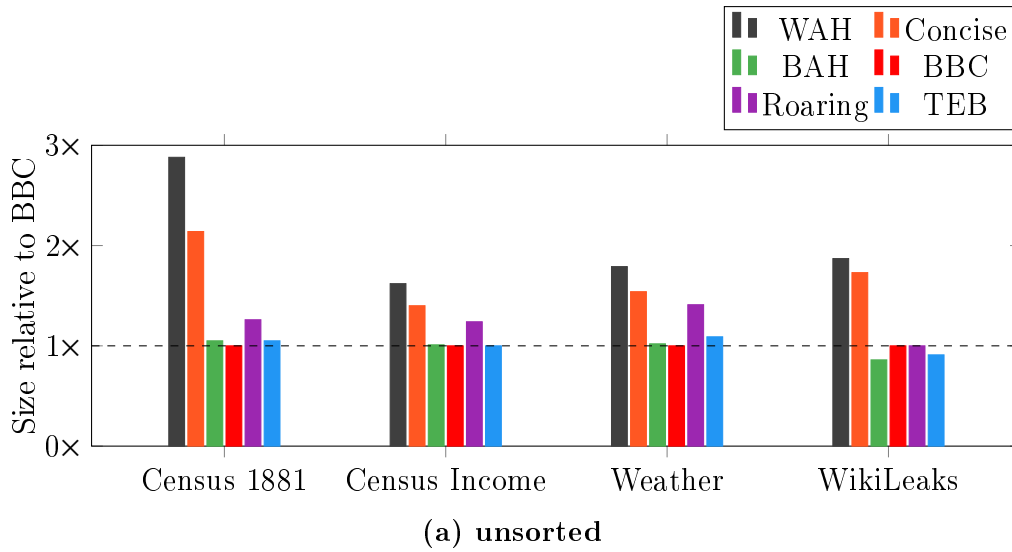


Figure 4.7: Space consumption of compressed bitmaps relative to BBC-compressed bitmaps. (Absolute values can be found in Table 4.1)

WAH, and Concise in most cases, but on the other hand, bitmaps compressed with Roaring consume up to $1.4\times$ the space of BBC-compressed bitmaps on unsorted data, and up to $2\times$ on sorted data. This significant difference in compression ratios was the main motivation for our research, that aimed for a technique that offers higher compression ratios, similarly to byte-based compression schemes, without giving up on the efficient random access property. Our approach, which we call *Tree-Encoded Bitmaps* (TEB), is briefly described in the following section. The full publication is found in Section C.

4.3 Tree-Encoded Bitmaps

Tree-Encoded Bitmaps (TEB) are a novel approach to representing bitmaps space-efficiently. In contrast to the previously existing bitmap compression

	WAH	Concise	BAH	BBC	Roaring	TEB
CENSUS 1881	34.39	25.56	12.58	11.95	15.08	12.59
CENSUS 1881 <i>(sorted)</i>	3.04	2.49	1.52	1.32	2.12	1.46
CENSUS INCOME	3.38	2.94	2.12	2.09	2.60	2.10
CENSUS INCOME <i>(sorted)</i>	0.66	0.55	0.54	0.31	0.60	0.36
WEATHER	6.83	5.88	3.89	3.83	5.38	4.16
WEATHER <i>(sorted)</i>	0.55	0.43	0.45	0.23	0.34	0.26
WIKILEAKS	11.12	10.27	5.11	5.94	5.93	5.41
WIKILEAKS <i>(sorted)</i>	2.90	2.25	1.82	1.38	1.67	1.68

Table 4.1: Space usage in bits per attribute value.

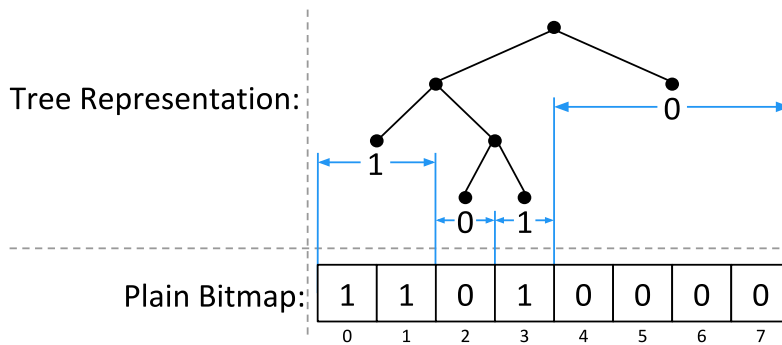


Figure 4.8: Tree-based representation of a bitmap. (Figure adapted from [C])

techniques, our approach does not rely on run-length encoding or integer arrays to encode bitmaps. Instead, a TEB employs a full binary tree to represent the content of a bitmap. Longer runs of 0's or 1's are thereby mapped to tree nodes closer to the root, and vice versa. I.e., nodes at the deepest tree level encode a single bit, nodes one level above span two bits of the bitmap, etc. An introductory example is illustrated in Figure 4.8. For instance, the leftmost leaf node of the binary tree represents a 1-run of length 2, starting at position 0 in the original bitmap, and the rightmost leaf node represents a 0-run of length 4, starting at position 4.

The construction of the tree-based representation is a two-phase process. In the first phase a perfect binary tree is established on top of the plain bitmap, so that each leaf node of the tree is associated with a single bit. The associated bit is the payload, or *label*, of the leaf node. In the second construction phase, the tree is pruned bottom-up. Thereby, sibling leaf nodes with identical labels are removed from the tree and the label is assigned to the parent node. Thus, with every pruning step, two leaf nodes and one label are removed from the tree structure. The important thing to note here, is that the pruning process removes only redundancies from the tree structure, but the original bitmap remains reconstructible. Further, consecutive identical bits, i.e., 0-runs and 1-runs, of the input bitmap are thereby *merged* together. The bottom-up

pruning can therefore be considered a *lossless compression* method.

A key component of TEB is the space-efficient encoding of the tree structure, for which we employ the *level-order binary marked* representation [81], which allows for representing the tree structure using one bit per tree node. The (pruned) tree is encoded in breath-first left-to-right order (i.e., level-order) as a sequence of bits. 1-bits thereby represent inner nodes and 0-bits leaf nodes. The corresponding labels of the leaf nodes are stored in an additional bit sequence. For instance, the tree from Figure 4.8 is encoded as $T = 1100100$, $L = 0101$, where T denotes the tree structure and L the labels of the leaf nodes. Beside the tree structure and the labels, a TEB contains an auxiliary data structure, a *rank dictionary* [81]. The dictionary enables efficient navigational operations within the encoded tree, which are required to support efficient random access, and consequently to efficiently compute bitmap intersections. Similarly to Roaring, TEB offers random accesses in logarithmic time complexity. It should further be noted, that some optimizations are performed during construction, which ensure that in worst case the size of a TEB does not exceed the size of the uncompressed bitmap; except for a small overhead caused by meta data. In worst case, the size of a TEB is $24 + 8 \times \lceil \log_2(n) \rceil + \lceil n/8 \rceil$ bytes, where n denotes the length of the bitmap⁹.

In terms of space efficiency, TEB offers significantly higher compression ratios than Roaring in a wide spectrum of bitmaps characteristics. Our experimental analysis in [C] shows that TEB saves up to 50 % space over Roaring with synthetic data sets, and up to 35 % with real-world data sets. The results in Figure 4.7 further show, that (i) TEB can compete with advanced byte-level compression schemes like BAH and BBC; and (ii) TEB is very robust regarding different bitmap characteristics, which is in contrast to BAH and Roaring. Both are less effective on sorted data. In particular with the CENSUS INCOME data set, Roaring occupies even more space than the word-based Concise method (cf., Figure 4.7b). Overall, Roaring consumes $1.22 \times$ the space of BBC on unsorted data (geo. mean among all four data sets) and $1.54 \times$ on sorted data, whereas TEB consumes only $1.01 \times$ the space of BBC-compressed bitmaps on unsorted data and $1.15 \times$ on sorted data, respectively.

In summary, our TEB approach is very close the byte-based compression schemes with regard to compression, without giving up on efficient random access. The tree structure, however, makes the implementation significantly more complex. In particular, to be competitive with the low access latencies of Roaring, TEB requires the underlying hardware to provide (at least) a fast population count (**popcount**) instruction, which most modern processors do. Other TEB algorithms can heavily benefit from instructions that are rather new or only available on Intel processors. For instance, our implementation of the tree-based compression algorithm makes use of the *bit extract* (**pext**) and *bit deposit* (**pdep**) instructions¹⁰, and our performance-optimized decompres-

⁹Assuming $n \geq 64$, otherwise the TEB size would be 32 bytes.

¹⁰We refer the reader to the Intel Intrinsics Guide for more details: <https://software.intel.com/sites/landingpage/IntrinsicsGuide/>

sion algorithm employs Intel’s AVX-512 SIMD instructions.

4.4 Future Work

Improvements in Compression. Motivated by the observation that the compressed size of TEBs is in many cases still significantly higher than the information theoretic lower boundary, which by the way applies to all compression methods discussed in this thesis, we plan to further investigate improving the compression ratio. A promising research direction is motivated by the Byte Aligned Hybrid (BAH) approach (cf., Section 4.2), which computes a dictionary with frequently occurring bit patterns. We plan to investigate whether something similar could be applied to TEB. The key idea thereby is to compute a dictionary of frequently occurring (sub-)tree structures, and instead of encoding the binary tree as a whole, parts of the tree are replaced by references to dictionary entries. We expect tree-based patterns to have a much higher impact on the overall compression ratios, since a tree pattern can be applied anywhere in the tree structure, and depending on its depth (relative to the root node) it can span a flexible number of bits in the original bitmap, ranging from a few bits up to the entire bitmap. In other words, a tree pattern decouples the span within the plain bitmap from its encoded size, which is in contrast to BAH, where a 16-bit pattern always spans 16 bits of the plain bitmap.

There are two major open challenges and research questions regarding extending TEBs with patterns: The first is to identify the proper set of tree patterns such that the overall TEB size is minimal. Second, since the tree gets fragmented, a new space- and time-efficient encoding needs to be found.

Lossy Compression. Another research direction, motivated by lightweight secondary indexing, is to integrate a lossy compression algorithm with TEBs. Lightweight index structures are used as filters and typically have a *one-sided error*, such that non-qualifying tuples (with some probability) pass the filter, but qualifying tuples are never erroneously filtered out. This, for example, can be achieved with bitmap indexes when binning is applied (cf., Section 4.1). An alternative orthogonal strategy would be to construct a bitmap index with lossy compressed individual bitmaps. The idea is to sprinkle in some 1-bits so that multiple shorter runs are merged into longer runs, and thereby reduce the overall space usage of the compressed bitmap. The tree structure of TEB easily allows to identify these odd 0-bits that interrupt longer 1-runs. When odd bits are observed during bottom-up pruning, the pruning process could then be continued even if two sibling leaf nodes have different labels. The inner node, that is collapsed into a leaf, is thereby assigned the label 1 and therefore introduces *false positive* bits. Figure 4.9 shows an example where a single false-positive bit allows for merging two adjacent 1-runs into single one and thereby removing four tree node and two labels. In general, depending on the labels l_1 and l_2 of two sibling nodes, denoted as $\langle l_1, l_2 \rangle$, we distinguish

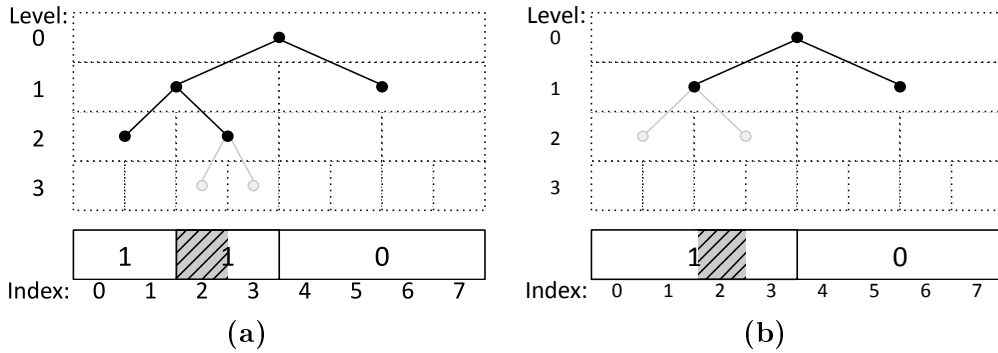


Figure 4.9: Pruning the leaf nodes at index 2 and 3 from Figure 4.8 causes a false positive bit at index 2 (shaded area). Afterwards, pruning can be continued at level 2 without further information loss (b).

three possible cases:

- $\langle 0, 0 \rangle$ can be pruned without an information loss. The number of false positives in this sub-tree is guaranteed to be zero.
- $\langle 1, 1 \rangle$ can be pruned without introducing false positives. But earlier pruning operations may have already introduced false positives.
- $\langle 0, 1 \rangle$ or $\langle 1, 0 \rangle$ introduces $2^{\log_2(n)-i}$ false positives, where i denotes the tree level (depth) of the pruned nodes.

It is important to note that the proposed approach only causes false positives, but no false negatives, since it introduces only 1-bits but no 0-bits. The introduction of false-positive bits aims for reducing the size of TEBs, thus we refer to TEBs that contains false positives as *lossy compressed* TEBs.

Lossy compression in general can either be applied to limit the space usage of the index or to reduce the space usage limited by a (user-)specified maximum error (or accuracy). In both cases however, further research is needed on how to identify the odd bits with the highest space-saving potentials *and* the lowest impact on accuracy. Furthermore, it needs to be investigated how binning compares to lossy compression with regard to accuracy and space consumption.

Publications

A Performance-Optimal Filtering: Bloom Overtakes Cuckoo at High Throughput

Harald Lang¹, Thomas Neumann¹, Alfons Kemper¹, Peter Boncz²

¹ Technical University of Munich

² Centrum Wiskunde & Informatica

Appeared in *Proceedings of the VLDB Endowment*, 12(5):502–515, Jan. 2019.
<https://doi.org/10.14778/3303753.3303757>

The content of this section is identical to the original publication. Only the format and the numbering have been adjusted.

In accordance with the TUM regulations for the award of doctoral degrees (TUM Promotionsordnung, 2014), a summary of the publication is included in the first part of this thesis. Please refer to Section 2.3. Furthermore, the printed version is included in the Appendix on page 173 ff.

The contributions of the thesis author to this publication are: the implementation, the evaluation, and authoring of substantial parts of the paper.

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.

A.1 Introduction

A Bloom filter [23] 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** [63] 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.

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 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 [25]. However, using a Bloom or Cuckoo filter can potentially backfire if it does not eliminate (enough) join candidates, certainly in cases where the selectivity $\sigma=1.0$ (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 [192]. 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 [53] (in order to reduce the number of structures to search), cold storage structures (idem) [12], 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 [93]. 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 [117], 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 $1-\sigma$ 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 ($1-\sigma$).

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. [144]. 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 bit-wise 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 $\sqrt{2} \approx 1.44$ 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 A.1 summarizes our key findings from our detailed experiments with regard to which filter type, Bloom or Cuckoo, performs best for a given problem

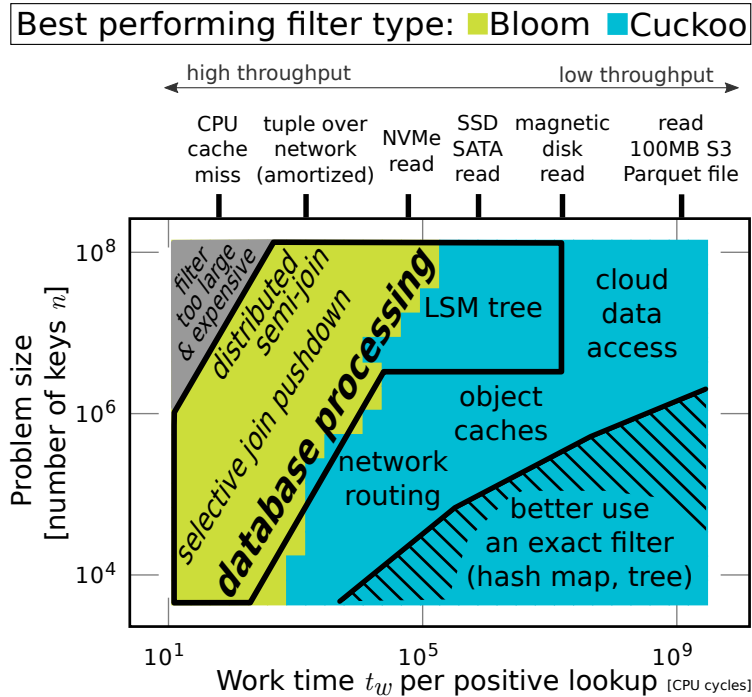


Figure A.1: Performance-optimal filter types for different problem sizes n and potential work savings t_w . Example reference points for t_w values are shown above the plot and applications inside the plot.

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, allow-

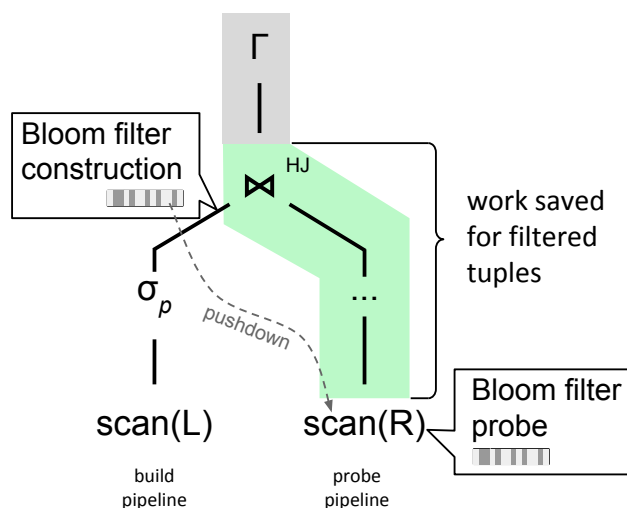


Figure A.2: Bloom filters with selective joins. Tuples without a join partner are filtered before entering the pipeline.

ing 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 structures¹.

A.2 Performance-Optimal Filtering

Figure A.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.

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.

¹Source code: <https://github.com/peterboncz/bloomfilter-repo>

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 [101] 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 [142], 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:

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

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: $\rho(F^{opt}) < (1 - \sigma) * t_w$.

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

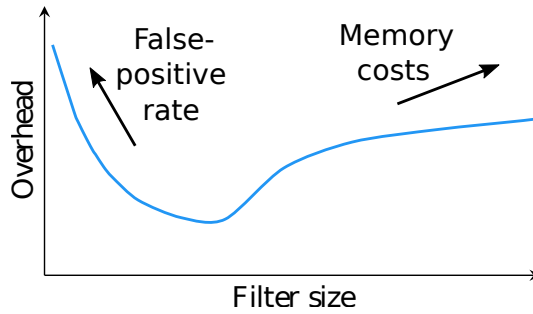


Figure A.3: Overhead ρ as a function in m for fixed configuration F , n and t_w .

due to more cache and TLB misses, t_l may increase). Figure A.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 [117] 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.

A.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.

A.3.1 Blocking

A *blocked* Bloom filter as proposed in [144] 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 A.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

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

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 [144] provides the following approximation for the false-positive probability:

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

where f_{std} denotes the false-positive rate of a classic Bloom filter with the arguments m , n and k (see Equation A.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 A.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 1/8th or 1/16th 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 A.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 A.4b shows the corresponding values for k , which indicates the computational efforts caused by hashing.

```

// Block addressing
h = consume  $\log_2(m/B)$  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  $\log_2(B/W)$  hash bits
    word = load word from block_idx + h
    h = consume  $\log_2(W)$  hash bits // Bit addressing
    found |= word & (1  $\ll$  h) // Bit testing
}
return found

```

Listing A.1: Lookup function of blocked Bloom filters.

```

// Block addressing
h = consume  $\log_2(m/B)$  hash bits
block = load word at position h mod ( $m/B$ )
search_mask = 0;
for each  $k$  do {
    // Bit addressing
    h = consume  $\log_2(B)$  hash bits
    search_mask |= 1  $\ll$  h
}
// Bit testing
return block & search_mask == search_mask

```

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

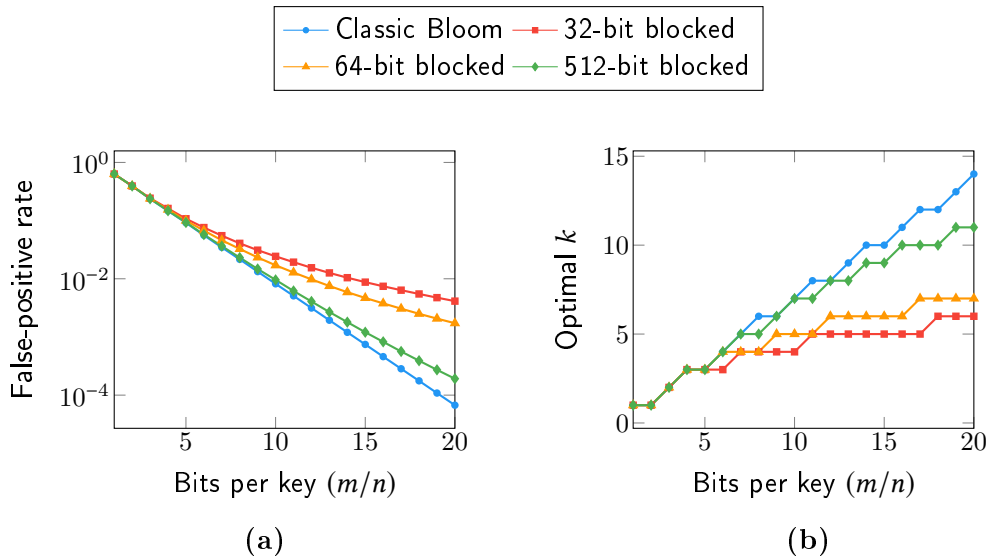


Figure A.4: Impact of blocking on the false-positive-rate and the hashing efforts.

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.

A.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 A.1 and A.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 [93]. The authors actually combined a *m/k -partitioning* scheme [92] 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 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 A.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 [144]. In contrast to m/k -partitioning as proposed by Kirsch et al. [92], 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 A.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 A.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 A.2 within each sector (in a sequential order).

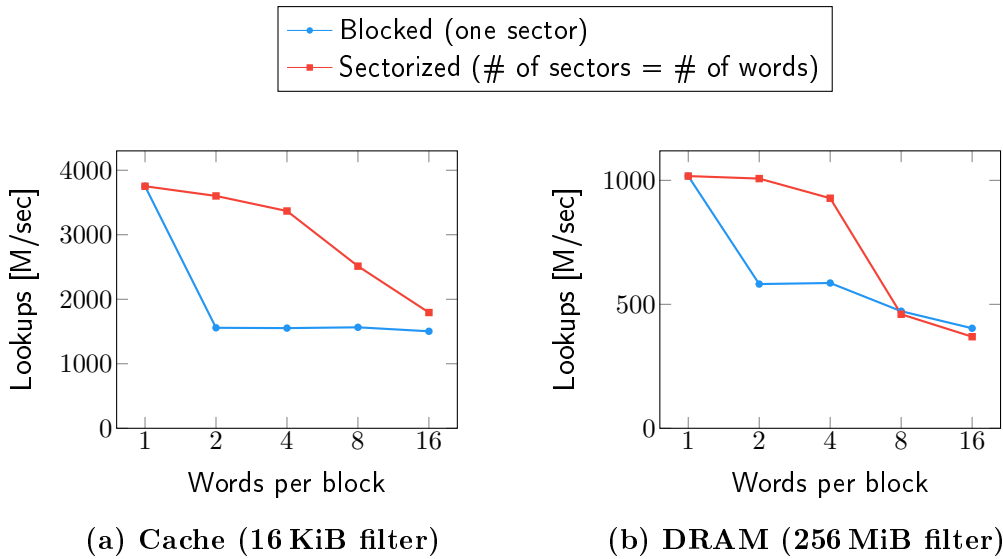


Figure A.5: Performance impact of sectorization for varying block sizes.

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 .

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

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 A.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 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 A.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:

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

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

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

A.4 Cuckoo Filter

With the Cuckoo filter [63], 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 [136] 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:

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

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

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

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 ($\sim 50\%$) 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 $b = 2, 4, \text{ or } 8$ 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

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

where l is the signature length in bits and α the load factor of the table. Thus, the false-positive rate primarily depends 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 A.8a. On the other hand, reducing the numbers of signatures per bucket b significantly improves the false-positive rate (see Figure A.8b), while coincidentally impairing memory efficiency due to lower load factors.

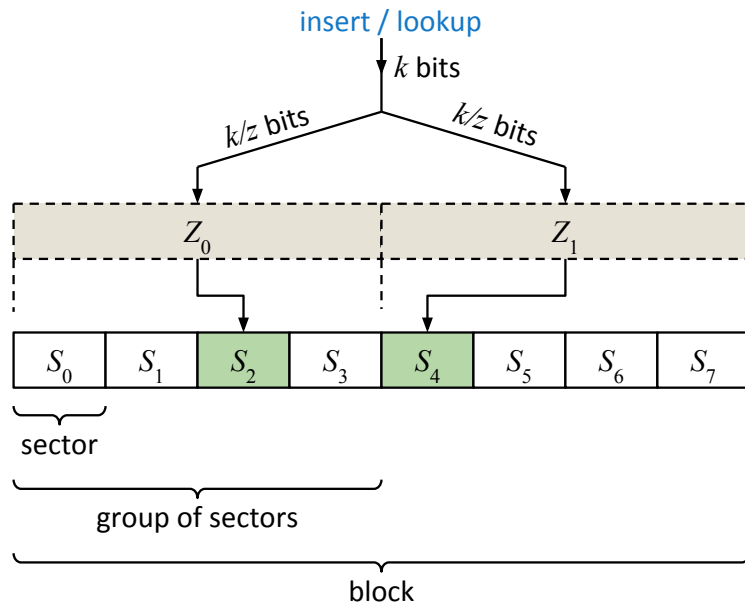


Figure A.6: Block partitioning scheme of cache-sectorized Bloom filters: hashing sets bits concentrated in z words spread over a cache line.

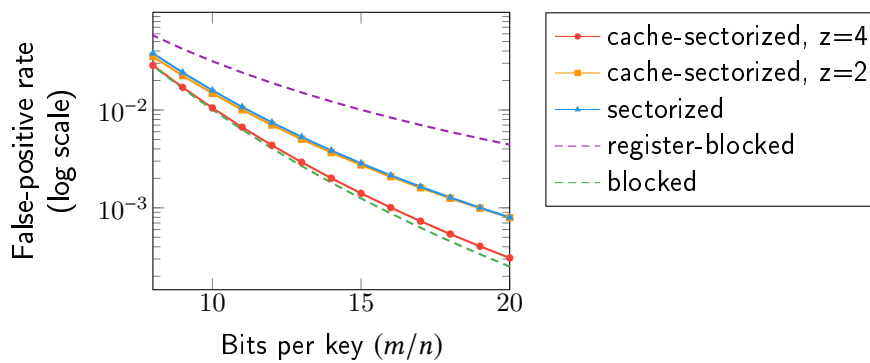


Figure A.7: Comparison of the false-positive rate of sectorized and cache-sectorized Bloom filters, with $k = 8$.

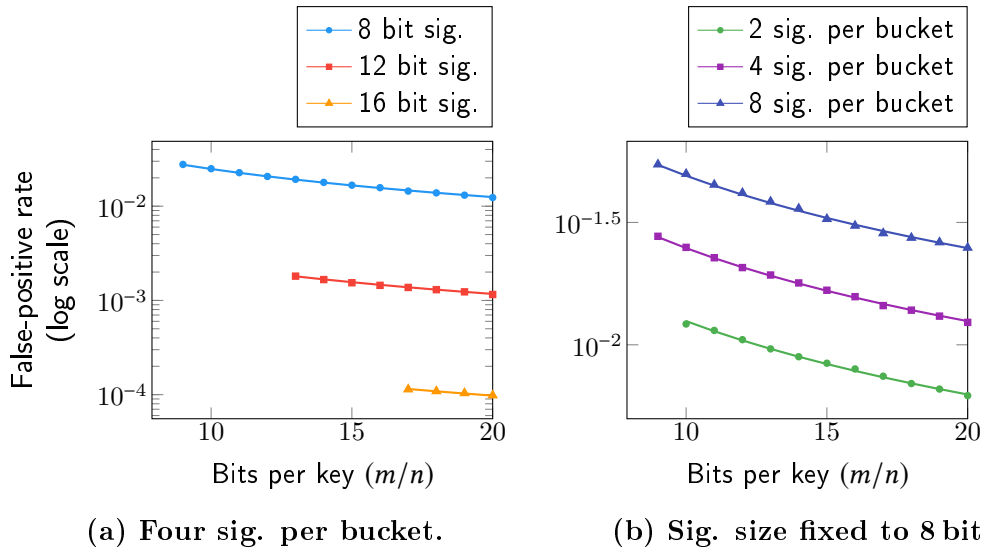


Figure A.8: The false-positive rate of Cuckoo filters for different signature lengths and bucket sizes.

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.

A.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.

A.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 than two cycles

(for low ks). Our actual C++ implementations of the Bloom filter contains functions are very similar to the scalar pseudocode in Listings A.1 and A.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 [10, 80]. 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.

A.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 ANDs (this applies, for example, to the Impala Bloom filter, the SIMD Bloom filter from [142], 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. [144] 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 $6\times$ 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* [84] to multiply with, a shift amount and a summand. On most platforms, the multiply-shift-add sequence 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 $i = \text{hash}(\text{key}) \bmod C$ is thereby replaced by

$$\begin{aligned} h &= \text{hash}(\text{key}) \\ i &= h - (\text{mulhi_u32}(h, \text{magicNo}) \gg \text{shiftAmount}) * h, \end{aligned} \tag{A.9}$$

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 block³ of the Bloom filter and a bucket in the Cuckoo filter, respectively. The *actual* filter size is therefore

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

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

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

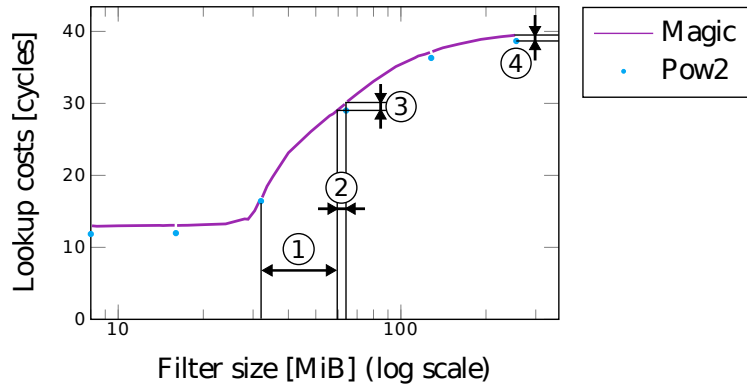


Figure A.9: Lookup performance of a cache-sectorized Bloom filter for varying filter sizes.

(see Equation A.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:

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

where C denotes the number of buckets.

Figure A.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.

A.6 Experimental Analysis

In this section, we present the results of our experiments, conducted on four different hardware platforms (see Table B.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

	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}

Table A.1: Hardware platforms

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 valid⁴ 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 $[10, 27]$ 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 A.10 summarizes the results from the four hardware platforms. For each $\langle n, t_w \rangle$ point, we report whether the

⁴Please 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.

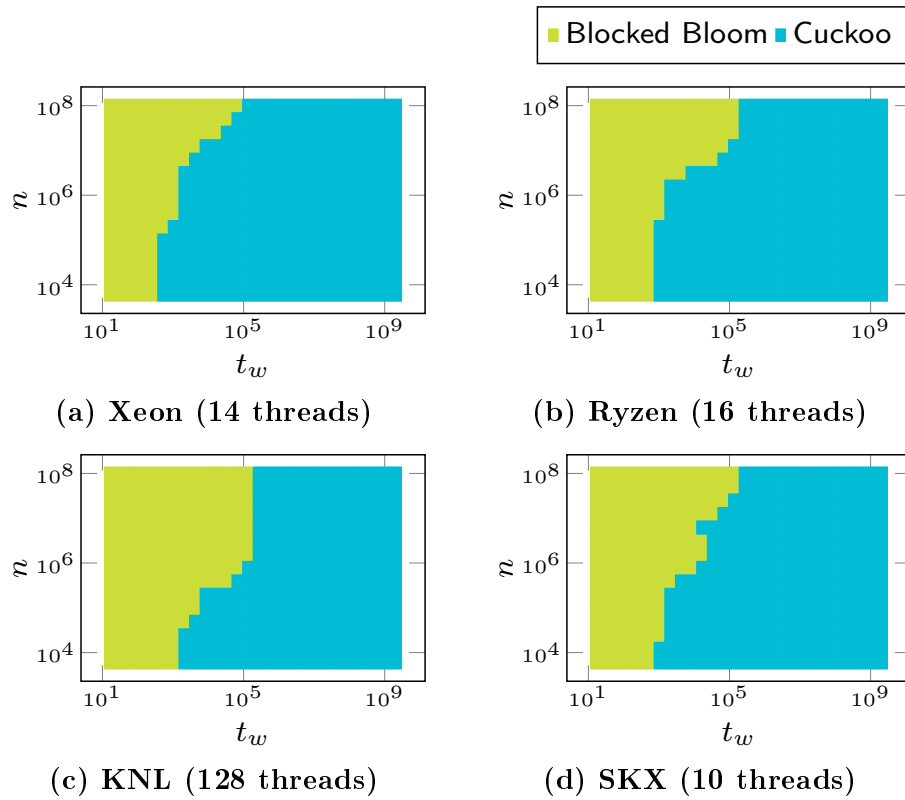
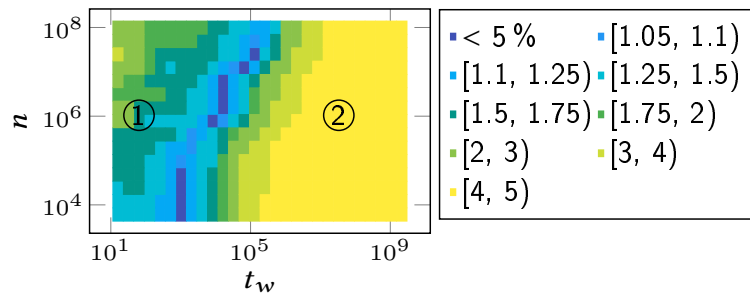
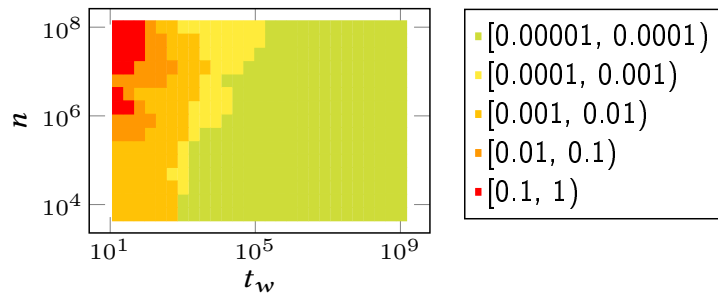


Figure A.10: Skyline of performance-optimal filters for varying n and t_w .



(a) Speedups of the best filter over its counterpart.



(b) False-positive rates of the best performing filters.

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

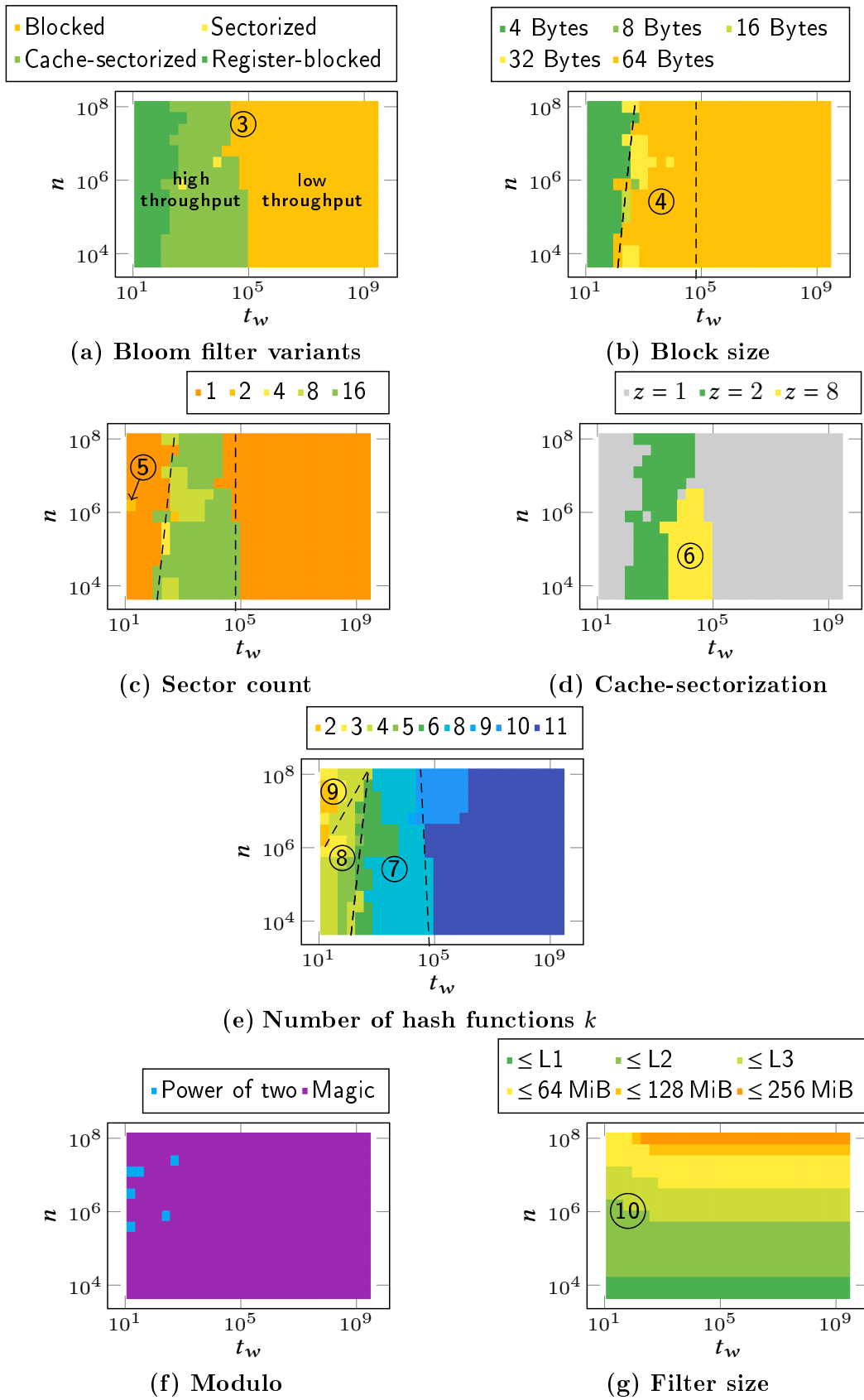


Figure A.12: Skyline of configurations of the best performing blocked Bloom filters (on SKX).

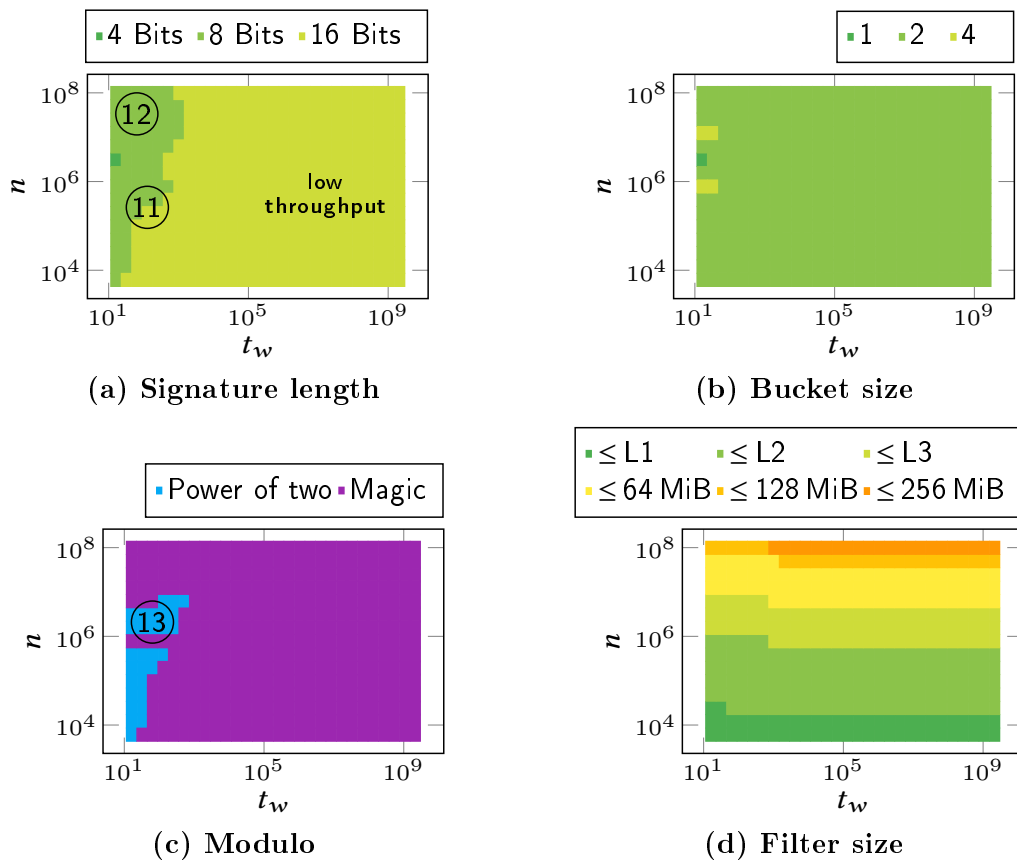


Figure A.13: Skyline of configurations of the best performing Cuckoo filters (on SKX).

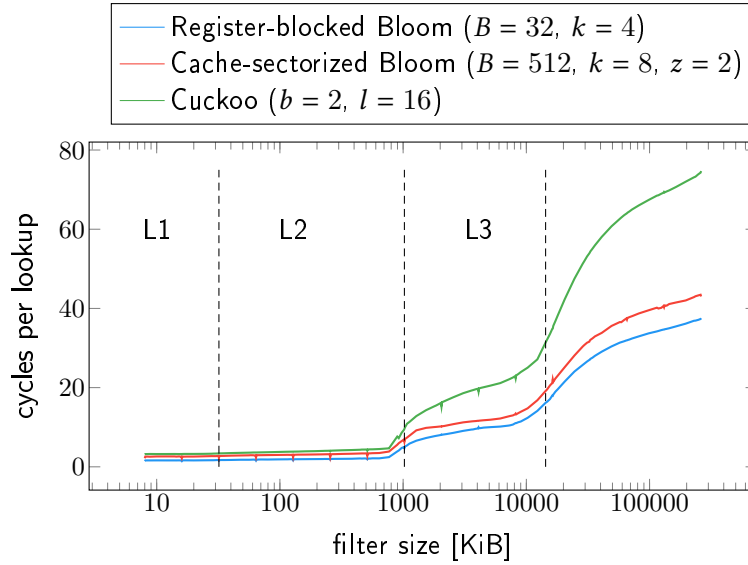


Figure A.14: Lookup time for varying filter sizes (on SKX).

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_w s 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 A.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 A.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 arbi-

trarily large t_w s, 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 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 A.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 f s are mostly observed in the area where filtering is not beneficial (top left corner).

Best performing Bloom filter variants. In Figure A.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 A.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 A.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 A.12b-A.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 A.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 .

As mentioned earlier, cache-sectorization covers the largest space in high-throughput scenarios (see Figure A.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 A.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 A.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 A.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 [63] (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 A.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 A.13c (13)), which is due to the higher costs involved with magic modulo, as described in Section A.5.2.

In general, we observed similar memory consumptions among the two filters under test. The claim that Cuckoo filters have better space efficiency [63] no longer holds when performance optimality is the objective.

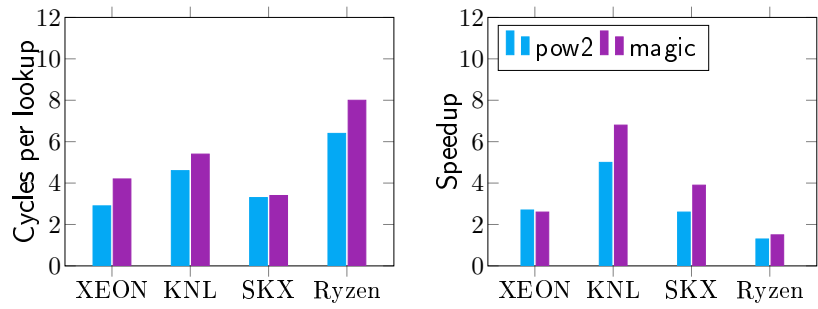
A.6.1 SIMD Optimizations

We present the performance impact of our SIMD optimizations. Figure A.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.

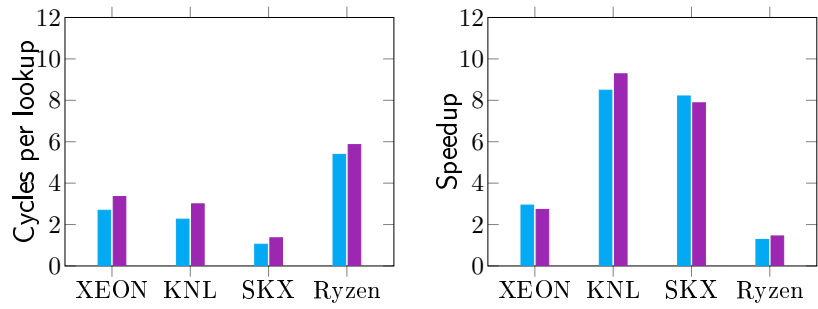
SIMD optimizations offer speedups of up to $10\times$ and therefore make filtering applicable for a larger spectrum in high-throughput scenarios (small t_{ws}). On AVX2 platforms, the (bare L1) performance of Cuckoo filters is very similar to register-blocked Bloom filters. If the filter size exceeds L2, blocked Bloom filters perform better with regard to CPU cycles per lookup due to better memory bandwidth efficiency (see Figure A.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-512BW (**Byte Word**) 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).

A.7 Related Work

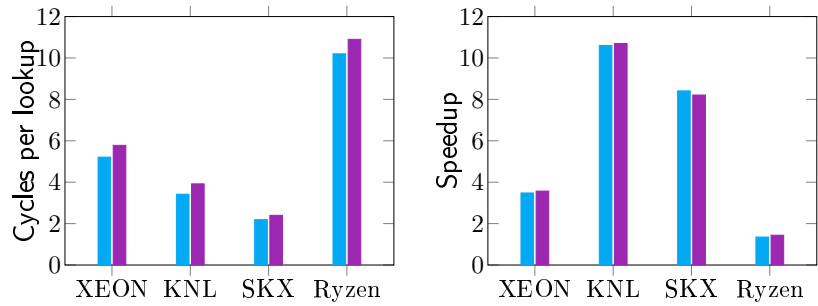
The survey [32] describes many of the application areas of Bloom filters [23]: databases, dictionaries, (P2P) networking and routing. In databases, log-structured merge-trees (LSM) have become important in write-optimized (cloud or cluster) storage, splitting up a structure into multiple layers that 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 [53] observes that different layers need differently tuned Bloom filters. That paper navigates correlated parameter spaces in a data



(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 A.15: Performance of our SIMD-optimized filter implementations.

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 [23]. Scalable Bloom filters [13] 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 [47] and counting Bloom filters [137, 28] can represent bags (duplicate keys) rather than sets. The Bloomier filter [42] can associate a value (rather than a bit) with a key. Retouched Bloom filters [60] 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. [144] 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 [142]: 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 [63] 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 [30]. 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 numbers⁵ on similar hardware (Ryzen Threadripper 1950X), showing that our implementation provides the same query performance with large filters ($\approx 200\text{MB}$); 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 [20] and TinySet [61] 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 [153] or MinMax indexes using Small Materialized Aggregates (SMAs) [118], (ii) skipping scan ranges within blocks, e.g., Positional SMAs [96] and Adaptive Range Filters [12], or (iii) by skipping (parts of) individual tuples, e.g., BitWeaving [110, 143] and ByteSlice [66]. The more recent Column Sketches [77] 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 [77]). 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.

A.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.

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

A.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.

B Make the Most out of Your SIMD Investments: Counter Control Flow Divergence in Compiled Query Pipelines

Harald Lang¹, Linnea Passing¹, Andreas Kipf¹, Peter Boncz², Thomas Neumann¹, Alfons Kemper¹

¹ Technical University of Munich

² Centrum Wiskunde & Informatica

Appeared in *The VLDB Journal*, 2019. <https://doi.org/10.1007/s00778-019-00547-y>

The content of this section is identical to the original publication. Only the format and the numbering have been adjusted.

In accordance with the TUM regulations for the award of doctoral degrees (TUM Promotionsordnung, 2014), a summary of the publication is included in the first part of this thesis. Please refer to Section 3.4. Furthermore, the printed version is included in the Appendix on page 187 ff.

The contributions of the thesis author to this publication are: developing the algorithms and strategies for countering control flow divergence, the implementation, the evaluation, and authoring of substantial parts of the paper.

Abstract

Increasing single instruction multiple data (SIMD) capabilities in modern hardware allows for the compilation of data-parallel query pipelines. This means GPU-alike challenges arise: *control flow divergence* causes the underutilization of vector-processing units. In this paper, we present efficient algorithms for the AVX-512 architecture to address this issue. These algorithms allow for the fine-grained assignment of new tuples to idle SIMD lanes. Furthermore, we present strategies for their integration with compiled query pipelines so that tuples are never evicted from registers. We evaluate our approach with three query types: *(i)* a table scan query based on TPC-H Query 1, that performs up to 34% faster when addressing underutilization, *(ii)* a hashjoin query, where we observe up to 25% higher performance, and *(iii)* an approximate geospatial join query, which shows performance improvements of up to 30%.

B.1 Introduction

Integrating SIMD processing with database systems has been studied for more than a decade [191]. Several operations, such as selection [96, 143], join [88, 18, 19, 166], partitioning [140], sorting [45], CSV parsing [119], regular expression matching [157], and (de-)compression [189, 143, 105] have been accelerated using the SIMD capabilities of the x86 architectures. In more recent iterations of hardware evolution, SIMD instruction sets have become even more popular in the field of database systems. Wider registers, higher degrees of data-parallelism, and comprehensive support for integer data have increased the interest in SIMD and led to the development of many novel algorithms.

SIMD is mostly used in interpreting database systems [86] that use the *column-at-a-time* or *vector-at-a-time* execution model [26]. Compiling database systems [86] like HyPer [85] barely use it due to their data-centric *tuple-at-a-time* execution model [128]. In such systems, therefore, SIMD is primarily used in scan operators [96] and in string processing [119].

With the increasing vector-processing capabilities for database workloads in modern hardware, especially with the advent of the AVX-512 instruction set, query compilers can now vectorize entire query execution pipelines and benefit from the high degree of data-parallelism [74]. With AVX-512, the width of vector registers increased to 512 bit, allowing for the processing of an entire cache line in a single instruction. Depending on the bit-width of the attribute values, data elements from up to 64 tuples can be packed into a single register.

Vectorizing entire query pipelines raises new challenges. One such challenge is keeping all SIMD lanes busy during query evaluation, as not all in-flight tuples follow the same control flow. For instance, some might be disqualified during predicate evaluation, while others may not find a join partner later on and get discarded. Whenever a tuple gets disqualified, the corresponding SIMD lane is affected. A scalar (non-vectorized) pipeline would take a branch and thereby return the control flow to a tuple producing operator to fetch the next tuple. In a vectorized pipeline, this is only possible iff **all in-flight tuples have been disqualified**. If this is not the case, the query of the subsequent operator still needs to be executed. Ignoring SIMD lanes containing disqualified tuples is the easiest way to deal with this situation, as it does not introduce branching logic and only requires a small amount of bookkeeping. A small bitmap is sufficient to keep track of disqualified elements. The bitmap is used at the pipeline sink, when the (intermediate) result is materialized, making sure that disqualified elements are not written to the query result set. The downside of this approach is, that within the pipeline, all instructions are performed on all SIMD lanes regardless of whether the SIMD lane contains an active or an inactive element. All operations that are performed on inactive elements can be considered overhead, as they do not contribute to the result. In other words, not all SIMD lanes perform useful work and if lanes contain disqualified elements, the vector-processing units (VPUs) can be considered **underutilized**. Therefore, efficient algorithms are required to counter the un-

derutilization of vector-processing units. In [115], this issue was addressed by introducing (memory) materialization points immediately after each vectorized operator. However, with respect to the more strict definition of pipeline breakers given in [128], materialization points can be considered as pipeline breakers because tuples are evicted from registers to slower (cache) memory. In this work, we present alternative algorithms and strategies that do not break pipelines. Further, our approach can be applied at the intra-operator level as well as at operator boundaries.

The remainder of this paper is organized as follows. In Section B.2, we briefly describe the relevant AVX-512 instructions that we use in our algorithms. The potential performance degradation caused by underutilization in holistically vectorized pipelines is discussed in Section B.3. In Section B.4, we introduce efficient algorithms to counter underutilization, and in Section B.5, we present strategies for integrating these algorithms with compiled query pipelines. The experimental evaluation of the proposed algorithms using a table scan query, a hashjoin query, and an approximate geospatial join query is given in Section C.4. The experimental results are summarized and discussed in Section B.7, followed by our conclusions in Section B.8.

B.2 Background

In this section, we briefly describe the key features of the AVX-512 instruction set that we use in our algorithms in Section B.4. In particular, we cover the basics of vector predication as well as the *permute* and the *compress/expand* instructions.

Mask instructions: Almost all AVX-512 instructions support *predication*. These instructions allow to perform a vector operation only on those vector components (or lanes) specified by a given bitmask, where the i^{th} bit in the bitmask corresponds to the i^{th} lane. For example, an add instruction in its simplest form requires two (vector) operands and a destination register that receives the result. In AVX-512, the instruction exists in two additional variants:

1. **Merge masking:** The instruction takes two additional arguments, a *mask* and a source register, for example, `dst = mask_add(src, mask, a, b)`. The addition is performed on the vector components in `a` and `b` specified by the `mask`. The remaining elements, where the mask bits are 0, are copied from `src` to `dst` at their corresponding positions.
2. **Zero masking:** The functionality is basically the same as that of merge masking, but instead of specifying an additional source vector, all elements in `dst` are set to zero if the corresponding bit in the mask is not set. Zero masking is therefore (logically) equivalent to merge masking with `src` set to zero: `maskz_add(mask, a, b) ≡ mask_add(0, mask, a, b)`. Thus, zero masking is a special case of merge masking.

Masked instructions can be used to prevent individual vector components from being altered, e.g., $x = \text{mask_add}(x, \text{mask}, a, b)$.

Typically, masks are created using comparison instructions and stored in special mask registers, which is a significant improvement over earlier SIMD instruction sets, in which these masks were stored in 256-bit vector registers.

Permute: The permute instruction shuffles elements within a vector register according to a given index vector:

$$\underbrace{[d, a, d, b]}_{\text{result vector}} = \text{permute}(\underbrace{[3, 0, 3, 1]}_{\text{index vector}}, \underbrace{[a, b, c, d]}_{\text{input vector}}).$$

It is noteworthy, that the permute instruction has already been available in earlier instruction sets. But due to the doubled register size, twice as many elements can now be processed at once. Further, in our application, we achieve a four times higher throughput compared to the earlier AVX2 instruction set. The reason is, that assigning new elements to idle SIMD lanes is basically a *merge* operation of the content of two vector registers. In combination with merge masking, this operation can be performed using a single instruction, whereas with AVX2, two instructions need to be issued, (*i*) a permute to move the elements into their desired SIMD lanes and (*ii*) a blend to select the desired lanes from two source registers and merge them into a destination register.

Compress / Expand: Typically, before a permute instruction can be issued, an algorithm needs to determine the aforementioned index vector, which used to be a tedious task that often induced significant overheads, such as additional accesses into predefined lookup tables [9, 96, 115, 142]. The key instructions introduced with AVX-512 to efficiently solve these types of problems, are called compress and expand. Compress stores the active elements (indicated by a bit-mask) contiguously into a target register, and expand stores the contiguous elements of an input at certain positions (specified by a *write mask*) in a target register:

$$\begin{aligned} & \downarrow \downarrow \downarrow \downarrow \downarrow \downarrow \downarrow \downarrow \\ & [a, d, 0, 0] = \text{compress}(1001, [a, b, c, d]) \\ & \downarrow \downarrow \downarrow \downarrow \downarrow \downarrow \downarrow \downarrow \\ & [0, a, 0, b] = \text{expand}(0101, [a, b, c, d]) \\ & \times \times \times \times \times \times \times \times \end{aligned}$$

Both instructions come in two flavors: (*i*) read/write from/to memory and (*ii*) directly operate on registers.

Our algorithms in general require both, permute and compress/expand instructions. There is only one special case, where a permute suffices, which we describe in the later Section B.4.

B.3 Vectorized Pipelines

As mentioned in the introduction, the major difference between a scalar (i.e., non-vectorized) pipeline, as pioneered by HyPer [85], and a vectorized pipeline is that in the latter, multiple tuples are pushed through the pipeline at once. This impacts the *control flow* within the query pipeline. In a scalar pipeline, whenever the control flow reaches any operator, it is guaranteed that there is *exactly one* tuple to process (tuple-at-a-time). By contrast, in a vectorized pipeline, there are several tuples to process. However, because the control flow is not necessarily the same for all tuples, some SIMD lanes may become inactive when a conditional branch is taken. Such a branch is only taken if *at least one* element satisfies the branch condition. This implies that a vector of length n may contain up to $n - 1$ *inactive* elements, as depicted in Figure B.1. The figure shows a simplified control flow graph (CFG) for an example query pipeline that consists of a table scan, a selection, and a join operator. The directed edges represent the branching logic. For instance, the *no match* edges are taken if a tuple is disqualified in the selection or the join operator. The *index traversal* (self-)edge is taken when an index lookup is performed. For instance, a hash table or tree lookup might require one to follow multiple bucket pointers until a join partner for the current tuples is found. The right-hand side of Figure B.1 visualizes the SIMD lane utilization over time. Initially, in the scan operator, all SIMD lanes are active (green color). Inside the select or join operator, elements are disqualified (marked with a X), but the *no match* branch is not taken, because some elements are still active. Lane 4 represents a different situation, where an SIMD lane becomes **temporarily inactive**. In that example, the element in lane 4 finds its join partner in the very first iteration of the index lookup. However, lanes 1 and 6 need three iterations until the index lookup terminates. During that time, lane 4 is idle and afterwards, it becomes active again.

In general, all conditional branches within the query pipeline are potential sources of control flow *divergence* and, therefore, a source of the underutilization of VPU, whereas, disqualified elements cause underutilization in all subsequent operators and lookups in index structures cause *intra-operator* underutilization. The latter is an inherent problem when traversing irregular pointer-based data structures in an SIMD fashion [146]. To avoid underutilization through divergence, we need to dynamically assign new tuples to idle SIMD lanes, possibly at multiple “points of divergence” within the query pipeline. We refer to this process as pipeline *refill*.

B.4 Refill Algorithms

In this section, we present our refill algorithms for AVX-512, which we later integrate into compiled query pipelines (cf., Section B.5). These algorithms essentially copy new elements to *desired positions* in a destination register.

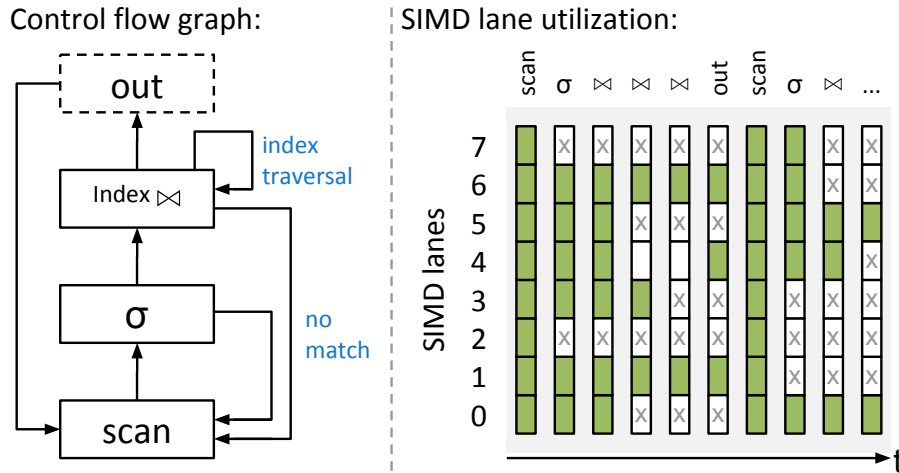


Figure B.1: During query processing, individual SIMD lanes may (temporarily) become inactive due to different control flows. The resulting underutilization of vector-processing units causes performance degradations. We propose efficient algorithms and strategies to fill these gaps.

In this context, these desired positions are the lanes that contain inactive elements. The active lanes are identified by a small bitmask (or simply *mask*), where the i^{th} bit corresponds to the i^{th} SIMD lane. An SIMD lane is active if the corresponding bit is set, and vice versa. Thus, the bitwise complement of the given mask refers to the inactive lanes and, therefore, to the write positions of new elements. We distinguish between two cases as follows: (i) where new elements are copied from a source memory address and (ii) where elements are already in vector registers.

In the following, we frequently use various constant values, which we write in capital letters. For instance, **ZERO** and **ALL** refer to constant values where all bits are zero or one, respectively. The vector constant **SEQUENCE** contains an integer sequence starting at 0 and **LANE_CNT** refers to the number of SIMD lanes.

B.4.1 Memory to Register

Refilling from memory typically occurs in the table scan operator, where contiguous elements are loaded from memory (assuming a columnar storage layout). AVX-512 offers the convenient **expand load** instruction that loads contiguous values from memory directly into the desired SIMD lanes (cf., Figure B.2). One mask instruction (**bitwise not**) is required to determine the *write mask* and one vector instruction (**expand load**) to execute the actual load. Overall, the simple case of refilling from memory is supported by AVX-512 directly out of the box.

The table scan operator typically produces an additional output vector con-

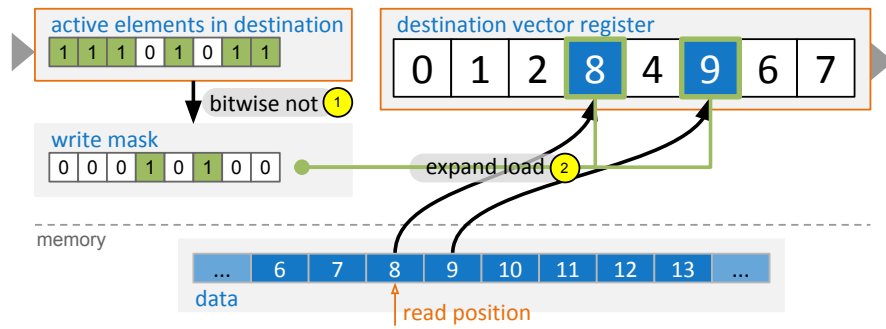


Figure B.2: Refilling empty SIMD lanes from memory using the AVX-512 `expand load` instruction.

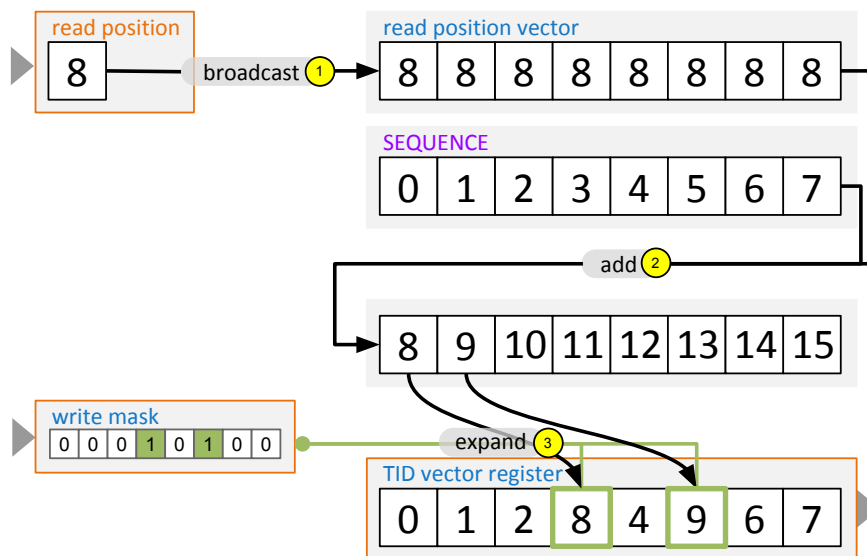


Figure B.3: TIDs are derived from the current read position and assigned to a TID vector register.

taining the tuple identifiers (TIDs) of the newly loaded attribute values. The TIDs are derived from the current read position and are used, for example, to (lazily) load attribute values of a different column later on or to reconstruct the tuple order. Figure B.3 illustrates, how the content of the TID vector register is updated, using the read position and write mask from Figure B.2.

B.4.2 Register to Register

Moving data between vector registers is more involved. In the most general case, we have a source and a destination register that contain both active and inactive elements at *random* positions. The goal is to move as many elements as possible from the source to the destination. This can be achieved using a single masked permute instruction. But before the permutation instruction

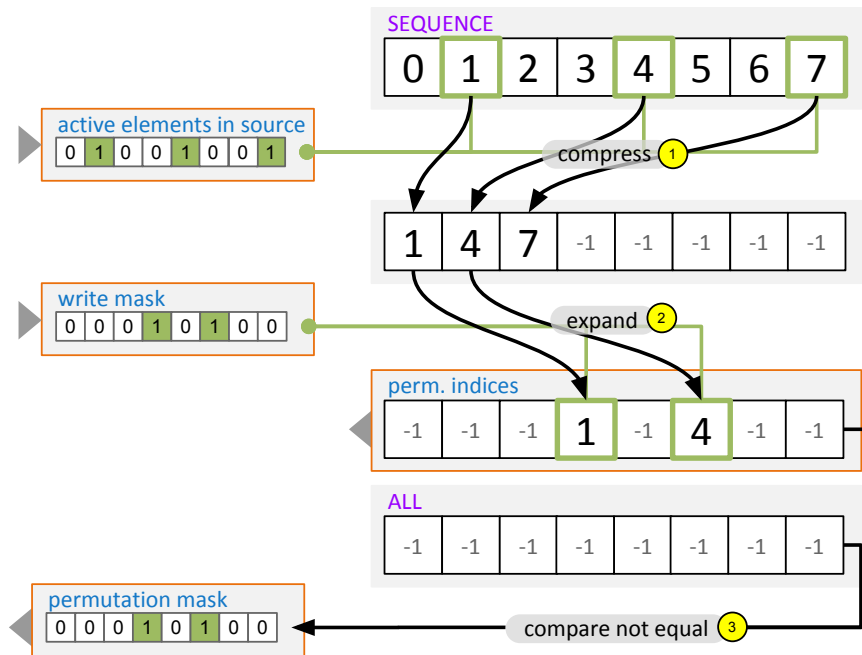


Figure B.4: Computation of the permutation indices and the permutation mask based on positions of the active elements in the source register and the inactive elements in the destination register.

can be issued, the *permutation indices* need to be computed, based on the positions of active elements in the source and the destination vector registers. This is illustrated in Figure B.4, where, as in the previous examples, the *write mask* refers to the inactive lanes in the destination register. In total, three vector instructions are required to compute the permutation indices and an additional permutation mask. The latter is required in case the number of active elements in the source is smaller than the number of empty lanes in the destination vector. In that case, the destination register still contains some inactive lanes, and the corresponding bitmask must be updated accordingly.

Once the permutation indices are computed, elements can be moved between registers accordingly. Notably, the algorithm can be adapted to move elements directly instead of computing the permutation indices first. However, if elements need to be moved between more than one source/destination vector pair, the additional cost of computing the permutation amortizes immediately with the second pair. In practice, the permutation is typically applied multiple times, for example, when multiple attributes are pushed through the pipeline or to keep track of the TIDs.

In the general case, there are no guarantees about the number of (active) elements nor their positions within the vector register. For example, the elements in the source may not be entirely consumed or the destination vector

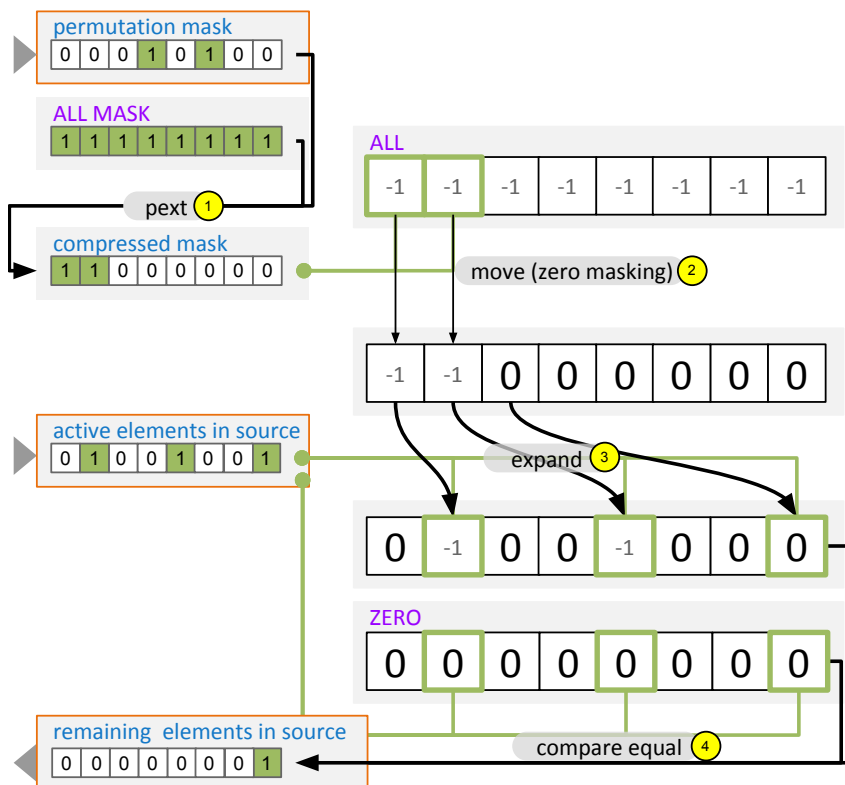


Figure B.5: If not all elements could be moved from the source to the destination register, the source mask needs to be updated accordingly.

may still contain inactive elements. Thus, it is necessary to update source and destination masks accordingly. Updating the destination mask is straightforward by using a bitwise or with the previously computed permutation mask. Updating the source mask is less obvious as illustrated in Figure B.5. As the figure shows, updating the source mask is as expensive as preparing the permutation. However, if it is guaranteed that all source elements fit into the destination vector, this phase of the algorithm can be skipped altogether. Listing B.1 shows the full algorithm formulated in C++.

In summary, a typical refill looks as follows:

```
[...]
// Prepare the refill.
fill_rr r(src_mask, dst_mask);
// Copy elements from src to dst.
r.apply(src_tid, dst_tid);
r.apply(src_attr_a, dst_attr_a);
r.apply(src_attr_b, dst_attr_b);
r.apply(..., ...);
// Update the destination mask,
r.update_dst_mask(dst_mask);
// and optionally the source mask.
r.update_src_mask(src_mask);
[...]
```

B.4.3 Variants

Depending on the position of the elements, cheaper algorithms can be used. Especially when the vectors are in a *compressed state*, meaning that the active elements are stored contiguously, it is considerably cheaper to prepare the permutation (compare Listing B.1 and B.2). Compared to the first algorithm, which can permute elements from/to random positions, the second algorithm does not need any bit masks to refer to the active lanes. Instead, it is sufficient to pass in the number of active elements. In Listing B.2, we refer to these numbers as `src_cnt` and `dst_cnt`. Based on these, the permutation indices, as well as the permutation mask, can be computed without any cross-lane operations, such as `compress/expand`. A noteworthy property of the second SIMD algorithm is that the source vector remains in a compressed state even if not all elements fit into the destination vector.

These two foundational SIMD algorithms cover the extreme cases where *(i)* active elements are stored at random positions and *(ii)* active elements are stored contiguously. Based on these cases, the algorithms can easily be adapted so that only one vector needs to be compressed, which is useful when vector registers are used as tiny buffers because those should always be in a compressed state to achieve the best performance. In total, there are four different algorithms. Each algorithm has two different flavors: *(i)* where all

Listing B.1: Generic refill algorithm

```
struct fill_rr {  
  
    __mmask8 permutation_mask;  
    __m512i permutation_idx;   
  
    // Prepare the permutation.  
    fill_rr(const __mmask8 src_mask,  
            const __mmask8 dst_mask) {  
        __m512i src_idx = _mm512_mask_compress_epi64(  
            ALL, src_mask, SEQUENCE);  
        __mmask8 write_mask = _mm512_knot(dst_mask);  
        permutation_idx = _mm512_mask_expand_epi64(  
            ALL, write_mask, src_idx);  
        permutation_mask = _mm512_mask_cmpneq_epu64_mask(  
            write_mask, permutation_idx, ALL);  
    }  
  
    // Move elements from 'src' to 'dst'.  
    void apply(const __m512i src, __m512i& dst) const {  
        dst = _mm512_mask_permutexvar_epi64(  
            dst, permutation_mask, permutation_idx, src);  
    }  
  
    void update_src_mask(__mmask8& src_mask) const {  
        __mmask8 compressed_mask = _pext_u32(~0u, permutation_mask);  
        __m512i a = _mm512_maskz_mov_epi64(compressed_mask, ALL);  
        __m512i b = _mm512_maskz_expand_epi64(src_mask, a);  
        src_mask = _mm512_mask_cmpeq_epu64_mask(src_mask, b, ZERO);  
    }  
  
    void update_dst_mask(__mmask8& dst_mask) const {  
        dst_mask = _mm512_kor(dst_mask, permutation_mask);  
    }  
  
};
```

Listing B.2: Refill algorithm for compressed vectors

```
struct fill_cc {  
  
    __mmask8 permutation_mask;  
    __m512i permutation_idx;   
    uint32_t cnt;  
  
    // Prepare the permutation.  
    fill_cc(const uint32_t src_cnt,  
            const uint32_t dst_cnt) {  
        const auto src_empty_cnt = LANE_CNT - src_cnt;  
        const auto dst_empty_cnt = LANE_CNT - dst_cnt;  
        // Determine the number of elements to be moved.  
        cnt = std::min(src_cnt, dst_empty_cnt);  
        bool all_fit = (dst_empty_cnt >= src_cnt);  
        auto d = all_fit ? dst_cnt : src_empty_cnt;  
        const __m512i d_vec = _mm512_set1_epi64(d);  
        // Note: No compress/expand instructions required  
        permutation_idx = _mm512_sub_epi64(SEQUENCE, d_vec);  
        permutation_mask = ((1u << cnt) - 1) << dst_cnt;  
    }  
  
    // Move elements from 'src' to 'dst'.  
    void apply(const __m512i src, __m512i& dst) const {  
        dst = _mm512_mask_permutexvar_epi64(  
            dst, permutation_mask, permutation_idx, src);  
    }  
  
    void update_src_cnt(uint32_t& src_cnt) const {  
        src_cnt -= cnt;  
    }  
  
    void update_dst_cnt(uint32_t& dst_cnt) const {  
        dst_cnt += cnt;  
    }  
};
```

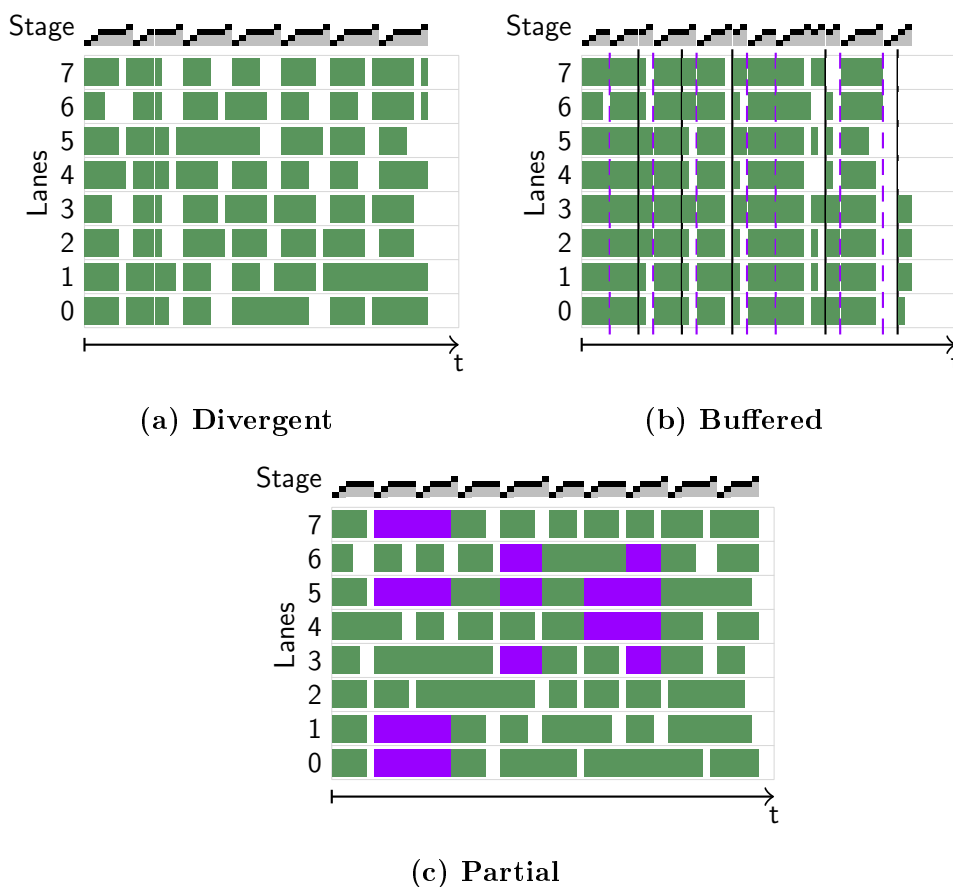


Figure B.6: SIMD lane utilization using different strategies. – In (a), no refilling is performed to visualize the divergence. – (b) uses the consume everything strategy, which performs refills when the utilization falls below 75%. The dashed purple lines indicate a write to buffer registers, black lines a read. – (c) shows a partial consume throughout the entire pipeline with the minimum required utilization set to 50%. Lanes colored in purple are protected.

elements from the source register are guaranteed to fit into the destination register or (ii) where not all elements can be moved and therefore elements remain in the source register. We do not show all variants here, but have released the C++ source code¹ under the BSD license.

B.5 Refill Strategies

We discuss the integration of these refill algorithms in data-centric *compiled* query pipelines. Such pipelines turn a query operator pipeline into a for-loop, and the code generated by the various operators is nested bottom-up in the body of such a loop [128]. Relational operators in this model generate code in two methods, namely, `consume()` and `produce()`, which are called in a depth-first traversal of the query tree: `produce()` code is generated before generating the code for the children, and `consume()` afterwards.

The main idea of data-centric execution with SIMD is to insert checks for each operator that control the number of tuples in play, i.e., if-statements nesting the rest of the body. Such an if-statement ensures that its body only gets executed if the SIMD registers are sufficiently full. Generally speaking, operator code processes input SIMD data computed by the outer operator and *refills* the registers it works with and the ones it outputs.

We identify two *base strategies* for applying this refilling.

B.5.1 Consume Everything

The consume everything strategy allocates additional vector registers that are used to *buffer* tuples. In the case of underutilization, the operator *defers* the processing of these tuples. This means the body will not be executed in this iteration (if-condition not satisfied) but instead (else) the active tuples will be moved to these buffer registers. It uses the refill algorithms from the previous section both to move data to the buffer and to emit buffered tuples into the unused lanes in a subsequent iteration. Listing B.3 shows the code skeleton as it would be generated by such a *buffering* operator. The `THRESHOLD` parameter specifies when a refill is triggered during query execution. Depending on the situation, the costs for refilling might not amortize if only a few lanes contain inactive elements. But if the remaining pipeline is very expensive, setting the threshold to the number of SIMD lanes could be the best option. The important thing to note here is that all SIMD lanes are empty when the control flow returns to the previous operator, thus we call it *consume everything*.

Compared to a scalar pipeline, this strategy only requires a minor change to the push model: handling a special case when the pipeline execution is about to terminate, flushing the buffer(s). The essence is that buffering only takes place in SIMD registers and it specifically does not cause extra in-memory materialization.

Figures B.6a and B.6b illustrate the effects of applying a refill strategy to a query pipeline by visualizing the SIMD lane utilization over time. The structure of the query is similar to the one shown in Figure B.1 and consists of a scan, a selection, a join, and a sink to where the output is written. The *stage* indicator on top of the plot refers to the node in the control flow graph in Figure B.1. In Figure B.6a, the query is executed without divergence handling,

¹Source code: https://github.com/harald-lang/simd_divergence

Listing B.3: Code skeleton of a buffering operator.

```
[...]
auto active_lane_cnt = popcount(mask);
if (active_lane_cnt + buffer_cnt < THRESHOLD
    && !flush_pipeline) {
    [...] // Buffer the input.
}
else {
    const auto bail_out_threshold =
        flush_pipeline ? 0
            : THRESHOLD;
    while (active_lane_cnt + buffer_cnt > bail_out_threshold) {
        if (active_lane_cnt < THRESHOLD) {
            [...] // Refill lanes with buffered elements.
        }
        //=====//
        // The actual operator code and
        // consume code of subsequent operators.
        [...]
        //=====//
        active_lane_cnt = popcount(mask);
    }
    if (likely(active_lane_cnt != 0)) {
        [...] // Buffer the remaining elements.
    }
}
// All lanes empty (consume everything semantics).
mask = 0;
[...]
```

and the white areas refer to underutilization. Figure B.6b visualizes the same workload with in-register buffering, following consume everything semantics. The purple and black vertical lines indicate that tuples are written to the buffers, or read from the buffer, respectively. Compared to the divergent implementation, the lane utilization has significantly increased, and the overall execution time has reduced. In this example, we require the utilization to be at least 75% (six out of eight lanes need to be active). Underutilization is observed only when the execution is about to finish, which triggers a pipeline flush, where all (potentially) buffered tuples need to be processed regardless of the minimum utilization threshold.

B.5.2 Partial Consume

As the name suggests, the second base strategy no longer expects the `consume()` code to process the entire input. The consume code can decide to defer execution by returning the control flow to the previous operator and leave the active

elements in the vector registers. New tuples are assigned only to inactive lanes by one of the preceding operators, typically a table scan. Naturally, the active lanes, that contain deferred tuples, must not be overwritten or modified by other operators. We refer to these elements (or to their corresponding lanes) as being *protected*. Another way of looking at a protected lane is that the lane is *owned* by a different operator. When an *owning* operator completes the processing of a tuple, it transfers the ownership to the subsequent operator. Alternatively, if the tuple is disqualified, it gives up ownership to allow a tuple producing operator to assign a new tuple to the corresponding lane.

Lane protection requires additional bookkeeping on a per operator basis. Each operator must be able to distinguish between tuples that *(i)* have just arrived, *(ii)* have been protected by the operator itself in an earlier iteration and *(iii)* tuples that have already advanced to later stages in the pipeline. To do so, an operator maintains two masks, one that identifies the lanes that are owned by the current operator and another one that identifies lanes that are owned by a later operator. Listing B.4 shows the structure of such an operator, where `this_stage_mask` and `later_stage_mask` are part of the operator's state and `mask` is used to communicate which lanes contain active elements (regardless of their stage).

Figure B.6c shows how the partial consume strategy affects the lane utilization with the minimum lane utilization threshold set to 50%. The lanes colored in purple are in a protected state. Compared to the divergent implementation, the lane utilization has increased. However, if we take protected lanes into account and consider them as idle, the overall utilization decreases. Thus, the example workload, used in Figure B.6, reveals an important drawback. If the lanes become protected in later stages of the pipeline, these lanes can cause significant underutilization in the preceding operators. We discuss this issue, among other things, in the following section.

B.5.3 Discussion and Implications

The two strategies are not mutually exclusive. Within a single pipeline, both strategies can be applied to individual operators as long as buffering operators are aware of protected lanes (*mixed* strategy). Moreover, the query compiler might decide to *not* apply any refill strategy to certain operators. Especially, when a sequence of operators is quite cheap, divergence might be acceptable as long as the costs for refill operations are not amortized. Naturally, this is a physical query optimization problem that we will leave for future work. Nevertheless, we briefly discuss the advantages and disadvantages, as this is the first work in which we present the basic principles of vector-processing in compiled query pipelines.

As mentioned above, *consume everything* requires additional registers, which increases the register pressure and may lead to spilling. *partial consume* allocates additional registers as well, but these are restricted to (smaller) mask registers. Therefore, it is unlikely to be affected by (potential) performance

Listing B.4: Code skeleton of a partial consume operator.

```
[...]
auto active_lane_cnt = popcount(mask);
if (active_lane_cnt < THRESHOLD && !flush_pipeline){
    // Take ownership of newly arrived elements.
    this_stage_mask = mask ^ later_stage_mask;
}
else {
    //=====//
    // The actual operator code and
    // consume code of subsequent operators.
    [...]
    // The later_stage_mask is set by the
    // consumer.
    //=====//
}
// Protect lanes in the preceding operator.
mask = this_stage_mask | later_stage_mask;
[...]
```

degradation due to spilling.

The second major difference lies in the cost of refilling empty lanes. In a pipeline that follows the partial consume strategy, the very first operator, that is, the pipeline source, is responsible for refilling empty lanes. If other operators experience underutilization, they return the control flow to the previous operator while retaining ownership of the active lanes. This cascades downward until the source operator is reached, as shown in Figure B.6c. All operators between the pipeline source and the operator that returned the control flow may be subject to underutilization because all lanes in later stages are protected. The costs of refilling, therefore, depend on the length of the pipeline and the costs of the preceding operators. In general, the costs increase in the later stages. Nevertheless, partial consume can improve query performance if it is applied only to the very first operators. By contrast, the refilling costs of buffering operators do not depend on the pipeline length. Instead, the crucial factor governing these costs is the number of required buffer registers. The greater the number of buffers, the greater the number of `permute` instructions that need to be executed, whereas the number of required buffers depends on (i) the number of attributes passed along the pipeline and optionally on (ii) the number of registers required to save the internal state of the operator (e.g., a pointer to the current tree node).

	Intel Knights Landing (KNL)	Intel Skylake-X (SKX)
model	Phi 7210	i9-7900X
cores (SMT)	64 ($\times 4$)	10 ($\times 2$)
SIMD [bit]	2 \times 512	2 \times 512
max. clock rate [GHz]	1.5	4.5
L1 cache	64 KiB	32 KiB
L2 cache	1 MiB	1 MiB
L3 cache	-	14 MiB

Table B.1: Hardware platforms

B.6 Evaluation

We evaluate our approach with two major sources of control flow divergence, (*i*) predicate evaluation as part of a table scan and (*ii*) a hash join. Additionally, we experiment with a more complex operator, an approximate geospatial join. The experiments were conducted on an Intel Skylake-X (SKX) and an Intel Knights Landing (KNL) processor (cf., Table B.1). The experiments were implemented in C++ and compiled with GCC 5.4.0 at optimization level three (-O3) and the target architecture set to knl. If not stated otherwise, we ran the experiments in parallel using two threads per core². We dispatched the work in batches to the individual threads using batch sizes between 2^{16} and 2^{20} tuples. On the KNL platform, we placed the data in high-bandwidth memory (HBM), otherwise the experiments would have been dominated by memory stalls. To measure the throughputs, we let each experiment run for at least three seconds, possibly consuming the input data multiple times.

B.6.1 Table Scan

To evaluate the effects of divergence handling in table scans, we integrate our refill algorithms into the AVX-512 implementation of TPC-H Query 1 of Gubner et al. [74]. Additionally, we implemented and integrated the *materialization* approach as proposed by Menon et al. in [115].

From a high-level perspective, TPC-H Query 1 (or short Q1) is a structurally simple query that operates on a single fact table (`lineitem`) with a single scan predicate. It involves several fixed-point arithmetic operations in the aggregation based on the group by clause. In total, five additional attributes are accessed to compute eight aggregated values per group. Almost all tuples sur-

²Please note that throughout our (multi-threaded) experiments, we did not observe any performance penalties through down clocking. Both processors KNL and SKX run stable at 1.4GHz and 4.0GHz, respectively.

vive the selection (i.e., selectivity ≈ 0.98). Therefore, in its original form, Q1 does not suffer from control-flow divergence. To simulate control-flow divergence and the resulting underutilization of SIMD lanes, we vary the selectivity of the scan predicate on the `shipdate` attribute.

We evaluate and compare a scalar³ (non-SIMD) implementation with four AVX-512 implementations:

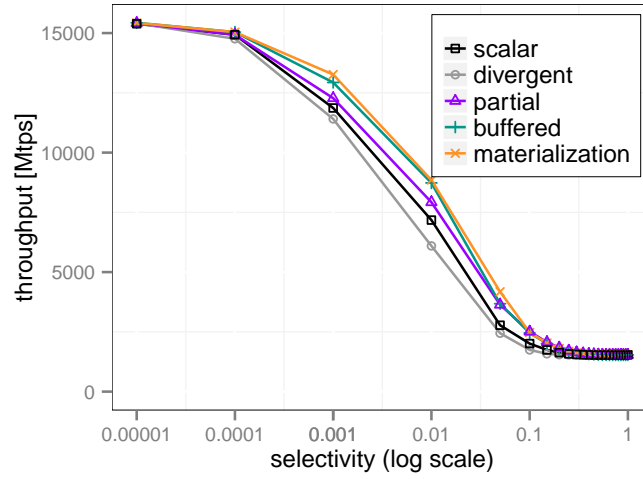
Divergent: The divergent implementation refers to the implementation published by the authors of [74], with a minor modification. In the original version, all tuples are pushed through the query pipeline and disqualified elements are ignored in the final aggregation by setting the lane bitmask accordingly. For our experiments, we introduced a branch behind the predicate evaluation code, which allows to return the control flow to the scan operator iff all SIMD lanes contain disqualified elements. In the case of Q1, the predicate is evaluated on 16 elements in parallel.

Partial / Buffered: The *partial* and *buffered* implementations make use of our refill algorithms. A major difference to the *divergent* implementation is that it can no longer make use of aligned SIMD loads. Instead, it relies on the `gather` instruction to load subsequent attribute values. The select operator therefore produces a tuple identifier (TID) list that identifies the qualifying tuples. The subsequent operators use the TIDs to compute the offset from where to load the additional attributes. – Both implementations are parameterized with the *minimum lane utilization threshold*, which limits the degree of underutilization.

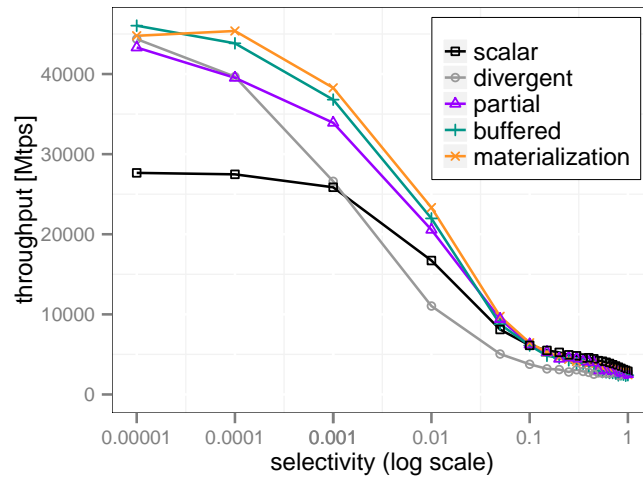
Materialization: The *materialization* implementation makes use of small (memory) buffers to consecutively store the output. Similarly to our approach, the select operator produces a TID list. The code of the subsequent operator(s) is executed when the buffer is (almost) full. The buffered TID list is then consumed (scanned) similarly to a table scan in the subsequent operator. Notably, the output contains only TIDs that belong to qualifying tuples, which is in contrast to our approach, where SIMD lanes may contain non-qualifying tuples, depending on the chosen threshold.

Figure B.7a shows the performance results for varying selectivities (between 0.00001 and 1.0) on SKX. In the extreme cases, all implementations perform

³Scalar refers to an implementation which does not use any SIMD instructions. We verified, that the compiler did not auto-vectorize the query pipelines.



(a) SKX



(b) KNL

Figure B.7: Performance of TPC-H Q1 with varying selectivities.

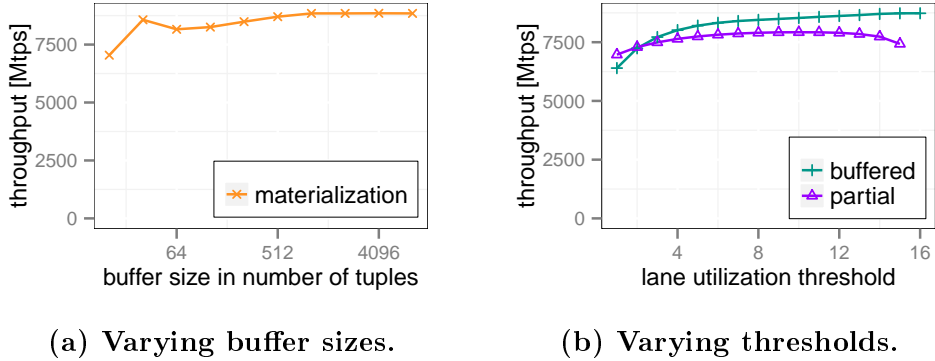


Figure B.8: Performance of TPC-H Q1 performance on SKX when varying algorithm parameters.

similarly. Interestingly, this includes the scalar implementation, which indicates that the SKX processor performs extremely well with respect to IPC, branch prediction, and out of order execution. With intermediate selectivities, divergence handling can make a significant difference. For instance, with $sel = 0.01$ the difference between the divergent and materialization implementation is 2.6 billion tuples per second (1.5 billion over scalar). The graph also shows that the materialization dominates over almost the entire range. Our approach (buffered) can compete, but is slightly slower in most cases. On KNL (Figure B.7b), we observed similar effects. The most important difference is that the divergent SIMD implementation is significantly slower than the scalar implementation with selectivities larger than 0.0001. Divergence handling extends the range in which SIMD optimizations become beneficial.

For this experiment, we varied the utilization threshold for partial and buffered as well as the buffer size for materialization and we reported only the best performing variant. In the following, we investigate the impact of these parameters. Figure B.8a shows the performance of the materialization approach for varying buffer sizes and a fixed selectivity ($sel = 0.01$). Peak performance for Q1 is achieved with a memory buffer of size 1024 elements or larger.

In Figure B.8b, we vary the SIMD lane utilization threshold for our approaches. The performance of the buffered implementation increases with the threshold. Peak performance is reached when only qualifying tuples pass the select operator (threshold = 16). But the performance only gradually increases for a threshold ≥ 6 . The reason for this behavior is that non-qualifying tuples only cause computational overhead in the remaining pipeline but no memory accesses, which would be significantly more expensive. On the other hand, the partial consume strategy favors a threshold that is approximately half the number of SIMD lanes. If the threshold is too low (left-hand side), many non-qualifying tuples pass the filter, and if it is set too high (right-hand side), the control flow is often returned to the scan code to fetch (a few) more values.

B.6.2 Hashjoin

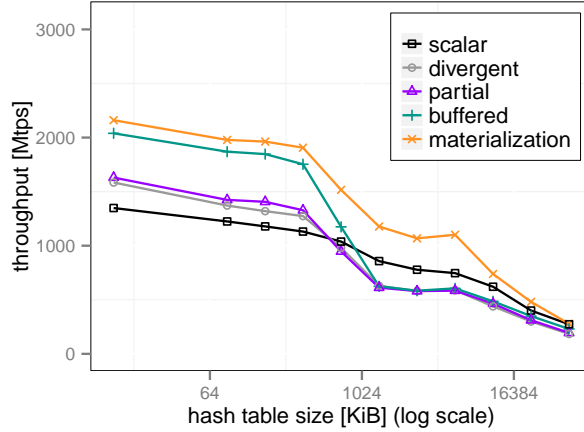
Probing a hash table is a search operation in a pointer-based data structure and therefore a prime source of control-flow divergence. Here, we evaluate the very common foreign-key join of two relations followed by (scalar) aggregations. The primary key relation constitutes the build size in such a way that the join is non-expanding, i.e., for a probe tuple, at most one join partner exists. The two input relations each have two 8-byte integer attributes: a key and a value. The relations are joined using the keys. Afterwards, three aggregations are computed on the join result: the number of tuples, the sum of the values from the left input relation, and the sum of the values from the right input relation. Our hash table implementation stores the first key-value pair per hash bucket in the hash table dictionary. In case of collisions, additional key-value pairs are stored in a linked list per hash bucket.

We evaluate and compare a scalar (non-SIMD) implementation with four AVX-512 implementations:

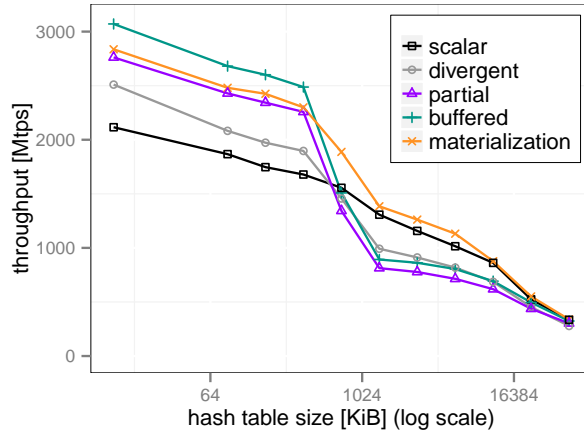
Divergent: This SIMD implementation handles eight tuples in parallel. The lane bitmask is used to keep track of disqualified tuples, such that they can be ignored at the end of the pipeline. As in the table scan evaluation, we add a branch to allow for an early return to the beginning of the pipeline iff all SIMD lanes contain disqualified tuples. We introduce this branch after the first hash table lookup, i.e., it is triggered when all probe tuples fall into empty hash buckets.

Partial / Buffered: These implementations make use of our in-register refill algorithms. In contrast to the table scan discussed in B.6.1, the hash table example uses only few relation attributes. Therefore, instead of loading the additional attributes using `gather`, here all attributes (i.e., key and value) are passed through the pipeline. If the number of active SIMD lanes drops below the *minimum lane utilization threshold*, a refill is performed.

Materialization: Menon et al. [115] propose operator fusion, which introduces buffers between operators to compact the stream of tuples flowing through a pipeline. Here, we introduce an *intra-operator* buffer to further densify the stream of tuples. At the beginning of the pipeline, we load key-value pairs from the probe side input, and compute the hash value and the pointer to the hash table dictionary. We store these key-value pairs and pointers in an input buffer. From this buffer, we then lookup eight pointers in the hash table in parallel, and determine if (i) we found a match, (ii) we need to follow a chain (further), or (iii) there is no match. Unfinished tuples



(a) SKX, 10 threads.



(b) SKX, 20 threads.

Figure B.9: Hashjoin performance when varying build sizes. SKX.

(case *(ii)*) are written back into the input buffer with an updated pointer; matching tuples (case *(i)*) are directly pushed to the subsequent aggregation operator without further buffering, which is not in line with [115], where materialization happens on operator boundaries. We also implemented a “fully” materialized version where the matches are first stored in an output buffer before the aggregation code is executed. However, our experiments have shown that two memory materializations are more expensive.

Figures B.9 and B.10 show the performance results for varying hash table sizes (between 10 KiB and 45 MiB). The hash table size is chosen depending on the build input size. We size the hash table dictionary so that it has the same number of buckets as there are build tuples. Among all evaluated approaches, as well as both platforms, the throughput shrinks with growing hash table

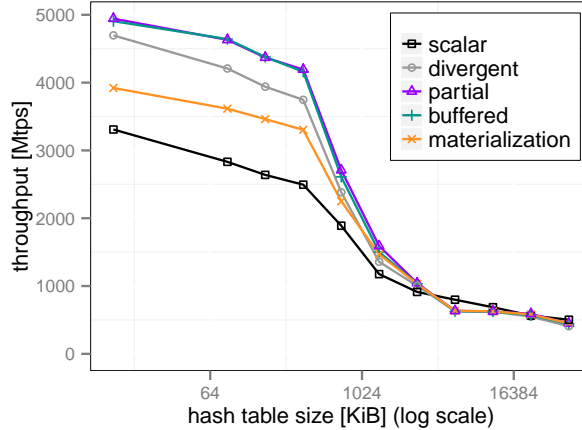
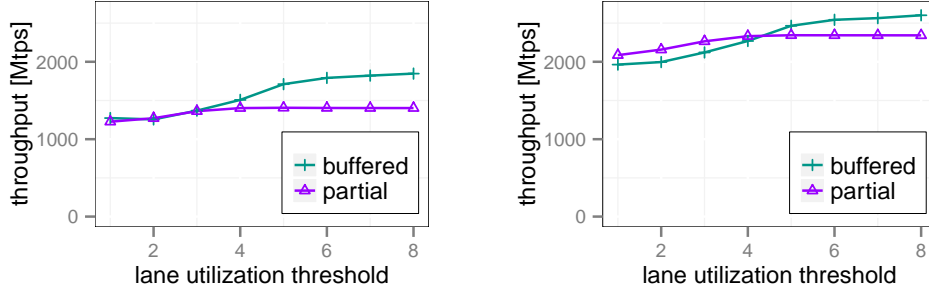


Figure B.10: Hashjoin performance when varying build sizes. KNL, 128 threads.

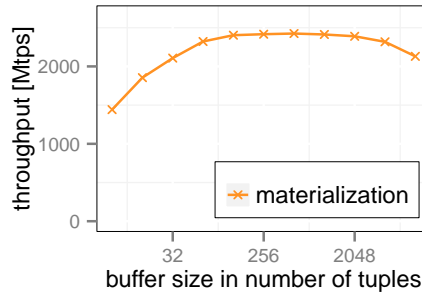
sizes. The overall throughput on Knights Landing is about twice as high as on Skylake-X, even though the performance of Skylake-X can be increased by 50% by using Hyper Threading. On Skylake-X (Figures B.9a and B.9b), as expected, a sharp performance decrease happens when the hash table grows beyond the size of the L2 cache at around 1 MiB, and at around 10 MiB when it exceeds the L3 cache. For large hash tables, that do not fit into the cache, all approaches converge. This has also been observed in earlier work, for instance, by Polychroniou et al. [140] and by Kersten et al. [86]. In these cases, partitioning the hash table might help (cf. the radix partitioning join proposed by Kim et al. [88]), but this is out of scope for this paper.

When the hash table is small enough to fit into the L1 or L2 cache, all SIMD approaches outperform the scalar baseline: Irrespective of the SIMD divergence handling deployed by the individual approaches, they all reach a higher throughput than the scalar approach. For larger hash tables, SIMD divergence no longer dominates the performance, and thus the scalar approach reaches similar throughput levels (using 10 threads) or even higher throughput (using 20 threads) than some SIMD variants. For the whole evaluated range of hash table sizes, the partial and buffered approaches that make use of the introduced refill strategies outperform or are on par with the divergent SIMD approach. Using 20 threads, the buffered approach achieves up to 32% higher throughput than the divergent approach, while the partial approach outperforms the divergent one by up to 19%.

When the hash table fits into the L1 cache, the buffered approach defeats the materialization approach by up to 8%. When the hash tables grows, the materialization approach dominates all other approaches. Two contradicting influences determine whether the materialization approach outperforms our divergence-handling approaches: First, the materialization approach can hide memory latencies better than the buffered and partial approaches because more memory is accessed at the same time (i.e., multiple outstanding loads). This



(a) Varying thresholds. SKX, 10 threads. (b) Varying thresholds. SKX, 20 threads.



(c) Varying buffer sizes. SKX, 20 threads.

Figure B.11: Hashjoin performance when varying algorithm parameters.

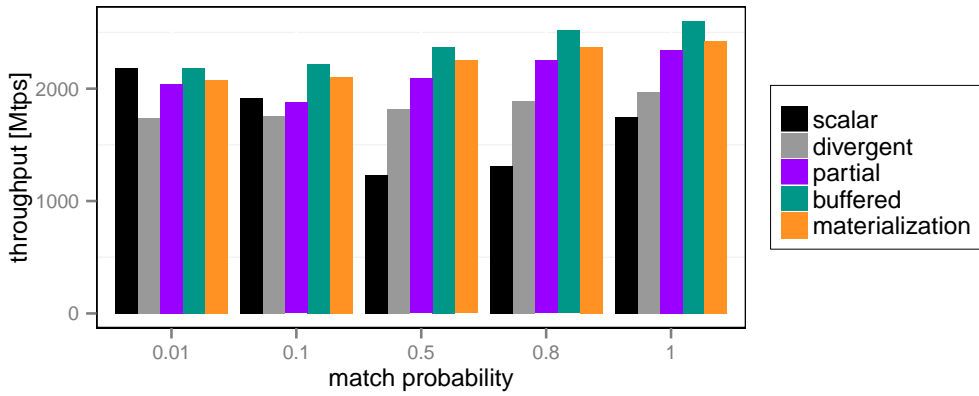
is shown in Figure B.9a and more severely in Figure B.9b when the hash table is large, because it then resides in slower memory. Second, the materialization approach suffers from the higher number of issued instructions, i.e., `load` and `store` instructions. In particular, when the hash table fits into L1, the number of instructions can become the limiting factor. On the Knights Landing platform in particular, the materialization approach has a significantly lower throughput compared to the other SIMD variants (Figure B.10). In contrast to the table scan, which we evaluated in Section B.6.1, the materialization buffer is read and written in the same loop multiple times—during index lookup, which exceeds the limited out-of-order execution capabilities of KNL.

In Figures B.11a and B.11b, we vary the SIMD lane utilization threshold for the partial and buffered approaches. Hyper Threading, i.e., using 20 threads instead of 10, increases throughput by about 50%. In general, a higher threshold, i.e., less inactive SIMD lanes and more refills, results in a higher throughput. There is little change in throughput when setting the threshold to six, seven or all eight tuples. This is because in most hash table lookups, only a few of the eight tuples need to be kept for additional pointer lookups in the collision chains. As a result, almost no refills are done differently when choosing six, seven or eight as the threshold. The buffered approach is more sensitive to the

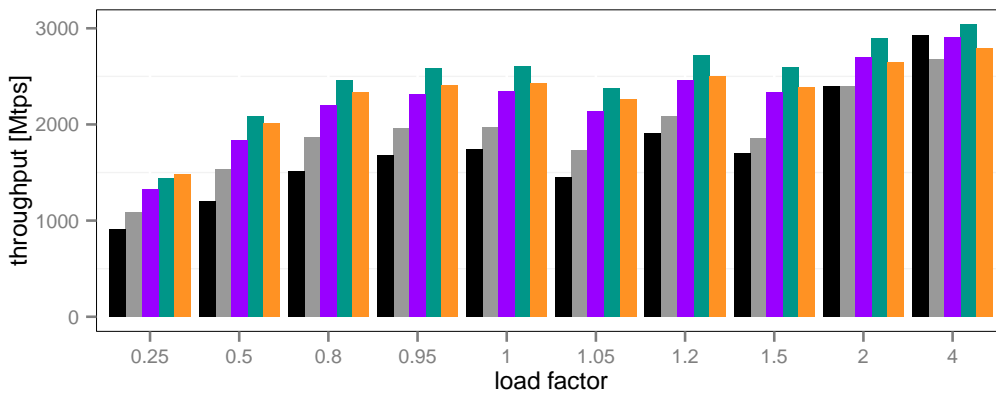
chosen threshold. For a low threshold, the partial approach reaches a higher throughput, but that changes at threshold 3 (using 10 threads) or 5 (using 20 threads). As mentioned, only few tuples need to be kept for additional lookups. Thus, only few tuples need to be buffered in the buffered approach, while the partial approach suffers from underutilization when frequently performing refills in the table scan.

Figure B.11c focuses on the materialization approach, varying the buffer size between eight and 8192 tuples and a fixed build cardinality (hash table size \approx 128 KiB). For the chosen configuration, the scalar approach reaches a throughput of 1747 Mtps. For small buffers, e.g., 8 tuples, the scalar approach outperforms the materialization approach. The materialization approach with an 8-tuple buffer is conceptually similar to the buffered approach with a SIMD line utilization threshold of 1. Both use a buffer the size of one SIMD vector. In the buffered approach, this buffer lives in registers, while the materialization approach stores it in memory. As a result, the buffered approach outperforms the materialization approach with a throughput of 1964 Mtps (i.e., about 2 billion tuples per second). A buffer size between 128 and 1024 results in the best performance of the materialization approach. The throughput shrinks gracefully when the buffer size is further increased. This is an effect of the chosen workload, especially the number of attributes beside the join attribute.

Two additional parameters affect throughput in the hashjoin evaluation: First, the *match probability* describes how likely a tuple from the probe side finds a join partner in the hash table. We vary this probability between 0.01 and 1. A low match probability therefore results in more disqualified tuples, which—depending on the approach—in turn leads to more ignored SIMD lanes, more refills, or a worse VPU utilization. Figure B.12a shows that the buffered approach, using the proposed refill strategies, outperforms both pre-existing approaches, scalar and divergent, irrespective of the match probability. The scalar approach is competitive with the SIMD approaches for low match probabilities. With few matches, the scalar approach can often exit the pipeline early, which leads to the high throughput rates we observed. The divergent approach, on the other hand, suffers from extreme under-utilization because frequently only few SIMD lanes stay active due to the low match probability. With a match probability of 50%, branches are mispredicted in the scalar approach, and its throughput subsequently tanks. When the match probability approaches 100%, almost all probe tuples find a non-empty hash bucket that then needs to be inspected further. Furthermore, the final aggregation becomes more expensive as more tuples make it into the join result. The scalar approach therefore performs best for low match probabilities and worst for a match probability of around 50%, and cannot fully recover its throughput even for a match probability of 100%. When looking at the SIMD approaches, we observe that the throughput difference between the divergent approach and our novel refill approaches increases with the match probability. A higher match probability comes along with more active SIMD lanes after the first lookup in the hash dictionary. Then, more divergence happens because these tuples will



(a)



(b)

Figure B.12: Hashjoin performance for varying match probabilities (a) and hash table load factors (b). SKX, 20 threads.

have to traverse collision chains of different lengths. The divergence-handling buffered and partial approaches can therefore outperform the divergent approach for high match probabilities.

Second, we define the hash table’s *load factor* as the number of buckets in the hash table divided by the number of keys stored in the hash table. While the load factor has been kept constant in all previous experiments (= 1.0), in real scenarios, the hash table size is not only determined by the size of the build side input, but also by the set load factor. With a low load factor, more collisions in the hash table occur, resulting in longer chains. With longer chains, the variance of the number of pointers that need to be followed to perform the hash table probe increases. This variance directly translates to higher SIMD divergence. A low load factor therefore leads to worse VPU utilization in the divergent approach, which can then be mitigated by applying the proposed in-register refill strategies. Figure B.12b shows how the load factor affects the throughputs reached by the different approaches. The hash table for load factor 4 is 16 times as big as the hash table for load factor 0.25. Over all approaches, the throughput of the bigger hash table is about three times as high as for the smaller one. For high load factors, the scalar approach performs well. This is because for high load factors, fewer and shorter collision chains exist. When zero of the eight tuples in a vector need to follow a chain, there is not SIMD divergence. Subsequently, for especially high load factors like 4, there is little difference between all approaches.

B.6.3 Approximate Geospatial Join

In the following, we evaluate and compare our approach with a modern and more complex operator, an approximate geospatial point-polygon join. Our approximate geospatial join [91] uses a quadtree-based hierarchical grid to approximate polygons. Figure B.13 shows such an approximation for the neighborhoods in New York City (NYC). The grid cells are encoded as 64-bit integers and are stored in a specialized radix tree, where the cell size corresponds to the level within the tree structure (larger cells are stored closer to the root node and vice versa). During join processing, we perform (prefix) lookups on the radix tree. Each lookup is separated into two stages: First, we check for a *common prefix* of the query point and the indexed cells. The common prefix allows for the fast *filtering* of query points. If the query point does not share the common prefix, there are no join partners. The actual tree traversal takes place in the second stage. We traverse the tree starting from the root node until we hit a leaf node (which contains a reference to the matching polygon).

An important property of our approximate geospatial join operator is that it can be configured to guarantee a certain precision. In the experiments, we used 60-, 15-, and 4-meter precision (as in [91]). The higher the precision guarantee, the smaller are the cells at the polygon boundaries, which in turn increases the total number of cells and, more importantly, the height of the radix tree. In general, the probability of control flow divergence during index



Figure B.13: Quad-tree based cell-approximation of neighborhood polygons in NYC.

	number of polygons	avg. number of vertices
boroughs	5	662.2
neighborhoods	289	29.6
census	39184	12.5

Table B.2: Polygon datasets

lookups increases with the tree height. Throughout our experiments, the tree height is ≤ 6 .

In our experiments, we join the boroughs, neighborhoods, and census blocks polygons of NYC⁴ with randomly generated points, uniformly distributed within the minimum bounding box of the corresponding polygonal dataset. The datasets vary in terms of the total number of polygons and complexity (with respect to the number of vertices).

Table B.2 summarizes the relevant metrics of the polygon datasets, and Table B.3 summarizes the metrics of the corresponding radix tree, including the probability distribution of the number of search steps during the tree traversal.

⁴The polygons of NYC are available at:

- <https://data.cityofnewyork.us/City-Government/Borough-Boundaries/tqmj-j8zm>
- <https://data.cityofnewyork.us/City-Government/Neighborhood-Tabulation-Areas/cpf4-rkhq>
- <https://data.cityofnewyork.us/City-Government/2010-Census-Blocks/v2h8-6mxf>










polygons	boroughs			neighborhoods			census		
precision [meter]	60	15	4	60	15	4	60	15	4
# of cells [M]	0.08	1.27	20.7	0.11	0.79	13.2	6.08	6.52	34.6
tree size [MiB]	1.39	168	168	25.3	139	139	1162	1205	1205
tree traversal depth									

Table B.3: Metrics of radix tree

Query Pipeline

The query pipeline of our experiments (point-polygon join) consists of four stages:

- (1) Scan point data (source)
- (2) Prefix check
- (3) Tree traversal
- (4) Output point-polygon pairs (sink)

Stages (2) and (3) are subject to control flow divergence, with (3) being significantly costlier than (2). For simplicity, the produced output (point-polygon pairs) is not further processed. We compile the pipeline in three different flavors:

Divergent: Refers to the baseline pipeline without divergence handling, thus the pipeline follows consume everything semantics. The code of subsequent operators is executed if at least one lane is active.

Partial: The partial consume strategy is applied to stages (2) and (3), which also affects the scan operator because it needs to be aware of protected lanes.

Buffered: Follows consume everything semantics with register buffers in stage (3). We check the lane utilization after each traversal step. Divergence in stage (2) is not handled at all.

Materialization: The integration of memory materialization is similar to the one used with the hash join operator (cf., Section B.6.2).

Results

Figure B.14 shows the performance results in million tuples per second on KNL using 128 threads. We observe that refilling from register buffers improves the overall throughput by up to 20% (=870 mtps) when joining with the boroughs or neighborhood polygons. The effect of divergence handling falls below 10%

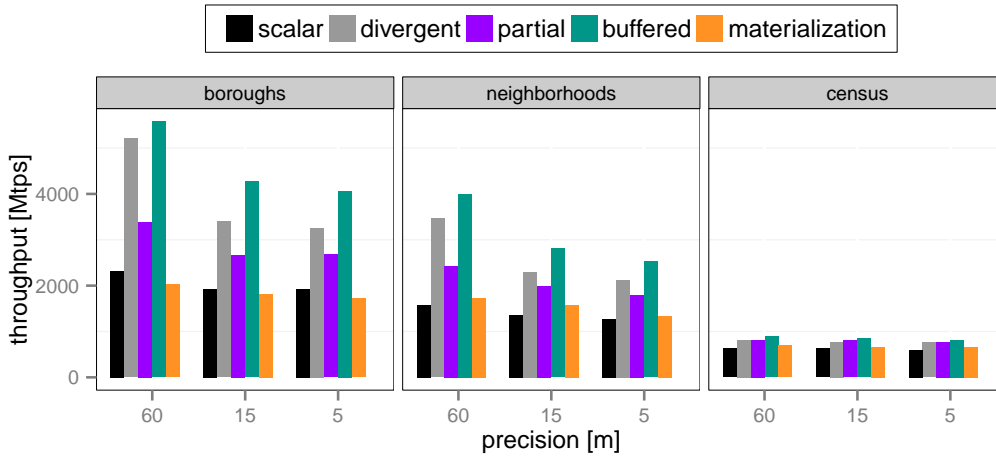


Figure B.14: Geospatial join performance for varying workloads and precisions.

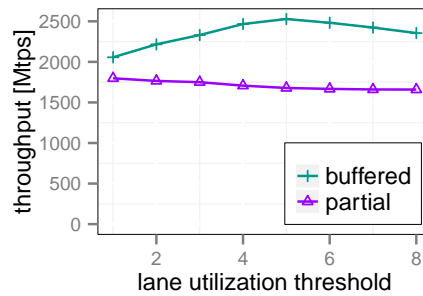


Figure B.15: Varying thresholds. KNL, 128 threads

with the census blocks polygons where the index structure is more than 1 GiB in size. In that case, the memory subsystem is the limiting factor.

As expected, the partial consume strategy exacerbates the divergence issue in most cases (cf., Section B.5.3), resulting in a 53% performance degradation in worst case.

The materialization approach performs poorly on KNL. The throughput is similar to the scalar implementation, thus cancelling out all SIMD optimizations. As in previous benchmarks, we observed a significantly better performance on SKX. Here, the materialization approach is on par with the buffered pipeline: in case of small index structures (boroughs) slightly worse, and with large indexes (census) slightly better. In the latter case, the materialization approach helps to hide memory latencies through out-of-order execution.

Unlike the previous experiments, the optimal lane utilization threshold for the buffered approach is less than the number of SIMD lanes (cf., Figure B.15), which is due to the higher refilling costs involved in the geojoin operator. During the radix tree traversal, refilling affects five vector registers, whereas

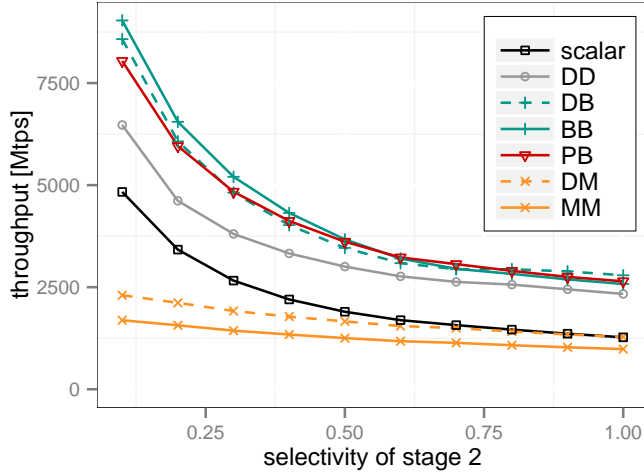


Figure B.16: 2-way divergence handling.

in the hash join experiment, refilling affects three registers; and only one in the table scan experiment. The optimal threshold for the partial approach is 1, indicating that a refill from the pipeline source is not efficient.

In the experiment above, all points pass the prefix check stage (2) and therefore cause an radix tree traversal. In the following, we also apply divergence handling on the second stage of the pipeline and we changed the workload so that a certain amount of points are disqualified in that stage. We compiled the query pipeline with several combinations of the different approaches. We refer to it using the first letter of the approach (**D**ivergent, **B**uffered, **P**artial, and **M**aterialization). For instance, **PB** refers to the pipeline that uses partial consume in stage two and in-register buffering in the third stage, and **BB** uses buffering in both stages. Figure B.16 shows the results for the neighborhood/4 meter precision workload with varying selectivities. We observe an 8% performance decrease when the buffered approach is applied to stages 2 and 3, and the selectivity remains at 1.0. In contrast, the materialization approach adds a significantly larger overhead (35% decrease). If materialization is applied in both pipeline stages, the performance is worse compared to the pipeline, where it is applied only in the tree traversal stage. Overall, the performance difference for lower selectivities is relatively small with the partial and buffered approaches: +5% with buffering applied in both stages, -7% when partial consume is applied in stage 2 and buffering in stage 3. Compared to the divergent pipeline, lane refilling increases the throughput of the neighborhood workload by up to 30% with lower selectivities.

B.6.4 Overhead

In our final experiment, we evaluate the overhead of divergence handling with a varying number of attributes. To quantify the overhead, we use a very simplistic query that consists of a simple selection and a scalar aggregation

(select sum(a1), sum(a2), ..., sum(aN) from...). Divergence is handled immediately after the selection and before the aggregation. In that scenario, we expect the divergent pipeline to perform best, as the remainder of the pipeline only consists of a single addition and thus the benefits of refilling are close to zero.

In the following, we consider two different selectivities:

sel = 1: For in-register buffering, this situation is the one with the lowest overhead, as the tuples are passed through to the subsequent operator and the buffer registers are not used altogether (cf. Listing B.3). Thus, the overhead is rather small, as it effectively consists of a `popcount` to determine the number of active lanes and a branch instruction. The same applies for partial consume pipelines.

sel = 0.125 = 1/LANE_CNT: A selectivity of 1/LANE_CNT results in one active lane per iteration (on average) and thus represents the most write intensive case for in-register buffering. I.e., the refill algorithm, which moves active elements to the buffer registers, is executed in almost every iteration. The partial consume strategy, on the other hand, suffers from lane underutilization caused by lane protection, and thus, the lower part of the pipeline is executed more frequently.

Throughout all experiments, the pipelines are 8-way data parallel and we set the minimum lane utilization threshold to 6 for buffered and 4 for partial; the size of memory buffers are fixed to 1024 elements (= 8 KiB). The number of attributes are varied within the range [1, 32].

Figure B.17 summarizes the results for both evaluation platforms. On KNL, all approaches perform similarly with up to four attributes and the overhead, i.e., the performance difference to the divergent pipeline, is barely measurable. The materialization approach degrades significantly when the number of attributes increases (2.5 CPU cycles per tuple per thread compared to 0.14 cycles for divergent). The throughput of the buffered approach degrades as well, which is also attributed to memory materializations. The high register file pressure forces the compiler to evict values to memory. Even though the buffer registers are not used in the case of *sel* = 1, register allocation is static and happens at query compilation time when the actual selectivity is not known. Therefore, a performance degradation can be observed even if register buffers are not used at query runtime. In contrast, the partial consume pipelines are on par with the divergent pipelines.

On the SKX platform, the performance degrades more steeply with an increasing number of attributes. In case of *sel* = 1, the throughput of the materialization approach decreases linearly with the number of attributes. Compared to KNL, the number of attributes has a higher impact on the overall performance on SKX. For instance, in-register buffering is 4× faster on KNL

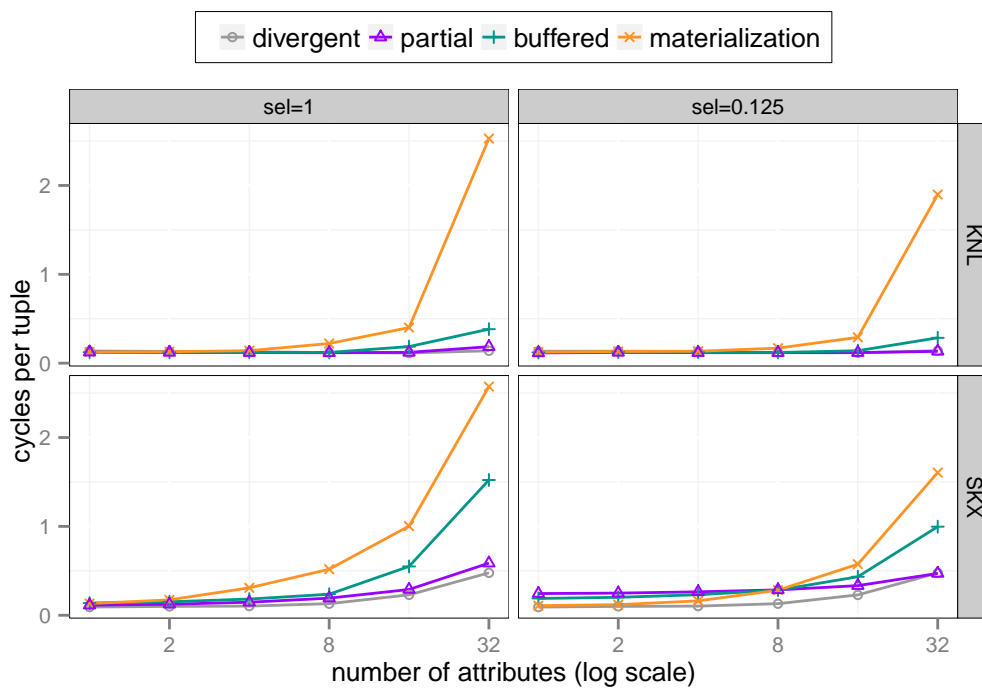


Figure B.17: Overhead of divergence handling for varying number of attributes.

with $sel = 1$ and $3.6\times$ faster with $sel = 0.125$. For $sel = 0.125$ and a single projected attribute, we measure an overhead of approximately 0.1 cycles per tuple for buffered and 0.15 cycles per tuple for partial, which is significantly higher than with the materialization approach (0.02 cycles). However, the per attribute overhead of buffered and partial decreases with more projected attributes, whereas the materialization approach shows an increasing overhead with an increasing number of attributes. The crossover point is reached with 8 projected attributes. Afterwards, our approaches are consistently faster.

In general, the partial consume approach shows no performance impact when the number of projected attributes increases, which is an expected result, because the bookkeeping overhead about protected lanes is constant, irrespective from the number of projected attributes. The actual overhead of the partial consume strategy depends on the pipeline costs, more precisely on the pipeline fragment before divergence handling (see Section B.5.3).

B.7 Summary and Discussion

The partial consume strategy shows performance improvements for relatively simple workloads. With more complex workloads, like the geospatial join, we observe severe performance degradations. The reason for that is two-fold. *(i)* Protected lanes inherently cause the underutilization of VPUs (as described in Section B.5) and *(ii)* they result in a suboptimal memory access pattern at the pipeline source where the refill happens. In contrast to the consume everything strategy, where in every iteration exact `LANE_CNT` elements are read from memory, a partial consume scan reads *at most* `LANE_CNT` elements. This circumstance reduces the degree of data parallelism (fewer elements are loaded per instruction) and also leads to unaligned SIMD loads. Even though the access pattern is still sequential, the alignment issues can reduce the load throughput by up to 25% (on our evaluation platforms), which could severely reduce the overall performance of scan-heavy workloads.

We found that the materialization approach is very sensitive to the underlying hardware, in particular, on KNL, the approach performs poorly when the buffer is read *and* written within a tight loop (intra-operator), an effect that could not be observed on SKX. On the other hand, if materialization is applied at operator boundaries and thus written and read only once, it performs similarly or better than in-register buffering, as it benefits from out-of-order execution, which allows the materialization approach to hide memory latencies. Memory access latencies play an important role when the data that is randomly accessed (like a hash table) does not fit into the L1/L2 cache. In contrast, when the data fits into cache or the workload is more compute-heavy, the in-register buffering approach dominates because the buffers provide much faster access.

The SIMD lane utilization threshold (refill more often vs. VPU underutilization) has a big impact on the buffered approach and less impact on partial.

As buffered shows better performance in general, this parameter is important. Choosing the highest possible threshold shows the best results in simple workloads, so going back down the pipeline to refill the vector is always better than having inactive lanes, we found. So the idea of materialization, where only active (or qualifying) elements are passed along the pipeline, was right in these scenarios. The picture changes with more complex operators like the geojoin, where refilling affects five vector registers. In this case, refilling doesn't pay off for a single idle SIMD lane. On average, the optimal utilization threshold was 5 out of 8 among the geospatial related experiments.

It remains an open question how the optimal threshold can be predicted at query compilation time, as it depends on hardware, refilling costs, the costs incurred by underutilized lanes, and the actual input data. A possible approach to address this issue is to adaptively adjust the threshold parameter at runtime (per batch or per morsel [100]). Nevertheless, divergence handling cannot fully be disabled once the pipeline has been compiled. One can set the threshold to 1, which is equivalent to a divergent execution, but some overhead remains in the compiled code, namely the the population count instruction and the branching logic. For instance, in our geospatial experiments on KNL, we observed an overhead of up to 6% over divergent with the boroughs workload when the utilization threshold is set to one (neighborhoods 3.6%, census 0.6%). Dynamically adjusting the threshold at query runtime provides some flexibility but due to the fact that divergence handling cannot be fully disabled, a database system needs to decide at compilation time whether to enable or disable divergence handling altogether.

Finally, we want to point out that our proposed refill algorithms and strategies are generally applicable to any data processing system that uses AVX-512 SIMD instructions. A prominent open-source representative is Apache Arrow [2] (in combination with Gandiva) which shares many similarities with state-of-the-art relational database systems (e.g., columnar storage, JIT compilation, and operator fusion). Further, our approaches are also applicable if the underlying database system uses compression in its storage layer. In particular, when compression is only used on secondary storage, it does not affect query execution. However, recent systems [129, 96] tend to use light weight compression techniques that allow for the processing of data without explicitly decompressing it. This implies that the degree of data-parallelism can be increased, as more attributes can be packed into a single vector register. Currently, our buffered approach is limited to 16-way data-parallelism on the KNL and SKX platforms, but it can be easily extended to 64-way data-parallelism for upcoming processors with the AVX-512/VBMI2 instruction set.

B.8 Conclusions

In this work, we presented efficient refill algorithms for vector registers by using the latest SIMD instruction set, AVX-512. Further, we identified and pre-

sented two basic strategies for applying refilling to compiled query pipelines for preventing the underutilization of VPUs. Our experimental evaluation showed that our strategies can efficiently handle control flow divergence. In particular, query pipelines that involve traversing irregular pointer-based data structures, like hash tables or radix trees, can significantly benefit from divergence handling. Especially when the workload is compute-intense or fits into fast caches, our novel approach shows better performance than existing approaches that rely on memory buffers.

Nevertheless, our research also showed that SIMD still cannot live up to the high expectations set by the promising features of the latest hardware, i.e., providing n -way data-parallelism. In practice, SIMD speedups are only a fraction of the advertised degree of data-parallelism, for many reasons, including underutilization. Our refill algorithms address this important reason, yet merely achieve a $2\times$ speedup over scalar code.

B.9 Acknowledgements

This work has been 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. Furthermore, this work is part of the TUM Living Lab Connected Mobility (TUM LLCM) project and has been partially funded by the Bavarian Ministry of Economic Affairs, Energy and Technology (StMWi) through the Center Digitisation.Bavaria, an initiative of the Bavarian State Government.

C Tree-Encoded Bitmaps

Harald Lang¹, Alexander Beischl¹, Viktor Leis², Peter Boncz³, Thomas Neumann¹, Alfons Kemper¹

¹ Technical University of Munich

² Friedrich Schiller University Jena

³ Centrum Wiskunde & Informatica

Appeared in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2020. <https://doi.org/10.1145/3318464.3380588>

The content of this section is identical to the original publication. Only the format and the numbering have been adjusted.

In accordance with the TUM regulations for the award of doctoral degrees (TUM Promotionsordnung, 2014), a summary of the publication is included in the first part of this thesis. Please refer to Section 4.3. Furthermore, the printed version is included in the Appendix on page 205 ff.

The contributions of the thesis author to this publication are: inventing the idea of tree-based bitmap compression, the implementation, the evaluation, and authoring of substantial parts of the paper.

Abstract

We propose a novel method to represent compressed bitmaps. Similarly to existing bitmap compression schemes, we exploit the compression potential of bitmaps populated with consecutive identical bits, i.e., 0-runs and 1-runs. But in contrast to prior work, our approach employs a binary tree structure to represent runs of various lengths. Leaf nodes in the upper tree levels thereby represent longer runs, and vice versa. The tree-based representation results in high compression ratios and enables efficient random access, which in turn allows for the fast intersection of bitmaps. Our experimental analysis with randomly generated bitmaps shows that our approach significantly improves over state-of-the-art compression techniques when bitmaps are dense and/or only barely clustered. Further, we evaluate our approach with real-world data sets, showing that our tree-encoded bitmaps can save up to one third of the space over existing techniques.

C.1 Introduction

Bitmap indexes have a long history in database systems and information retrieval [131, 171, 36, 156, 134, 113, 33]. They have many applications, such as efficiently evaluating predicates [131, 133, 125] and have been used to accelerate join [130] and aggregation [133, 34] queries. For medium or high cardinality columns, bitmap indexes consist of many individual bitmaps that are sparsely populated with 1-bits. Therefore, plain bitmaps consume large amounts of space, and compression is essential.

Consider the case of a bitmap index on an attribute A consisting of A individual bitmaps of length n , where A is the number of distinct values of A and n the number of tuples in the corresponding relation. The total number of 1-bits in the index is also n , whereas each bitmap receives $\frac{n}{A}$ 1-bits on average. A high number of distinct values, or the presence of skew, results in bitmap indexes with many sparsely populated bitmaps. Sparsity implies that these bitmaps mostly consist of consecutive 0-bits, i.e., *0-runs*. Having long runs of identical bits offers great compression potential, which all existing bitmap compression schemes try to exploit.

One simple, but fairly effective bitmap compression scheme is the Word-Aligned Hybrid [179] (WAH) approach, whose compression is based on run-length encoding (RLE). A WAH-compressed bitmap is a sequence of machine words, typically 32 or 64 bits in size. Each word either encodes a run or represents a small part of the original bitmap as is. The first is called a *fill word* and the latter a *literal word*. While WAH offers significantly better performance than its predecessor the Byte-Aligned Bitmap Compression [14] (BBC), its compression effectiveness suffers from two major weaknesses: (i) runs need to be rather long for the RLE-based compression to be effective and (ii) WAH has linear space overhead (one bit per word) for distinguishing between fill and literal words. In particular, the first weak point impairs compression when some random bits (also called *dirty bits* or *odd bits*) disrupt long runs. Over the years, several extensions to WAH have been proposed to solve this issue, i.e., PLWAH [55], Concise [48], VAL-WAH [76], EWAH [103], and SBH [89].

All the aforementioned compression techniques are based on RLE and therefore share another disadvantage, namely the linear time complexity of random access. Supporting efficient random access directly affects the efficiency of logical operations like bitwise AND, which are common operations in analytical queries.

Chambi et al. identified this problem and proposed the Roaring Bitmap format [35]. In contrast to the aforementioned compression techniques, Roaring Bitmap does not rely on RLE. Instead it partitions the input bitmap into equally sized chunks of length 2^{16} bits, where each chunk is physically stored in a separate container, as illustrated in Figure C.1. Roaring implements three different container types and each container type represents the corresponding part of the bitmap differently. Depending on the number of bits set and on the presence of 1-runs, Roaring chooses the container type that consumes

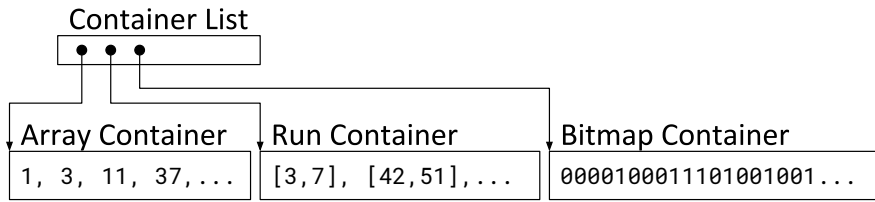


Figure C.1: Roaring partitions the bitmap and stores each partition using the best suitable container type.

the smallest amount of memory. More precisely, if the number of 1-bits is less than or equal to 4096, an *array container* is used that stores a sorted list of 16-bit integers, one for each set bit. The integer values correspond to the positions of those bits within the current partition. If the number of set bits exceeds 4096, Roaring either employs a plain *bitmap container* or a *run container* [102]. A bitmap container stores the partition as is. A run container on the other hand stores the 1-runs as a list of 16-bit integer pairs $\langle a, b \rangle$, where $[a, b]$ is the range spanned by the 1-run.

Overall, Roaring is a very lightweight approach in terms of compression, as it only relies on integer arrays to represent bitmaps. Integer values are thereby truncated to 16 bits as every container encodes 2^{16} bits of the bitmap. Nevertheless, it results in significantly lower space consumption compared to RLE-based techniques in most scenarios. Due to the fact that the bit positions, the runs, and the containers themselves are sorted, a random access can be performed in logarithmic time, which significantly improves the performance of bitwise operation and thus of analytical queries [34].

It is worth mentioning that in principle Roaring is an extendable format, as it could employ any bitmap compression technique at the container level; including the tree-encoded bitmaps, we present in this work.

At the time of writing, Roaring was available in 11 programming languages and was widely used in Apache projects like Druid, Hive, Kylin, Lucence, Spark, and other systems¹. This shows that today’s applications not only demand high compression ratios but also efficient logical operations on compressed bitmaps. Further, we see a trend in database systems towards denser bitmaps—in particular, when bitmap indexes use histogram-based binning or are constructed to support range queries [36, 37]. In both cases, the resulting bitmaps exhibit higher bit densities compared to simple bitmap indexes as described at the beginning of this section².

With this work, we contribute a novel method to compress bitmaps. The compressed representation, which we call a *tree-encoded bitmap*, provides high compression ratios paired with logarithmic access time. Its primary strengths are the abilities (i) to compress both long and short runs and (ii) to significantly improve the compression ratios with denser bitmaps over existing

¹We refer the reader to the official web site [7] for more details.

²In Section C.5 we give a brief overview on the design space of bitmap indexes.

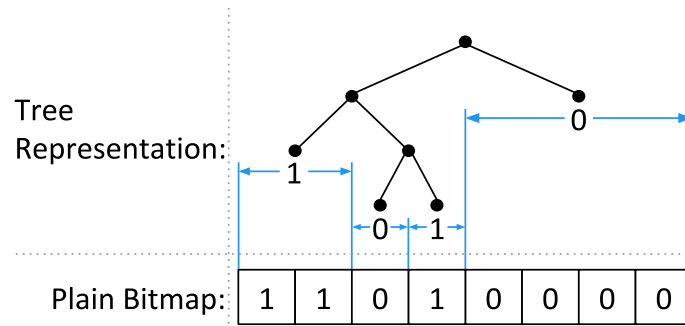


Figure C.2: The key idea is to represent bitmaps as full binary trees. Longer runs are mapped to tree nodes closer to the root, and vice versa.

techniques. The major conceptual difference compared to other compressed bitmap formats is that our approach employs a binary tree to represent bit runs of various lengths as illustrated in Figure C.2. Tree nodes in the upper tree levels (closer to the root) thereby correspond to longer runs, and tree nodes in the lower levels to shorter runs. The low space requirement is achieved by using a succinct tree encoding and additional space optimizations that truncate balanced parts of the tree structure from the compressed representation. A key insight is that although our approach initially triples the size of a given bitmap to establish the tree structure, it does not only amortize this overhead, but also ultimately offers overall better compression ratios than RLE-based compression methods or the state-of-the-art Roaring Bitmap in a wide spectrum of moderately populated and clustered bitmaps. Using a collection of real-world data sets, we empirically found that tree-encoded bitmaps offer the best compression in 7 out of 8 cases, saving up to $1/3$ space in comparison with the second best solution.

Notation. Throughout the paper, we let n denote the length of a bitmap. Further, since compression heavily depends on the data distribution, we use the following two metrics to characterize individual bitmaps: (i) The *bit density* denoted as d refers to the fraction of bits set to 1, where $0 \leq d \leq 1$. The total number of set bits in a bitmap is therefore $d \cdot n$. (ii) The *clustering factor* denoted as f , with $1 \leq f \leq n$, indicates the degree of clustering of the 1-bits in a bitmap, i.e., how likely a 1-bit is followed by another 1-bit. Formally, it is defined as the average length of the 1-runs in a bitmap [179]. For instance, the bitmap 01110010 (with $d = 0.5$) contains two 1-runs, one of length 3 and one of length 1. The clustering factor f therefore equals to 2. As both d and f refer to the set bits, they are dependent and the following restrictions apply: The clustering factor cannot exceed the total number of bits set ($f \leq d \cdot n$). Further, when the bit density exceeds 50%, the smallest possible value for f increases as well. E.g., given the bitmap 01010101 with $d = 0.5$ and $f = 1$; when the leftmost 0-bit is toggled (11010101), d increases to 0.625 and f to 1.25. In that particular case, 1.25 is the smallest possible clustering for a bitmap of length

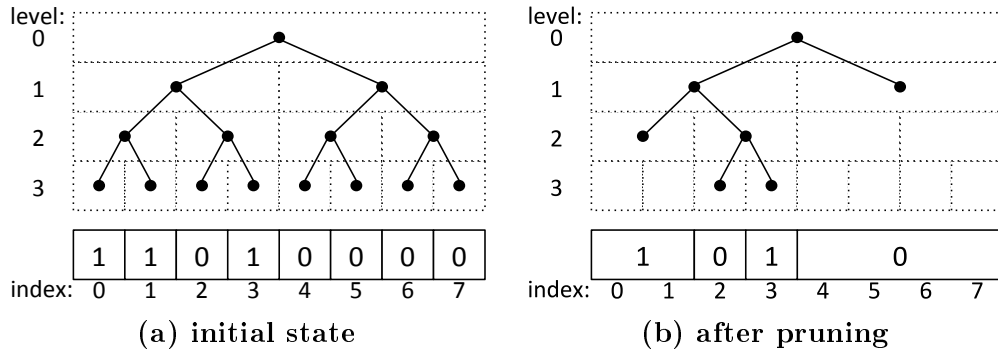


Figure C.3: A bitmap represented as a binary tree. Initially, each leaf node is assigned a single bit (label). Sibling leaf nodes with identical labels are then pruned and the label is assigned to their parent. After pruning, the prior parent node becomes a leaf and represents multiple consecutive bits, a 0-run or a 1-run.

$n = 8$ and $d = 0.625$. In the general case, the smallest possible clustering is $\max(1, d/(1-d))$. Clustered bitmaps can be synthetically generated using a two-state Markov process, which we describe in the evaluation section.

C.2 Tree-Encoded Bitmaps

In this section, we present our **Tree-Encoded Bitmaps (TEB)**. The key idea behind TEB is to represent bitmaps as binary trees, which enables efficient navigation and therefore fast random access. The data structure is best explained by describing the construction algorithm. We therefore first present the tree-based compression algorithm. Later in this section, we describe how the tree is encoded space efficiently.

C.2.1 Compression

A TEB is constructed in two phases. In the first phase, a perfect binary tree is established on top of a given bitmap, as shown in Figure C.3a. Each bit in the bitmap is associated with a single leaf node of the binary tree. Only leaf nodes carry a payload, which we refer to as *labels*. A label can either be a 0-bit or a 1-bit.

In the second construction phase, the binary tree is pruned bottom-up. Thereby, the algorithm removes all sibling leaf nodes with identical labels l , and the label l is assigned to the parent node. The pruning process stops when all pairs of sibling leaf nodes have different labels. Figure C.3b depicts a fully pruned tree. The important thing to note here is that the newly created leaf nodes in the upper tree levels no longer represent individual bits of the bitmap; instead they represent consecutive bits that form either a 0-run or

a 1-run. For instance, the leftmost node in Figure C.3b represents a 1-run of length 2, starting at index 0 and the rightmost node represents a 0-run of length 4, starting at index 4.

With every single pruning step, two nodes are eliminated from the tree structure and one bit from the labels. Bottom-up pruning can therefore be considered a *lossless compression* method. Compressing the tree structure is a crucial part of TEB because the space overhead of the tree structure needs to be amortized. The tree initially consists of $2n - 1$ nodes, assuming n is a power of two. When the tree structure is encoded using one bit per node, then the space consumption of a TEB, including the labels, is initially, and in worst case, $3n - 1$ bits. Even though the worst case space consumption is relatively high, we will show that our tree-based representation of bitmaps often achieves significantly lower space usage than other compression schemes.

C.2.2 Encoding

An important part of TEB is the space-efficient way the tree structure is stored. We employ a *level-order binary marked* representation [81], which requires one bit per tree node. The encoded tree itself therefore is a sequence of bits (a bitmap).

We have to differentiate between the tree data structure that is used during compression and the *encoded* tree that is eventually stored in a TEB. For the tree-based compression, we temporarily make use of an implicit data structure [174] that allows for fast modifications, but occupies a constant amount of space – constant in the sense that its size does not change when nodes are removed. The level-order binary marked representation, on the other hand, is static but requires less space once the tree has been pruned. Thus, *encoding* is the process with which we transform the pruned tree into a more compact form.

To encode the pruned tree structure we traverse it in breadth-first left-to-right order (or level-order) and for each visited node a single bit is emitted, a 1-bit for inner nodes and a 0-bit for leaf nodes. These bits are appended to the bit sequence that represents the encoded tree, denoted as T . The labels of the leaf nodes are stored as a separate bit sequence to which we refer as L . When a leaf node is observed during traversal, its label bit is appended to L . For instance, the tree in Figure C.3b is encoded as $T = 1100100$, $L = 0101$.

To support efficient random access and bitwise operations, it is necessary to traverse the tree. Internally, the most important primitive operation is to determine the two child nodes of some given tree node, i.e., navigating downwards the tree. Within the encoded tree, each tree node is identified by its position in the bit sequence T . The sequence starts with the root node at position 0. For any given tree node i , the child nodes can then be determined

as follows [81]:

$$\begin{aligned}\text{left-child}(i) &:= \text{right-child}(i) - 1 \\ \text{right-child}(i) &:= 2 \cdot \text{rank}(i)\end{aligned}$$

where $\text{rank}(i)$ refers to the number of 1-bits (inner nodes) in T within the range $[0, i]$.

Computing the rank of a node is a linear-time operation, and navigating from the root to any leaf node is therefore an $O(n \cdot \log n)$ operation. However, the rank operation can be turned into an $O(1)$ operation at the cost of additional space consumption [81]. TEB uses an implementation similar to the one used in [188], which pre-computes the rank on 512-bit block granularity and stores the values in an auxiliary integer array; which results in a 6.25% increased memory footprint. The rank is then computed as

$$\text{rank}(i) := R[\lfloor i/512 \rfloor] + \text{popcount}(T, \lfloor i/512 \rfloor \cdot 512, i)$$

where R refers to the array with the pre-computed values at block level and popcount counts the 1-bits in the last block up to index i .

Using an additional integer array populated with pre-computed ranks (a lookup table) is a common approach [70, 126, 69, 190] and changing the granularity of the lookup table offers a space/time trade-off. The more coarse-grained the lookup table is, the lower its space requirement and the higher the costs for counting the 1-bits within the last block; and vice versa. For TEB, we empirically determined that a granularity of 512 bits offers competitive performance at a reasonable space overhead. On a reasonably modern 64-bit hardware, a navigational operation in the tree therefore requires at most eight population count instructions (four on average) and one array lookup.

Besides the downward navigation, the rank of a tree node is further required to determine the node's label. If the node i is a leaf, then the position of the label within L is equal to the number of 1-bits in T preceding node i , which corresponds to the non-inclusive rank of i . However, because only leaf nodes have labels, we can use the inclusive³ rank from above, because $T[\text{rank}(i)]$ is guaranteed to be a 0-bit. In summary, a label is accessed as follows:

$$\text{label}(i) := L[i - \text{rank}(i)]$$

Let us close by mentioning that the chosen encoding requires the tree structure to be a full binary tree, i.e., each node has either zero or two child nodes. It is easy to show that this holds for the tree structure of a TEB: Since the initial binary tree is perfect, and pruning always affects two sibling leaf nodes, the resulting tree structure remains full binary.

C.2.3 Optimizations

The basic idea of TEB we have presented so far already shows promising results with regard to compression ratios. For instance, Figure C.4 shows a space

³We chose the inclusive rank as it results in fewer arithmetic instructions.

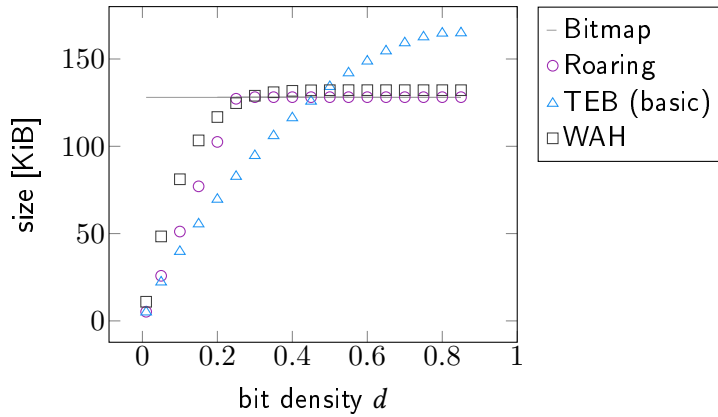


Figure C.4: Size comparison for varying bit densities and a fixed clustering factor of 8.

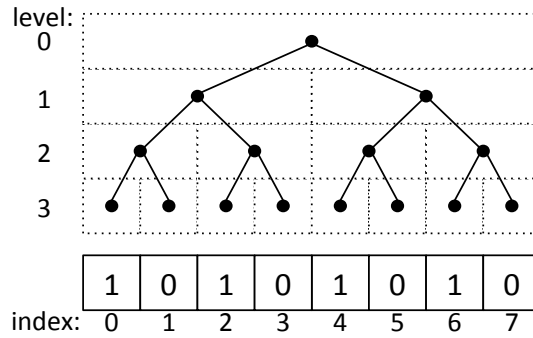


Figure C.5: In worst case, the tree cannot be pruned (compressed) and the resulting TEB consumes approximately three times the space of the original bitmap.

comparison of the TEB approach with two state-of-the-art bitmap compression techniques, Roaring and WAH. The compressed size (y-axis) depends on the ratio of 1-bits in the original bitmap (x-axis). Sparsely populated bitmaps offer higher compression potentials than densely populated bitmaps. In that particular case, if more than $\sim 25\%$ of the bitmap is populated with 1-bits, Roaring and WAH do not offer any compression at all. Both fall back to an uncompressed (literal) representation. TEB, on the other hand, is able to compress bitmaps with a bit density of up to $\sim 45\%$.

The downside of the basic TEB approach is that in corner cases it can significantly exceed the size of the plain bitmap. In contrast to Roaring and WAH, our approach does not support an alternative representation to which it could fall back. In the following, we show that it is in fact not necessary to switch between different representations to address the high space consumption of TEB in unfavorable cases. It just requires a few minor modifications to the data structure and the compression algorithm, which we discuss in the following.

Implicit Tree Nodes. We motivate our first space optimization by considering the worst-case scenario for TEB. Figure C.5 illustrates such a case. The depicted alternating bit sequence does not offer any compression potential. All pairs of sibling leaf nodes have different labels and therefore bottom-up pruning cannot remove any tree nodes. The resulting TEB would consist of $n - 1$ 1-bits for the inner nodes, followed by n 0-bits for the leaf nodes, and n label bits. In this extreme case, the label bits in L are identical to the uncompressed bitmap. Thus, storing the encoded tree structure is pure overhead.

Our first space optimization is to omit the leading 1-bits as well as the trailing 0-bits of the encoded tree structure. Only the intermediate bits of the tree structure are stored in the physical representation of a TEB. We refer to the omitted nodes as *implicit* tree nodes, and to the remaining as *explicit* tree nodes.

With regard to the worst case, this simple modification allows for the elimination of the entire tree encoding from the physical representation. Only the n label bits remain:

$$T = \underbrace{1111111}_{\text{leading 1-bits}} \underbrace{00000000}_{\text{trailing 0-bits}}, \quad L = 10101010$$

As mentioned before, the labels in L are identical to the original bitmap, i.e., the TEB degraded into an uncompressed bitmap. Thus, the size of the TEB is equal to the size of the plain bitmap, except for a small overhead that is caused by metadata.

However, further optimizations are needed, as this minor modification only mitigates the high space consumption of TEBs when the plain bitmap is poorly compressible. The TEB size may still significantly exceed the size of the uncompressed bitmap, i.e., the worst case has shifted. The modification, however, has two important implications:

- (i) The encoded tree structure T is an optional part of the physical TEB data structure, as the entire tree may be implicit.
- (ii) The space minimal TEB instance does not necessarily contain a fully pruned tree.

We give an example for (ii) in Figure C.6a. The depicted TEB consists of three explicit tree nodes and four labels. Thus the space requirement is $3 \cdot 1.0625 + 4 = 7.1875$ bits, where the factor 1.0625 is to incorporate the space consumption of the rank helper structure (cf. Section C.2.2). Figure C.6b shows the TEB instance with the minimum size. The difference between the two TEB instances is that in Figure C.6a the tree is fully pruned, whereas in C.6b the two sibling leaves in the highlighted subtree have been preserved. The second instance therefore comprises a larger tree, but even though the total number of tree nodes and labels are higher, the second instance occupies less

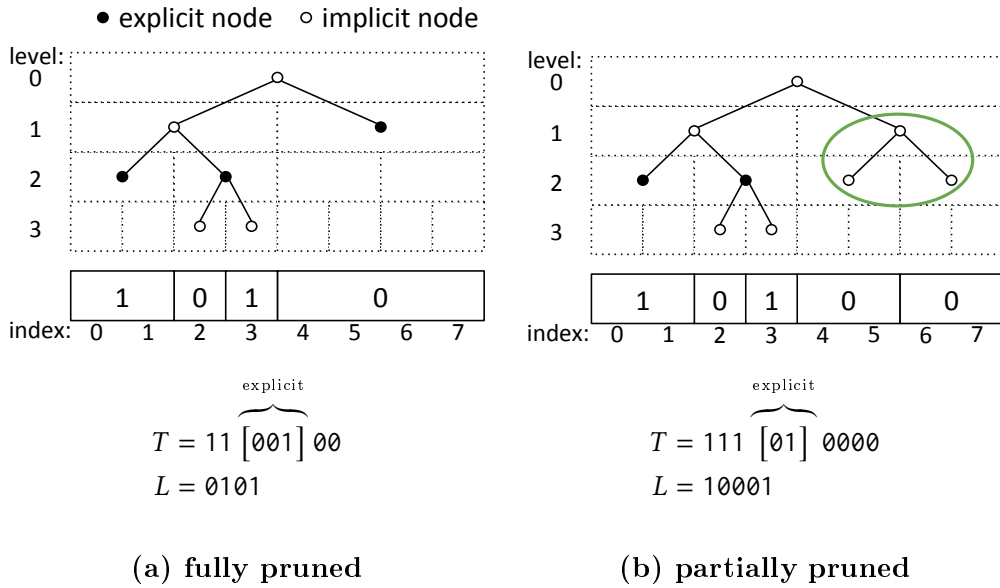


Figure C.6: Two different tree representations of the bitmap 11010000. The fully pruned tree (a) occupies more space than the partially pruned tree (b), as more tree nodes need to be stored explicitly.

space ($2 \cdot 1.0625 + 5 = 7.125$ bits), as fewer tree nodes need to be stored explicitly. The circumstance that a fully pruned tree, in general, no longer corresponds to the smallest TEB instance requires a modification to the bottom-up pruning algorithm: Instead of returning the fully pruned tree, the algorithm needs to return the smallest tree instance observed during pruning, where the size is computed based on the number of explicit nodes, rather than the total number of nodes.

Implicit Labels. Our second modification is to omit leading and trailing 0-labels in the physical TEB representation, similarly to implicit tree nodes. Omitting the leading 0-labels reduces the space consumption in particular with very sparse bitmaps. The tree representation of a sparse bitmap typically consists of a few leaf nodes with 1-labels at the deepest tree level $\log_2(n)$. But most of the leaf nodes with 0-labels can be found in the tree levels 1 to $\log_2(n) - 1$. Due to the tree being encoded in level order, the label bit sequence L tends to start with a long run of 0-labels, which we do not need to store explicitly. Trailing 0-labels on the other hand can occur when the length of the input bitmap is not a power of two. In that case, a TEB internally rounds up to the next power of two and fills the range $[n, 2^{\lceil \log_2(n) \rceil})$ with 0-bits. Omitting these trailing 0-bits ensures that the number of stored labels never exceeds the length of the original bitmap.

The presented modifications reduce the overall space usage, as shown in Figure C.7. In particular, the worst-case space consumption reduced from $3n-1$ to n bits, excluding the (small) metadata. We observe that in an optimized

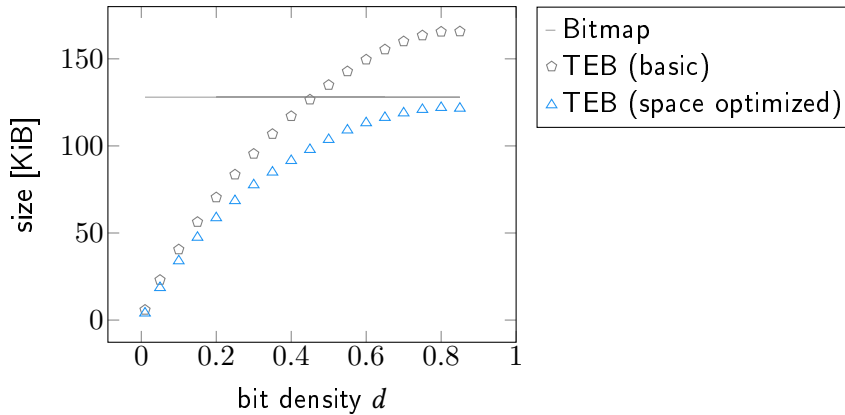


Figure C.7: Size comparison of basic and space optimized TEBs using a clustering factor of 8.

TEB the fraction of space occupied by the tree, the rank helper structure, and the labels is no longer fixed; compare Figures C.8a and C.8b. With sparse bitmaps, the labels occupy significantly less space. With denser bitmaps, on the other hand, we see that the fraction of space occupied by the tree structure decreases. Figure C.9 shows how the implicit tree nodes and the implicit labels optimizations contribute to the space savings. The implicit labels optimization is most effective with sparse bitmaps and the implicit tree node optimization, on the other hand, favors denser bitmaps.

An important implication is that the space optimizations balance the upper part of the tree structure, as the example in Figure C.6 has shown. The partially pruned tree in Figure C.6b is perfectly balanced until level two, whereas the fully pruned tree in Figure C.6a is only perfectly balanced until level one. Thus in general, the tree can be split into an upper balanced and a lower imbalanced part. This property allows for the reduction of the cost of navigational operations. We exploit the fact that within a perfect binary tree we can directly address the individual tree nodes, i.e., without computing ranks. If the number of the upper *perfect levels* is known, these levels of the tree can be logically cut off, and only the remaining sub-trees need to be considered. In our case, we can directly compute the number of perfect levels u based on the number of implicit inner nodes c that are already known when the space optimizations have been applied: $u := \lfloor \log_2(c+1) \rfloor + 1$. The corresponding node IDs for the last perfect level are within the range $[t_{\text{begin}}, t_{\text{end}})$, with $t_{\text{begin}} := 2^{u-1} - 1$ and $t_{\text{end}} := 2^u - 1$. Each of these nodes, or the sub-trees rooted at these nodes, respectively, span a range of length $2^{\log_2(n)-u-1}$ in the original bitmap. Thus, it can be considered as a uniform partitioning scheme, similar to the one used in Roaring Bitmaps, but with the major difference that the partition size is chosen adaptively.

The number of perfect tree levels is correlated with the effectiveness of the tree-based compression. The less effective the compression, the larger the number of perfect levels, and vice versa. In worst case, the entire tree is

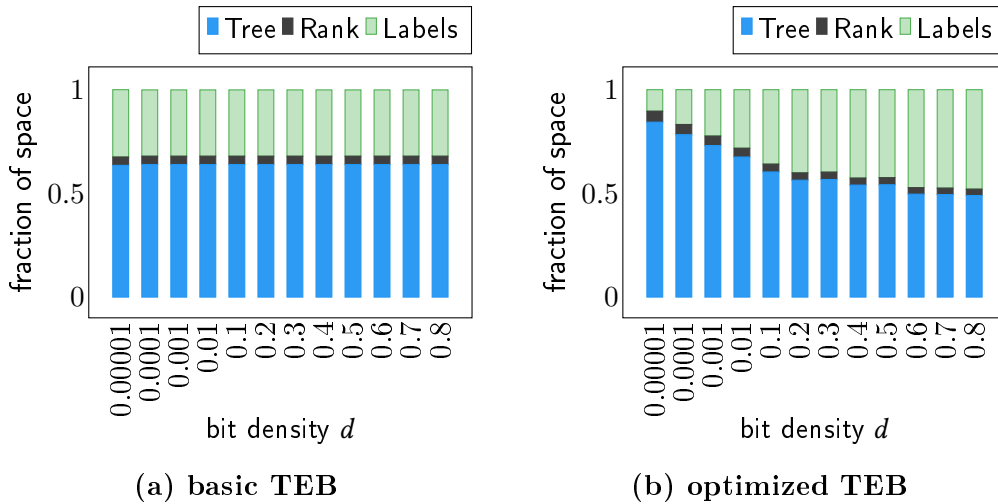


Figure C.8: The fraction of space occupied by the tree, the rank helper structure, and the labels.

implicit and the number of perfect levels corresponds to the tree height. In other words, TEBs gradually degrade into literal bitmaps, but unlike Roaring and WAH, TEBs remain homogeneous and do not need to switch between different encodings or representations.

C.3 Operations

In this section, we describe the operations supported by TEB. Fundamentally, a TEB supports two access methods: (i) a point lookup and (ii) a 1-run iterator. High-level functionalities, like decompressing a bitmap or logical operations are implemented on top of the 1-run iterator.

C.3.1 Point Lookup

A point lookup is a straightforward operation that navigates downward the tree until a leaf node is reached. The index k of the bit to look up thereby specifies the path to take within the tree. For performance reasons, the downward navigation starts at the last perfect tree level rather than at the root node. The details are shown in Algorithm 1.

C.3.2 Run Iterator

The iterator interface allows for efficient iteration over a TEB. Unlike the iterators implemented in Roaring and WAH, the TEB iterator does not iterate over the individual 1-bits, instead it iterates over the 1-runs of a bitmap. A 1-run is thereby represented as two integer values $\langle begin, end \rangle$, pointing to the position of the first 1-bit and to the position one past the last 1-bit. The iterator traverses the tree in depth-first left-to-right order. To navigate

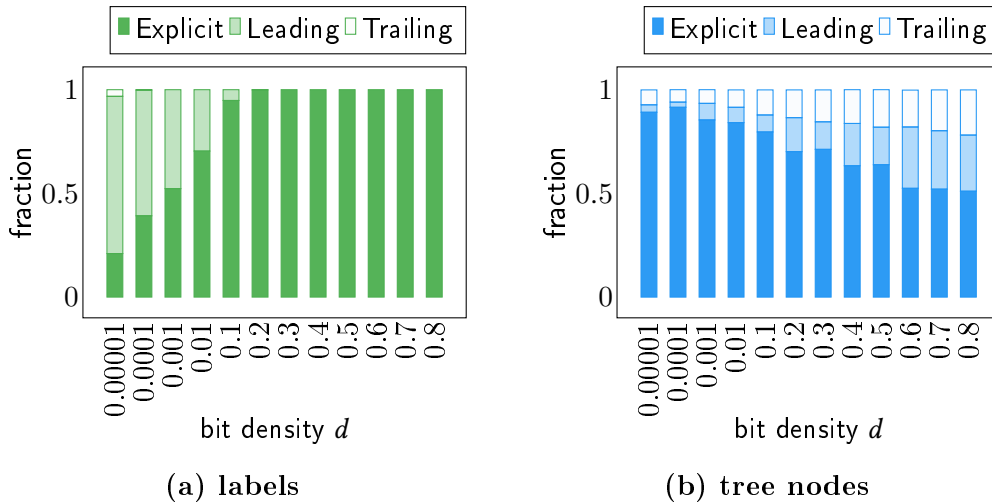


Figure C.9: The fraction of explicitly stored labels (a) and tree nodes (b).

Algorithm 1: Point lookup

Input : The bit index k to test
Returns: true if the k^{th} bit is set, false otherwise

```

// Determine the tree node at the last perfect level.
 $t_{\text{offset}} \leftarrow k \gg (\text{tree\_height} - \text{perfect\_levels} - 1)$ 
 $i \leftarrow t_{\text{begin}} + t_{\text{offset}}$ 
 $j \leftarrow \text{tree\_height} - 1 - \text{perfect\_levels} - 1$ 
// Navigate downwards until a leaf node is observed.
while  $i$  is an inner node do
    |  $\text{direction} \leftarrow \text{extract } j^{\text{th}} \text{ bit from } k$ 
    |  $i \leftarrow \text{left-child}(i) + \text{direction}$ 
    |  $j \leftarrow j - 1$ 
end
return label( $i$ )

```

down the tree, the functions `left-child()` and `right-child()` are used, as described in Section C.2.2. To navigate upwards, the iterator makes use of a small stack that is populated during downward navigation. Other data structures like SuRF [188] implement upwards navigation using the *select* primitive, the counterpart to rank. For TEB, we prefer a classic stack-based approach as it is significantly faster in practice and saves space.

During tree traversal, the iterator needs to keep track of its position (and level) within the tree structure. This information is required to determine the start index and length of a 1-run when the iterator reaches a leaf node with label 1 and thus needs to produce an output. The iterator therefore maintains a *path* variable that encodes the path from the root to the current node using a single integer. The initial (and minimum) value of the path variable p is 1. During downwards navigation, a 0-bit is shifted in when navigating to

Algorithm 2: Forward the iterator to the next 1-run.

```

while  $t < t_{\text{end}}$  do
  while  $stack$  is not empty do
    // Pop tree node  $i$  and its path  $p$  from the stack.
     $\langle i, p \rangle \leftarrow stack.pop()$ 
    while  $i$  is an inner node do
      // Push right child on stack and go to left child.
       $i \leftarrow \text{left-child}(i)$ 
       $p \leftarrow p \ll 1$ 
       $stack.push(\langle i + 1, p$ 
    end
    // Reached a leaf node.
    if  $\text{label}(i) = 0$  then continue
    // Found a 1-run. Update the iterator state.
     $level \leftarrow \text{sizeof}(p) \cdot 8 - 1 - \text{lzcount}(p)$ 
     $begin \leftarrow (p \oplus (1 \ll level)) \ll (tree\_height - level)$ 
     $end \leftarrow begin + (n \gg level)$ 
    return
  end
   $t \leftarrow t + 1$ 
   $p \leftarrow (t - t_{\text{begin}})$ 
end
 $begin \leftarrow end \leftarrow n$  // Reached the end.
return

```

the left child $p := (p \ll 1)$ and a 1-bit when navigating to the right child $p := (p \ll 1) \mid 1$. The index of the most significant 1-bit (the *sentinel bit*) indicates the level of the corresponding tree node:

$$\text{level}(p) := \text{sizeof}(p) \cdot 8 - 1 - \text{lzcount}(p)$$

where $\text{sizeof}(p)$ refers to the size of the variable p in bytes and $\text{lzcount}(p)$ to the number of leading zeros in p . A tree node that is identified by its path p then represents a run that starts at position

$$\text{pos}(p) := (p \oplus (1 \ll \text{level}(p))) \ll (tree_height - \text{level}(p))$$

with length

$$\text{length}(p) := n \gg \text{level}(p) \cong 2^{\log_2(n) - \text{level}(p)}.$$

Similarly to the point lookup access method, the upper perfect levels of the tree are skipped. The iterator only considers the sub-trees rooted in $[t_{\text{begin}}, t_{\text{end}})$, as described in Section C.2.3. Algorithm 2 shows how the iterator is forwarded to the next 1-run.

As mentioned earlier, a time-critical operation is to *fast-forward* the iterator to a desired position, thereby skipping all set bits in between. Thanks to the

navigable tree structure, the operation can be performed in logarithmic time. Nevertheless, to achieve competitive performance in practice, we optimize the skip operation so that unnecessary navigation steps are avoided. The primary decision that is to be made is whether to (i) navigate up the tree to the common ancestor of the current and the destination node, and then downwards in the right sub-tree to the desired position, or (ii) start at the root node (or at the corresponding tree node in the last perfect level) and navigate only downwards until the desired position has been reached. Depending on the source and destination nodes, one option might be more efficient than the other. The two options may differ in the number of required navigation steps. But we also need to consider that navigating upwards is less costly in terms of issued CPU instructions than navigating downwards. The asymmetrical costs are mostly caused by the rank primitive, which is significantly more costly than accessing the stack. We experimentally determined that a downward step is approximately $9\times$ more expensive than an upward step (~ 55 cycles vs. ~ 6 cycles).

Our decision logic works as follows: We start with a fast test to determine whether the destination position is outside of the current sub-tree:

$$pos \gg (h - u - 1) \neq to_pos \gg (h - u - 1)$$

where h refers to the tree height and u to the number of perfect levels. If the expression evaluates to true, we can directly go to the corresponding node at the last perfect level and navigate downwards until the desired position has been reached. Otherwise, if the destination node is within the current sub-tree, we (i) determine the common ancestor node (ii), estimate the navigational costs for both options, and (iii) pick the cheaper path.

It is worth mentioning that an iterator with skip support is not the most efficient way to decompress (rather than intersect) a TEB. For these cases we provide an alternative iterator to which we refer to as *scan iterator*. Unlike the regular iterator, the scan iterator's seek function operates in $O(n)$, but it offers a significantly higher read throughput, as it (i) decodes the tree in batches and (ii) does not rely on the rank primitive to traverse the tree.

C.3.3 Tree Scan

In this section, we present a tree traversal algorithm that is optimized for modern x86 hardware. The algorithm takes a level-order encoded binary tree and iterates over all leaf nodes in left-to-right order. We refer to the algorithm as *tree scan*. The tree scan is the basic building block for the TEB scan iterator.

Generally speaking, navigating from one leaf node to the next one is a 3-step process: (i) navigate up the tree until a left child is observed, (ii) go to its right sibling, and (iii) walk down the tree to the leftmost leaf node. The key idea behind our solution is to have multiple lightweight bit iterators for the encoded tree structure T , one iterator per tree level, and then scan the bit sequence T in parallel. We denote the bit iterators in T as t_l with $0 \leq l < h$. Initially, all

Algorithm 3: Tree scan

```
p // The current path. Initially points to the leftmost leaf.
do
  // Produce an output, if the label of the current node is 1.
  ...
  // Walk upwards until a left child is found.
  up_steps ← tzcount(~p)
  last ← level(p) + 1
  p ← p >> up_steps
  p ← p
  // Walk downwards to the leftmost leaf in that sub-tree.
  down_steps ← tzcount(~(α >> level(p)))
  p ← p << down_steps
while not done
```

iterators point to the first bit in T at their corresponding level l . We expose the values each iterator points to as an integer value, denoted as α . The bits in $\alpha := b_{h-1} \dots b_1 b_0$ are populated with $b_l = *t_l$, where $*$ denotes the dereference operator. A path variable p identifies the position and the level within the tree, as described earlier. Initially, p points to the leftmost leaf node. Using the two values α and p , we can efficiently iterate over all leaf nodes in left-to-right order, cf., Algorithm 3. Thereby, p determines the number of upward steps and α determines the number of downward steps to perform in each iteration.

The bit iterators are implemented using the AVX-512 SIMD instruction set as follows. We use a 512-bit SIMD register to buffer the tree structure. The register is interpreted as 32×16 -bit integers, i.e., the register is split into 32 lanes. Thereby, each SIMD lane corresponds to a tree level. For each level we load up to 16 bits from the encoded tree T . For instance, Figure C.10 illustrates a buffer that contains the tree from Figure C.6b. To consume the buffered tree bit by bit, we use a second SIMD register to which we refer as *read mask*. The read mask again consists of 32 lanes, and a single bit is set within each lane. Initially, the least significant bit is set to 1. The position of that bit represents the current read position in the corresponding buffer lane. Thus, the read mask represents the state of all (up to) 32 lightweight bit iterators. The increment of an iterator is then implemented as a left shift of the corresponding lane. The implementation has the advantages that we can work with multiple iterators in parallel and that the most important operations can be performed in a single instruction. For instance, multiple iterators can be incremented using a single masked shift instruction (`_mm512_mask_slli_epi16`⁴) and all iterators can be dereferenced in parallel to retrieve the aforementioned α value; cf., Figure C.10.

The presented algorithm is used in the TEB scan iterator that is supposed to be used when efficient skip support is not required, e.g., when decompressing

⁴We refer the reader to the Intel Intrinsics Guide for more details on the SIMD instruction set architectures: <https://software.intel.com/sites/landingpage/IntrinsicsGuide>

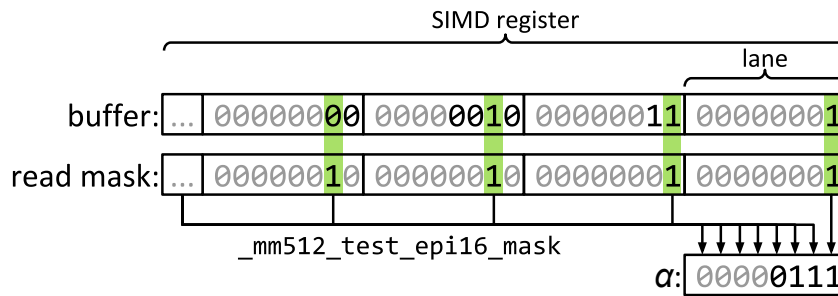


Figure C.10: AVX-512 allows for the instantiation of up to 32 lightweight bit iterators (one for each tree level) using only two SIMD registers: The first is used to buffer the encoded tree level by level and the second represents the iterators’ read positions.

an entire TEB. With regard to performance, the scan iterator benefits from the predictable memory access pattern, as well as from the reduced number of memory loads, due to buffering. However, a problem not mentioned above is that we need to know the start offset in T for each tree level. Unfortunately, determining these offsets is a linear time operation. Therefore, we store the offsets as part of the TEB metadata, which now is logarithmic in size. For brevity we have omitted some of the implementation details, such as how buffers are refilled and how labels are buffered and accessed; which works similarly to the buffering of the tree structure. We invite the interested reader to examine the source code of TEB⁵.

C.3.4 Logical Operations

As mentioned earlier, high-level functionality is implemented on top of the 1-run iterator. Operations like a bitwise AND are themselves implemented as iterators and can therefore be arbitrarily chained and combined to evaluate complex expressions. Algorithm 4, for instance, shows how two bitmaps are intersected using the iterator API. In contrast to the implementations in Roaring and WAH, the iterator approach does not produce a compressed bitmap. We think this is not a disadvantage because producing compressed intermediate results when evaluating complex compressions could harm performance. For instance, when bitmap indexes are used to evaluate multi-dimensional selection predicates, it is sufficient to identify the ranges (or pages) that contain qualifying tuples; an intermediate bitmap would be discarded afterwards anyhow.

⁵TEB source code: <https://db.in.tum.de/research/publications/#teb>

Algorithm 4: Next function of the AND iterator.

```
Input: Run iterators a and b.
while  $!(a.begin \neq n \wedge b.begin \neq n)$  do
    begin_max  $\leftarrow \max(a.begin, b.begin)$ 
    end_min  $\leftarrow \min(a.end, b.end)$ 
    overlap  $\leftarrow begin\_max < end\_min$ 
    if overlap then
        if  $a.end \leq b.end$  then a.next()
        if  $b.end \leq a.end$  then b.next()
        begin  $\leftarrow begin\_max$  // Update the iterator state.
        end  $\leftarrow end\_min$ 
        return
    else
        if  $a.end \leq b.end$  then a.skip_to(b.begin)
        else b.skip_to(a.begin)
    end
end
begin  $\leftarrow end \leftarrow n$  // Reached the end.
```

C.3.5 Updates

Data structure design in general is a trade-off between read, update, and memory overheads. The RUM conjecture [16] states that when optimizing (reducing) two of these overheads, it impairs the third one. TEBs are optimized for efficient read access and low memory consumption, and similarly to existing RLE-based compression schemes, the static nature of TEBs does not allow for in-place updates. In the following, we discuss various approaches that can be combined with TEBs to achieve updatability.

The naïve and costly way to support random updates is to decompress the bitmap, perform the update on the uncompressed representation, and (re-)compress it again afterwards. Prior work [17] proposed to reduce the update costs by staging updates in an auxiliary differential data structure and to apply these pending updates in batches, rather than one-by-one. Thereby, another compressed bitmap is used as a differential data structure. While this approach greatly reduces the number of decompression/compression cycles, it also causes redundancies (slightly higher memory consumption) and requires the differential data structure to be consulted (XORed) during read access.

Roaring bitmap applies a different strategy. Due to the fixed size partitioning, an update affects only a single container, rather than the entire bitmap. Thus, in worst-case, 2^{16} bits need to be re-compressed during updates. Updates can therefore be performed in constant time⁶, even though the constant is quite large. Nevertheless, the partition size has been chosen sufficiently small to fit in an L1 cache to enable efficient decompression/compression cycles.

⁶Assuming the corresponding containers reside in heap memory. Modifications to the serialized format would still be in linear time.

	WAH	EWAH	Concise	Roaring	TEB
CENSUS INCOME	3.4	3.3	2.9	2.6	2.1
CENSUS INCOME <small>(sorted)</small>	0.66	0.64	0.55	0.6	0.36
CENSUS 1881	34.4	33.8	25.6	15.1	12.6
CENSUS 1881 <small>(sorted)</small>	3.0	2.9	2.5	2.1	1.5
WEATHER	6.8	6.7	5.9	5.4	4.2
WEATHER <small>(sorted)</small>	0.55	0.54	0.43	0.34	0.26
WIKILEAKS	11.1	10.9	10.2	5.9	5.4
WIKILEAKS <small>(sorted)</small>	2.9	2.7	2.2	1.7	1.7

Table C.1: Space usage in bits per attribute value.

Both approaches can be used with TEBs. Partitioning could further be combined with differential updates so that a separate diff is maintained per partition. We will show in the later evaluation section that the combined approach offers the highest throughput regarding updates, with minor compromises regarding reads.

C.4 Experimental Analysis

In the following, we evaluate our approach with regard to its compression ratio and performance. We begin by using a number of real-world data sets before performing a detailed evaluation using synthetic data.

C.4.1 Real-World Data

We evaluate TEBs with bitmaps from bitmap indexes constructed from four real-world data sets that have been previously used in the experimental evaluation of Roaring Bitmaps [102]. The data sets, namely CENSUS INCOME, CENSUS 1881, WEATHER, and WIKILEAKS, come in two flavors: *as is* and *sorted*. The latter relies on a-priori sorting of the raw input data, which leads to significantly better compression ratios [103, 139, 104]. Following the prior work, we compress the individual bitmaps, 200 per data set, and report the average number of bits per attribute value. We compare TEB with Concise, EWAH, Roaring, and WAH. The results for Concise and EWAH are taken from [102]. We reproduced the results for Roaring with very minor differences with the sorted CENSUS 1881 and WIKILEAKS data. But we observed a higher discrepancy for WAH. Among our experiments, we observed a slightly higher space usage than reported earlier, except for CENSUS 1881 where we observed a significantly better compression ratio (34.4 vs. 43.8 bits per element). We attribute these discrepancies to the fact that we use a different implementation [5] of WAH. Please note that EWAH and WAH use 32-bit words; we omit the results for the 64-bit implementations, as those have a higher space consumption among all tested workloads.

	Rank LuT resolution [bits]					no
	64	128	256	512	2048	LuT
CENSUS 1881	1.10	0.95	0.87	0.83	0.81	0.80
CENSUS 1881 <small>(sorted)</small>	0.87	0.76	0.71	0.69	0.67	0.66
CENSUS INC.	0.93	0.86	0.82	0.81	0.79	0.79
CENSUS INC. <small>(sorted)</small>	0.76	0.66	0.62	0.60	0.58	0.58
WEATHER	0.93	0.84	0.80	0.77	0.76	0.75
WEATHER <small>(sorted)</small>	0.97	0.84	0.79	0.76	0.74	0.73
WIKILEAKS	1.18	1.02	0.95	0.91	0.89	0.88
WIKILEAKS <small>(sorted)</small>	1.25	1.11	1.04	1.01	0.98	0.98

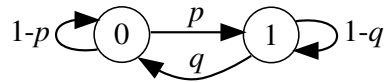
Table C.2: Relative size of TEB compared to Roaring
 $(\frac{\text{TEB size}}{\text{Roaring size}})$ for varying rank resolutions.

Table C.1 summarizes the experimental results. TEB offers the best compression ratios, except for the sorted WIKILEAKS data, where Roaring is slightly better (1.667 vs. 1.677 bits per element). TEB saves up to 22% space on unsorted data and up to 34.6% on sorted data compared to the second best compression technique, which in most cases is Roaring.

The rank lookup table (LuT) thereby accounts for 2.2% to 4.4% of the TEB size (3.7% geo. mean, among all real-world data sets). As mentioned earlier, changing the resolution of the LuT offers a space/time trade-off. A fine-grained LuT with one entry per 64 bit offers the best performance. We observe a 30% lower execution time for computing bitmap intersections. The memory overhead of the LuT thereby increases significantly to up to 27%, which almost cancels out the improvements in compression. Decreasing the LuT resolution to 2048 bits on the other hand reduces the TEB size by up to 2.8% but also causes the intersection time to increase by up to 10%. Table C.2 shows how the space consumption of TEBs changes for varying rank resolutions compared to Roaring. Throughout our experiments, we found that a 512-bit resolution offers a reasonable space/time trade-off, which we use as our default setting in the following. Nevertheless, it is noteworthy that the rank LuT could be omitted when TEBs are written to persistent storage, and could be recomputed on-the-fly when TEBs are loaded back into main memory, allowing one to save additional disk space and I/O (cf., rightmost column in Table C.2).

C.4.2 Synthetic Data

For an in-depth analysis we generate random bitmaps, where the individual 1-bits are either uniformly distributed or clustered. *Uniform* random bitmaps are random bitmaps where each bit is independently generated following an identical probability distribution [179], i.e, each bit is set with probability d . *Clustered* random bitmaps on the other hand are generated using a two-state Markov process



with the transition probabilities p and q set to

$$p := \frac{d}{(1-d) \cdot f}, \text{ and } q := \frac{1}{f}$$

with $0 < d < 1$ and $1 \leq f \leq n$. We make a minor change over the definition given in [179]; which is that we choose the initial state randomly with a probability of 0.5, whereas in [179] the initial state is ①, meaning that a randomly generated bitmap would always start with a 1-run.

We generate bitmaps of length $n = 2^{20}$ and report the averaged results over 10 independent experiments. We compare TEB with WAH [179], which is the most popular RLE-based bitmap compression scheme, and with Roaring Bitmap [102], which is the state-of-the-art with regard to performance and compression ratio. The thorough study of Wang et al. [171] found Roaring to be superior over other bitmap compression techniques such as Concise [48], WAH, EWAH [103], VALWAH [76], PLWAH [55], and SBH [89]. We therefore limit our evaluation to Roaring and WAH.

For the experiments we use FastBit [5, 176] v2.0.3, which provides a C++ implementation of WAH, and CRoaring [8] v0.2.60 (unity build), the official C/C++ implementation of Roaring Bitmap. The `dynamic_bitset` from the Boost C++ libraries [4] v1.67.0 is used for uncompressed bitmaps. We compile with GCC v8.3.0 (`-O3 -march=native`) and execute on an Intel Core i9-7900X CPU @ 4 GHz.

Compression

Uniform Bitmaps. In the following, we examine the compression ratios with uniform random bitmaps with varying bit densities. The results in Figure C.11 show that TEB and Roaring are on par in the case of sparse bitmaps ($d < 0.005$). With an increasing d , TEB shows the lowest space usage. When more than 13% of the bitmap is populated, TEB is no longer able to compress; Roaring and WAH already stop at 5%. With dense bitmaps ($0.5 < d \leq 1$) we observed symmetrical results for TEB and WAH, only Roaring requires a density of more than 97% for the compression to work again (rather than 95%). This is attributed to the different containers being used in Roaring, and the fact that Roaring encodes 0-runs and 1-runs differently, which is in contrast to TEB and WAH.

Clustered Bitmaps. With our third experiment, we examine the compression ratios with clustered bitmaps, using varying bit densities d and clustering factors f . We start with an exploration of the space spanned by d and f . Thereby, we consider the ranges $0.0001 \leq d < 1$ and $1 \leq f \leq n$. We make the following observations:

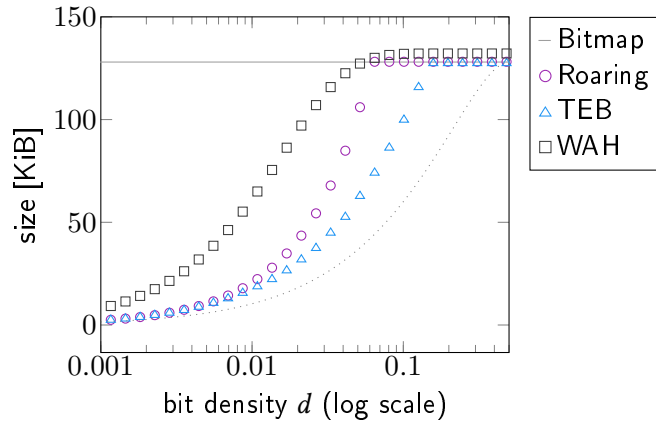


Figure C.11: Size of uniform random bitmaps with varying bit densities. The dotted line refers to the information theoretic minimum.

- When the input bitmaps are very sparsely populated or exhibit a strong clustering, all bitmap compression techniques under test perform well. In the dotted area (·) in Figure C.12, the compressed bitmaps occupy less than 1% of the space of the uncompressed bitmap, irrespective from the employed compression scheme.
- TEB offers better compression ratios than WAH throughout all measurements; and only in some rare cases does WAH compress slightly better than Roaring.
- When comparing TEB and Roaring, TEB does not always offer the best compression ratios. However, in these cases, the differences in size are marginal. The largest difference in size we observed throughout all experiments is 1.6% of the original bitmap size. In the area marked with ◊ in Figure C.12, TEB and Roaring perform similarly.
- TEB in contrast, shows significantly higher compression ratios with denser bitmaps and bitmaps with lower clustering, cf. the area marked with △ in Figure C.12. In comparison to Roaring, we observed a difference in size of up to 56% of the plain bitmap size, in favor of TEB. Figure C.13 shows a qualitative side-by-side comparison.

Figure C.14 gives a detailed view on how the size of the compressed bitmaps change for varying d and fixed f . Figure C.14a shows that the TEB approach is able to exploit short 1-runs in sparse bitmaps, resulting in up to ~50% space savings over Roaring. With a moderate clustering, as shown in Figure C.14b, our approach is also able to compress dense bitmaps. Figure C.14c, on the other hand, reveals that our approach has a slightly higher space usage than

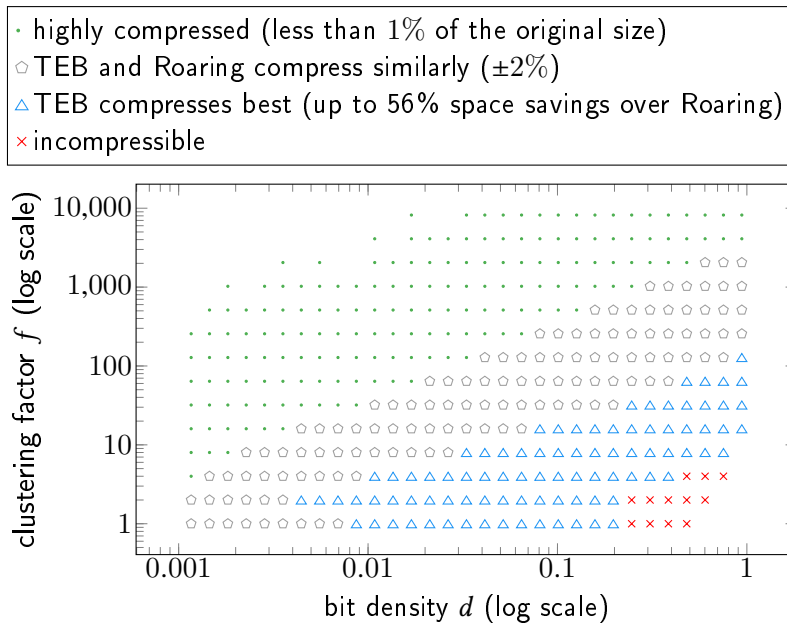


Figure C.12: Summary of our findings when compressing clustered bitmaps.

Roaring with strongly clustered bitmaps, which implies that Roaring can encode longer runs more space efficiently.

Figure C.15 illustrates how f affects the compression ratios. Figures C.15a and C.15b show that already a slight clustering can lead to significant space savings with TEB. Roaring requires a significantly higher clustering to be competitive. With sparser bitmaps, TEB falls slightly behind Roaring (see Figure C.15c), whereas WAH cannot compete.

Performance

In the following, we evaluate the read and update performance of TEB, and show how it compares to Roaring and WAH.

Read Access. We first investigate the read (or decompression) throughput. We thereby iterate over all 1-runs of a bitmap and measure the duration in wall-clock time. In our initial performance experiment, we again explore the space spanned by d and f . Thereby we observe that an uncompressed bitmap performs better than the compressed formats when $16 \leq f \leq 128$ and $0.01 \leq d < 1$. It should be noted that the `dynamic_bitset` implementation, which we use for uncompressed bitmaps, is very straightforward and does not include any hardware specific optimizations. Thus, we expect a performance-optimized implementation to dominate an even larger space. When we consider only the performance of compressed bitmaps, we observe that the clustering mostly determines the best performing compression technique: Roaring is dominant when $f \leq 16$, followed by WAH until f is approximately 128. TEB requires an evenly higher clustering ($f > 128$) to outperform Roaring and WAH.

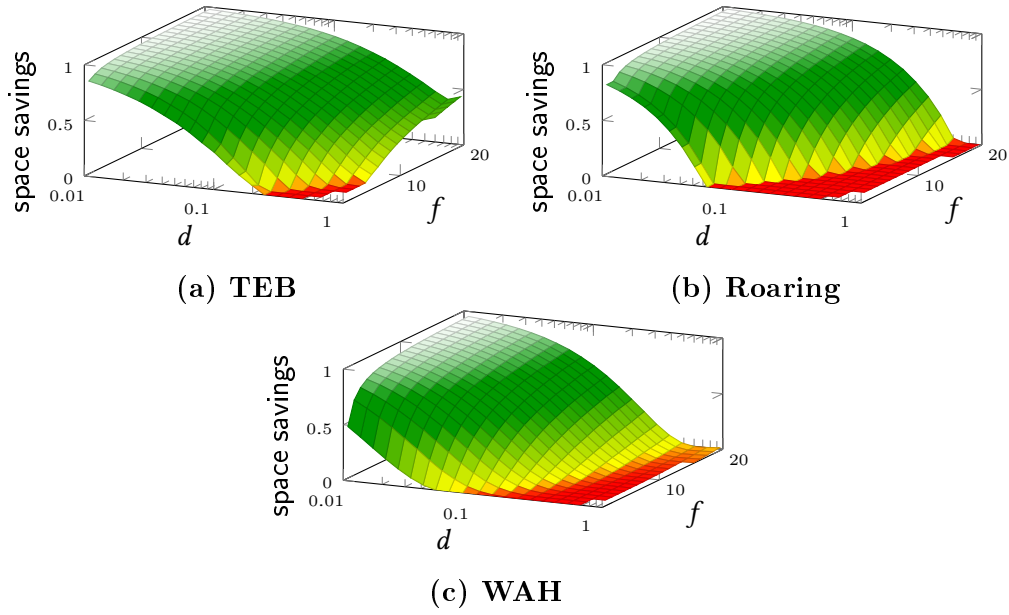


Figure C.13: Space savings $\left(1 - \frac{\text{compressed size}}{\text{uncompressed size}}\right)$ for varying d and f .

In Figure C.16, we compare the performance for reasonable values of d and f , which we expect to occur in practice. We fixed d to $\{0.25, 0.1, 0.01\}$ and varied f within the range $[1, 20]$. We observe that the time to read the bitmap decreases with an increasing f , which is due to the smaller size of the input and due to less branching; the higher f is, the lower the number of 1-runs to iterate over. Figure C.16a, with d set to 0.25, shows that TEB offers a similar performance as WAH, and that both are close to the performance of Roaring. Still, a plain bitmap performs best in most cases. The outliers at $f = 1$ and $f = 2$ are due to specialized code paths that are taken when the bitmaps are not compressed (or just barely compressed). In the Figures C.16b and C.16c, with bit densities reduced to 0.1 and 0.01, we observe that the absolute time to read a bitmap decreases for all implementations under test (note the different y-axis scales), but also that TEB falls behind relative to Roaring and WAH, indicating that the average cost per 1-run increases with lower d . Naturally, this is an expected result, as lower bit densities result in sparse and imbalanced trees, which in turn increases the number of tree levels that need to be traversed (cf., Section C.2).

In our second experiment, we evaluate the effectiveness of efficient tree navigations within logical operations. We intersect (bitwise AND) two bitmaps with different characteristics. The density and the clustering in the first bitmap is thereby fixed to $d_1 = 0.01$ and $f_1 = 8$. In Figure C.17a, we fix the clustering in the second bitmap to $f_2 = 4$ and vary the density d_2 . We observe that the density of the second bitmap only has a minor impact on the overall intersection time, except for WAH. The intersection of uncompressed bitmaps, with constant time random access, is fastest in this setting. Roaring takes $\sim 1.5\times$ the time of the plain bitmap intersection, and TEB $\sim 1.9\times$ the time of Roaring.

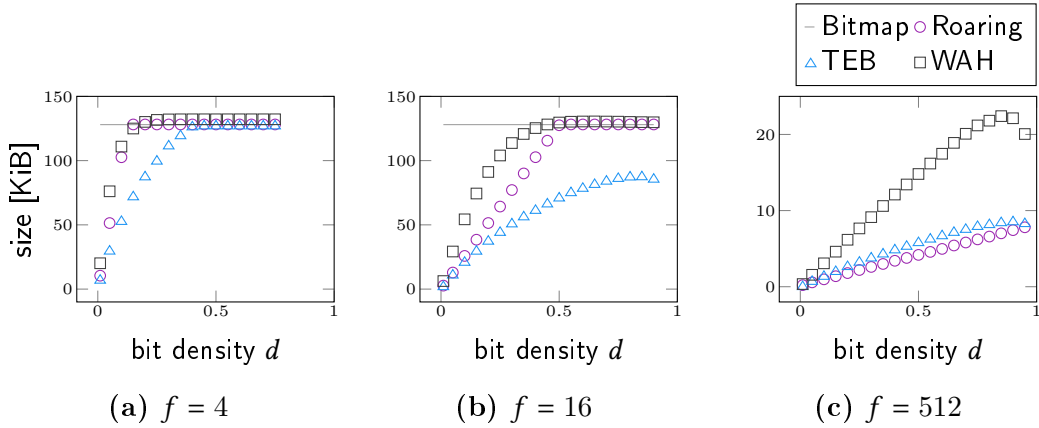


Figure C.14: Compressed bitmap size for varying bit densities and fixed clustering factors.

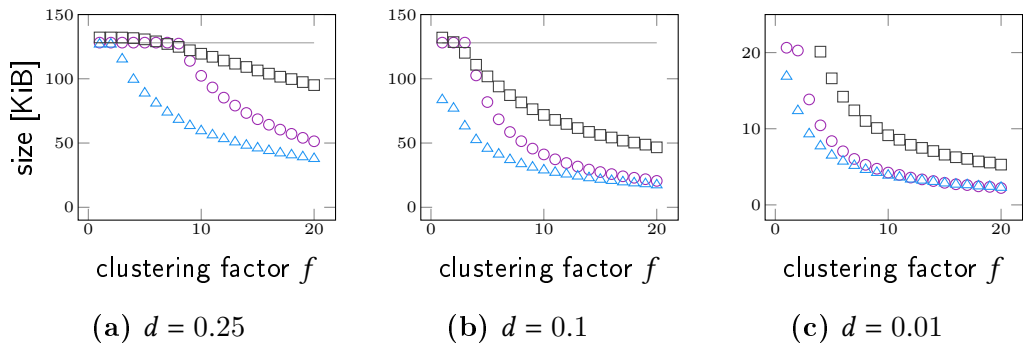


Figure C.15: Compressed bitmap size for varying clustering factors and fixed bit densities.

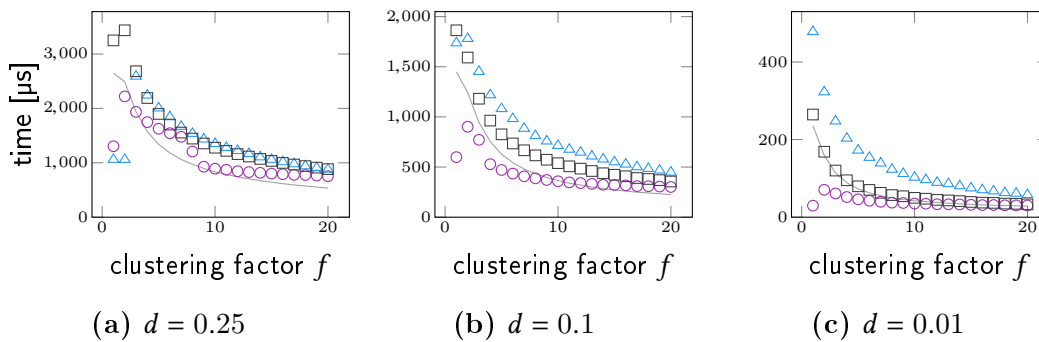


Figure C.16: Read performance for varying clustering factors and fixed bit densities.

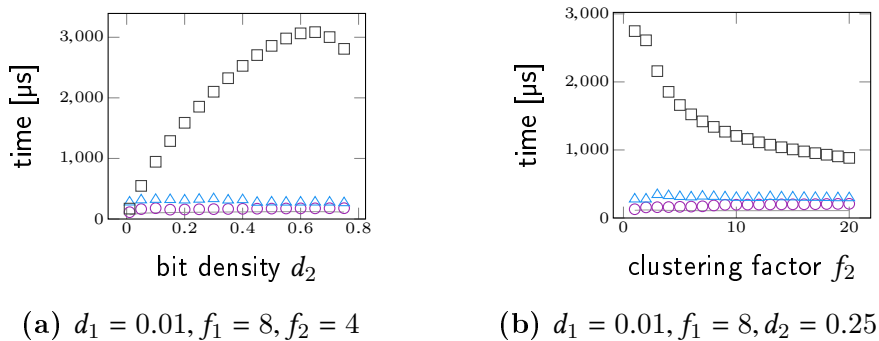


Figure C.17: Intersection performance.

Compression method	avg. time per update [ns]	
	non-partitioned	partitioned
TEB	599	218
Roaring	480* / 574	121* / 216
WAH	17634	794

* using the in-memory layout (non serialized)

Table C.3: The average time to apply an update.

In Figure C.17b, we fix the density of the second bitmap to $d_2 = 0.25$ and vary f_2 . Again, only WAH is sensitive to the varying clustering factor and thus to the size of the second bitmap. On average, Roaring needs $\sim 1.8\times$ the time of the plain bitmap intersection, and TEB $\sim 1.6\times$ the time of Roaring.

Differential Updates. In our final experiments, we extend TEB and the other bitmap compression techniques under test by a differential data structure and evaluate the update performance. Our experiments revealed that WAH is not well suited as a differential data structure. We found that Roaring significantly outperforms WAH in that regard, because (i) the partitioned in-memory layout of Roaring offers significantly faster updates and (ii) the better compression ratios of Roaring reduce the amount of memory occupied by pending updates. We therefore use Roaring as a differential data structure in the following and omit the results for WAH.

We measure the update throughput by applying 100k point updates to a compressed bitmap (with $n=2^{20}$, $d=0.1$, $f=8$) and report the average execution time. The number of pending updates is limited to 20k; i.e., a merge is triggered when this threshold is reached. Further, we examine how partitioning affects the execution time of point updates. We partition the bitmap into chunks of 2^{16} bits, whereas each chunk has its own diff. The results in Table C.3 show that TEB and Roaring are on par, whereas WAH is several times slower. WAH suffers from the linear time complexity of point lookups

that are involved with updates. Data partitioning helps to reduce the access latency significantly, but the average time of an update is still $3.6\times$ higher. The performance of Roaring on the other hand could be improved by using its in-memory layout and its specialized XOR implementations for the individual container combinations (cf., the results marked with * in Table C.3). The optimization is enabled by the fact that both the value bitmap and the differential bitmap are Roaring bitmaps. In a pure in-memory setting, Roaring therefore outperforms TEB by up to $1.8\times$ and WAH by more than $6\times$ in terms of update latency (in the partitioned case).

Pending updates naturally impair read latency. We observed a 30% penalty for TEB and Roaring with 20k pending updates (20% with WAH), irrespective of partitioning. For more general information on the trade-offs involved with differential updates, we refer the reader to UpBit [17].

C.5 Related Work

Throughout the paper, we already covered the related work regarding bitmap compression techniques [14, 179, 48, 103, 76, 102, 89, 55, 17, 171], except for the HICAMP bitmap [170] which is designed for a special kind of memory system [44]. In the following, we discuss other related work.

Bitmap Indexes. Bitmap indexes and bitmap compression are orthogonal topics, as bitmap indexes may also be constructed with verbatim bitmaps. However, in practice, compression is commonly used to reduce space consumption and to improve query performance. Thus, the term bitmap index often refers to a *compressed* bitmap index. Compression, however, is just one aspect of a bitmap index. Other techniques that are involved when a bitmap index is constructed are (i) *binning* [94, 182, 183] which groups multiple attribute values together and (ii) *encoding* [36, 37, 133] which translates the bins into a set of bitmaps [180]. Thereby, an encoding scheme is chosen that best supports the query workload. Common encodings are equality encoding, range encoding and interval encoding, whereas the latter two allow for arbitrary range queries by accessing at most two bitmaps. Optionally, an attribute value may be *decomposed* into multiple components that are individually assigned to bins afterwards. A single attribute value may therefore map to multiple bins. An extreme case is the bit-sliced index [133, 147], where the attribute values are decomposed bit-by-bit, and the number of bins (and bitmaps) is equal to the bit-width of the attribute.

Binning, encoding, and decomposition influence the characteristics of the individual bitmaps [180] of an index. Consequently, they affect the overall index size and eventually the query performance [82, 178]. A thorough evaluation of TEBs within the large design space of bitmap indexes is therefore beyond the scope of this work.

Succinct Data Structures. The space efficiency of TEBs is founded on the idea of mapping tree nodes to integer values [99] and the foundational work on succinctly encoded binary trees [81] that efficiently support the necessary navigational operations using the rank and select primitives. Both primitives require a helper structure to lower the time complexity of tree navigations from linear to constant time. Several implementations have been proposed [70, 126, 69, 190, 169] to achieve the performance of pointer-based tree structures. A key to success, in terms of performance, was the introduction of the population count instruction, which unfortunately was quite late in widespread x86 processors (AMD 2007, Intel 2008). Over the years, other succinct tree encodings have been proposed [123, 122, 46, 145, 21, 52] that support a richer set of operations or being updatable [124]; both, however, would incur higher space consumption and/or lower performance with TEB.

Lightweight Indexing. Space-efficient secondary 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 skipping (i) blocks of tuples [3, 184, 153, 154, 118, 12], (ii) scan ranges within blocks [96], or (iii) (parts of) individual tuples [110, 143, 109, 66, 77]. Other index structures were designed to support specific kinds of queries, e.g., queries with a LIMIT clause [87], or for specific kinds of data, e.g., observational data [173]. Most of these index structures rely on lightweight statistical data that is easy to maintain and query. The more heavyweight approaches either store approximations of the indexed columns [153, 77] or even require a different storage layout [110, 66].

C.6 Conclusion

The Tree-Encoded Bitmap (TEB) is a novel approach for compressing bitmaps. Its tree-based compression algorithm maps 0- or 1-runs of various lengths to binary tree nodes, where the depth of a node implicitly determines its run length. The resulting tree structure is then encoded using a succinct physical data structure that supports logarithmic access time and therefore allows for efficient logical operations (such as intersections) on compressed data. We experimentally showed that TEB saves considerable space compared to other compressed bitmap formats—in particular at higher bit densities, i.e., those cases where memory consumption would otherwise be fairly high. In terms of access speed, TEB is quite fast for intersection operations: almost as fast as the competing approach Roaring, and much faster than WAH. In the data distributions where TEB is strongest in saving space, its raw scan performance is also close to Roaring. As such, TEB encoded chunks could also be used as a worthwhile addition to the adaptive Roaring approach, significantly improving compression in the most difficult data distributions, while preserving performance.

Acknowledgements. This work was supported by the DFG project KE401/22.

Bibliography

- [1] 7-Zip LZMA Benchmark website. <https://www.7-cpu.com/>. [Online; accessed 07-Feb-2020].
- [2] Apache Arrow. <https://arrow.apache.org/>.
- [3] Block Range Index (BRIN) in PostgreSQL. <https://www.postgresql.org/docs/11/brin.html>. [Online; accessed 01-Jul-2019].
- [4] Boost C++ Libraries. <https://www.boost.org/>. [Online; accessed 04-Jun-2019].
- [5] FastBit: An Efficient Compressed Bitmap Index Technology. <https://sdm.lbl.gov/fastbit/>. [Online; accessed 27-May-2019].
- [6] *MonetDB GeoSpatial*. <https://www.monetdb.org/Documentation/Extensions/GIS>.
- [7] Official Roaring Bitmap website. <https://roaringbitmap.org>. [Online; accessed 27-May-2019].
- [8] Roaring Bitmap. <https://github.com/RoaringBitmap/RoaringBitmap>. [Online; accessed 27-May-2019].
- [9] <https://stackoverflow.com/questions/36932240/avx2-what-is-the-most-efficient-way-to-pack-left-based-on-a-mask>, 2016.
- [10] Advanced Micro Devices, Inc. *Software Optimization Guide for AMD Family 17h Processors (rev. 3.00)*. 2017.
- [11] M. Albutiu, A. Kemper, and T. Neumann. Massively parallel sort-merge joins in main memory multi-core database systems. *PVLDB*, 5(10):1064–1075, 2012.
- [12] K. Alexiou, D. Kossmann, and P. Larson. Adaptive range filters for cold data: Avoiding trips to siberia. *PVLDB*, 6(14):1714–1725, 2013.
- [13] P. S. Almeida, C. Baquero, N. M. Prego, and D. Hutchison. Scalable bloom filters. *Inf. Process. Lett.*, 101(6):255–261, 2007.
- [14] G. Antoshenkov. Byte-aligned bitmap compression. In *Proceedings DCC '95 Data Compression Conference*, pages 476–, March 1995.

- [15] M. Athanassoulis and A. Ailamaki. Bf-tree: Approximate tree indexing. *PVLDB*, 7(14):1881–1892, 2014.
- [16] M. Athanassoulis, M. S. Kester, L. M. Maas, R. Stoica, S. Idreos, A. Ailamaki, and M. Callaghan. Designing access methods: The RUM conjecture. In *Proceedings of the 19th International Conference on Extending Database Technology, EDBT 2016, Bordeaux, France, March 15-16, 2016, Bordeaux, France, March 15-16, 2016.*, pages 461–466, 2016.
- [17] M. Athanassoulis, Z. Yan, and S. Idreos. Upbit: Scalable in-memory updatable bitmap indexing. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 1319–1332, 2016.
- [18] C. Balkesen, G. Alonso, J. Teubner, and M. T. Özsu. Multi-core, main-memory joins: Sort vs. hash revisited. *PVLDB*, 7(1):85–96, 2013.
- [19] C. Balkesen, J. Teubner, G. Alonso, and M. T. Özsu. Main-memory hash joins on multi-core cpus: Tuning to the underlying hardware. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, pages 362–373, 2013.
- [20] 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.
- [21] D. Benoit, E. D. Demaine, J. I. Munro, R. Raman, V. Raman, and S. S. Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, 2005.
- [22] S. Blanas, Y. Li, and J. M. Patel. Design and evaluation of main memory hash join algorithms for multi-core cpus. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*, pages 37–48, 2011.
- [23] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.
- [24] P. A. Boncz, S. Manegold, and M. L. Kersten. 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*, pages 54–65, 1999.
- [25] 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.

- [26] P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR 2005, Second Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2005, Online Proceedings*, pages 225–237, 2005.
- [27] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese. Beyond bloom filters: from approximate membership checks to approximate state machines. In *Proceedings of the ACM SIGCOMM 2006 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, Pisa, Italy, September 11-15, 2006*, pages 315–326, 2006.
- [28] 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.
- [29] K. Bratbergsengen. Hashing methods and relational algebra operations. In *Tenth International Conference on Very Large Data Bases, August 27-31, 1984, Singapore, Proceedings*, pages 323–333, 1984.
- [30] 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.
- [31] A. D. Breslow and N. Jayasena. Morton filters: fast, compressed sparse cuckoo filters. *VLDB J.*, 29(2-3):731–754, 2020.
- [32] A. Z. Broder and M. Mitzenmacher. Network applications of bloom filters: A survey. *Internet Mathematics*, 1(4):485–509, 2003.
- [33] M. Cain and K. Milligan. IBM DB2 for i indexing methods and strategies. IBM White Paper, 2011.
- [34] S. Chambi, D. Lemire, R. Godin, K. Boukhalfa, C. R. Allen, and F. Yang. Optimizing druid with roaring bitmaps. In *Proceedings of the 20th International Database Engineering & Applications Symposium, IDEAS 2016, Montreal, QC, Canada, July 11-13, 2016*, pages 77–86, 2016.
- [35] S. Chambi, D. Lemire, O. Kaser, and R. Godin. Better bitmap performance with roaring bitmaps. *CoRR*, abs/1402.6407, 2014.
- [36] C. Y. Chan and Y. E. Ioannidis. Bitmap index design and evaluation. In *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA.*, pages 355–366, 1998.

- [37] C. Y. Chan and Y. E. Ioannidis. An efficient bitmap encoding scheme for selection queries. In *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadelphia, Pennsylvania, USA.*, pages 215–226, 1999.
- [38] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A distributed storage system for structured data (awarded best paper!). In *7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA*, pages 205–218, 2006.
- [39] D. X. Charles and K. Chellapilla. Bloomier filters: A second look. In *Algorithms - ESA 2008, 16th Annual European Symposium, Karlsruhe, Germany, September 15-17, 2008. Proceedings*, pages 259–270, 2008.
- [40] B. Chattopadhyay, P. Dutta, W. Liu, O. Tinn, A. McCormick, A. Mokashi, P. Harvey, H. Gonzalez, D. Lomax, S. Mittal, R. A. Ebenstein, N. Mikhaylin, H. ching Lee, X. Zhao, G. Xu, L. A. Perez, F. Shahmohammadi, T. Bui, N. McKay, V. Lychagina, and B. Elliott. Procella: Unifying serving and analytical data at youtube. *PVLDB*, 12(12):2022–2034, 2019.
- [41] S. Chaudhuri and U. Dayal. An overview of data warehousing and OLAP technology. *SIGMOD Record*, 26(1):65–74, 1997.
- [42] 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.
- [43] H. Chen, L. Liao, H. Jin, and J. Wu. The dynamic cuckoo filter. In *25th IEEE International Conference on Network Protocols, ICNP 2017, Toronto, ON, Canada, October 10-13, 2017*, pages 1–10, 2017.
- [44] D. R. Cheriton, A. Firoozshahian, A. Solomatnikov, J. P. Stevenson, and O. Azizi. HICAMP: architectural support for efficient concurrency-safe shared structured data access. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2012, London, UK, March 3-7, 2012*, pages 287–300, 2012.
- [45] J. Chhugani, A. D. Nguyen, V. W. Lee, W. Macy, M. Hagog, Y. Chen, A. Baransi, S. Kumar, and P. Dubey. Efficient implementation of sorting on multi-core SIMD CPU architecture. *PVLDB*, 1(2):1313–1324, 2008.
- [46] D. R. Clark and J. I. Munro. Efficient Suffix Trees on Secondary Storage. volume 96 of *SODA '96*, pages 383–391, USA, 1996. Society for Industrial and Applied Mathematics.

- [47] 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.
- [48] A. Colantonio and R. D. Pietro. Concise: Compressed 'n' composable integer set. *Inf. Process. Lett.*, 110(16):644–650, 2010.
- [49] D. R. Cutting and J. O. Pedersen. Optimizations for dynamic inverted index maintenance. In *SIGIR'90, 13th International Conference on Research and Development in Information Retrieval, Brussels, Belgium, 5-7 September 1990, Proceedings*, pages 405–411, 1990.
- [50] S. E. Czerwinski, B. Y. Zhao, T. D. Hodes, A. D. Joseph, and R. H. Katz. An architecture for a secure service discovery service. In *MOBICOM '99, The Fifth Annual ACM/IEEE International Conference on Mobile Computing and Networking, Seattle, Washington, USA, August 15-19, 1999*, pages 24–35, 1999.
- [51] B. Dageville, T. Cruanes, M. Zukowski, V. Antonov, A. Avanes, J. Bock, J. Claybaugh, D. Engovatov, M. Hentschel, J. Huang, A. W. Lee, A. Motivala, A. Q. Munir, S. Pelley, P. Povinec, G. Rahn, S. Triantafyllis, and P. Unterbrunner. The snowflake elastic data warehouse. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 215–226, 2016.
- [52] P. Davoodi, R. Raman, and S. R. Satti. On succinct representations of binary trees. *Mathematics in Computer Science*, 11(2):177–189, 2017.
- [53] 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.
- [54] N. Dayan, M. Athanassoulis, and S. Idreos. Optimal bloom filters and adaptive merging for LSM-Trees. *ACM Trans. Database Syst.*, 43(4):16:1–16:48, 2018.
- [55] F. Delière and T. B. Pedersen. Position list word aligned hybrid: optimizing space and performance for compressed bitmaps. In *EDBT 2010, 13th International Conference on Extending Database Technology, Lausanne, Switzerland, March 22-26, 2010, Proceedings*, pages 228–239, 2010.
- [56] F. Deng and D. Rafiei. Approximately detecting duplicates for streaming data using stable bloom filters. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, pages 25–36, 2006.

- [57] H. Deshmukh, B. Sundarmurthy, and J. M. Patel. To pipeline or not to pipeline, that is the question. *CoRR*, abs/2002.00866, 2020.
- [58] A. D. Desk. Tiff 6.0 specification, 1992.
- [59] M. Dietzfelbinger and R. Pagh. Succinct data structures for retrieval and approximate membership (extended abstract). In *Automata, Languages and Programming, 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part I: Tack A: Algorithms, Automata, Complexity, and Games*, pages 385–396, 2008.
- [60] 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.
- [61] G. Einziger and R. Friedman. TinySet - An access efficient self adjusting bloom filter construction. *IEEE/ACM Trans. Netw.*, 25(4):2295–2307, 2017.
- [62] O. Erdogan and P. Cao. Hash-av: fast virus signature scanning by cache-resident filters. *IJSN*, 2(1/2):50–59, 2007.
- [63] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher. Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International Conference on Emerging Networking Experiments and Technologies, CoNEXT '14*, pages 75–88, New York, NY, USA, 2014. ACM.
- [64] L. Fan, P. Cao, J. M. Almeida, and A. Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Trans. Netw.*, 8(3):281–293, 2000.
- [65] W. Feng, D. D. Kandlur, D. Saha, and K. G. Shin. Stochastic fair blue: A queue management algorithm for enforcing fairness. In *Proceedings IEEE INFOCOM 2001, The Conference on Computer Communications, Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies, Twenty years into the communications odyssey, Anchorage, Alaska, USA, April 22-26, 2001*, pages 1520–1529, 2001.
- [66] 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.
- [67] A. Fog. Software optimization resources: Instruction tables. <https://www.agner.org/optimize/>. [Online; accessed 12-Feb-2020].

- [68] F. Fusco, M. P. Stoecklin, and M. Vlachos. Net-fli: On-the-fly compression, archiving and indexing of streaming network traffic. *PVLDB*, 3(2):1382–1393, 2010.
- [69] S. Gog, T. Beller, A. Moffat, and M. Petri. From theory to practice: Plug and play with succinct data structures. In *13th International Symposium on Experimental Algorithms, (SEA 2014)*, pages 326–337, 2014.
- [70] R. González, S. Grabowski, V. Mäkinen, and G. Navarro. Practical implementation of rank and select queries. In *Poster Proc. Volume of 4th Workshop on Efficient and Experimental Algorithms (WEA)*, pages 27–38, 2005.
- [71] T. M. Graf and D. Lemire. Xor filters: Faster and smaller than bloom and cuckoo filters. *CoRR*, abs/1912.08258, 2019.
- [72] L. L. Gremillion. Designing a bloom filter for differential file access. *Commun. ACM*, 25(9):600–604, 1982.
- [73] B. Grönvall. Scalable multicast forwarding. *Computer Communication Review*, 32(1):68, 2002.
- [74] T. Gubner and P. Boncz. Exploring Query Compilation Strategies for JIT, Vectorization and SIMD. In *Eighth International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures, ADMS 2017, Munich, Germany, September 1, 2017*, 2017.
- [75] T. Gubner, D. G. Tomé, H. Lang, and P. A. Boncz. Fluid co-processing: GPU bloom-filters for CPU joins. In *Proceedings of the 15th International Workshop on Data Management on New Hardware, DaMoN 2019, Amsterdam, The Netherlands, 1 July 2019*, pages 9:1–9:10, 2019.
- [76] G. Guzun, G. Canahuate, D. Chiu, and J. Sawin. A tunable compression framework for bitmap indices. In *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*, pages 484–495, 2014.
- [77] 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.
- [78] Y. Hu, S. Sundara, T. Chorma, and J. Srinivasan. Supporting rfid-based item tracking applications in oracle DBMS using a bitmap datatype. In *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005*, pages 1140–1151, 2005.

- [79] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, Sep. 1952.
- [80] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. 2018.
- [81] G. Jacobson. Space-efficient static trees and graphs. In *30th Annual Symposium on Foundations of Computer Science, Research Triangle Park, North Carolina, USA, 30 October - 1 November 1989*, pages 549–554, 1989.
- [82] T. Johnson. Performance measurements of compressed bitmap indices. In *VLDB’99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 278–289, 1999.
- [83] P. Jokela, A. Zahemszky, C. E. Rothenberg, S. Arianfar, and P. Nikander. LIPSIN: line speed publish/subscribe inter-networking. In *Proceedings of the ACM SIGCOMM 2009 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, Barcelona, Spain, August 16-21, 2009*, pages 195–206, 2009.
- [84] H. S. W. Jr. *Hacker’s Delight, Second Edition*. Pearson Education, 2013.
- [85] A. Kemper and T. Neumann. Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots. In *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, pages 195–206, 2011.
- [86] T. Kersten, V. Leis, A. Kemper, T. Neumann, A. Pavlo, and P. A. Boncz. Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. *PVLDB*, 11(13):2209–2222, 2018.
- [87] A. Kim, L. Xu, T. Siddiqui, S. Huang, S. Madden, and A. G. Parameswaran. Speedy browsing and sampling with needletail. *CoRR*, abs/1611.04705, 2016.
- [88] C. Kim, E. Sedlar, J. Chhugani, T. Kaldewey, A. D. Nguyen, A. D. Blas, V. W. Lee, N. Satish, and P. Dubey. Sort vs. hash revisited: Fast join implementation on modern multi-core cpus. *PVLDB*, 2(2):1378–1389, 2009.
- [89] S. Kim, J. Lee, S. R. Satti, and B. Moon. SBH: super byte-aligned hybrid bitmap compression. *Inf. Syst.*, 62:155–168, 2016.
- [90] A. Kipf, D. Chromejko, A. Hall, P. Boncz, and D. G. Andersen. Cuckoo index: A lightweight secondary index structure. 13(13):3559–3572, Sept. 2020.

- [91] A. Kipf, H. Lang, V. Pandey, R. A. Persa, P. Boncz, T. Neumann, and A. Kemper. Approximate geospatial joins with precision guarantees. In *34rd IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*, 2018.
- [92] A. Kirsch and M. Mitzenmacher. Less hashing, same performance: Building a better bloom filter. *Random Struct. Algorithms*, 33(2):187–218, 2008.
- [93] 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.
- [94] N. Koudas. Space efficient bitmap indexing. In *Proceedings of the 2000 ACM CIKM International Conference on Information and Knowledge Management, McLean, VA, USA, November 6-11, 2000*, pages 194–201, 2000.
- [95] H. Lang, V. Leis, M. Albutiu, T. Neumann, and A. Kemper. Massively parallel NUMA-aware hash joins. In *Proceedings of the 1st International Workshop on In Memory Data Management and Analytics, IMDM 2013, Riva Del Garda, Italy, August 26, 2013*, pages 1–12, 2013.
- [96] 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.
- [97] J. Larisch, D. R. Choffnes, D. Levin, B. M. Maggs, A. Mislove, and C. Wilson. Crlite: A scalable system for pushing all TLS revocations to all browsers. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 539–556, 2017.
- [98] C. Lattner and V. S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*, pages 75–88, 2004.
- [99] C. C. Lee, D. T. Lee, and C. K. Wong. Generating binary trees of bounded height. *Acta Inf.*, 23(5):529–544, 1986.

- [100] V. Leis, P. A. Boncz, A. Kemper, and T. Neumann. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 743–754, 2014.
- [101] 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.
- [102] D. Lemire, G. S. Y. Kai, and O. Kaser. Consistently faster and smaller compressed bitmaps with roaring. *Softw., Pract. Exper.*, 46(11):1547–1569, 2016.
- [103] D. Lemire, O. Kaser, and K. Aouiche. Sorting improves word-aligned bitmap indexes. *Data Knowl. Eng.*, 69(1):3–28, 2010.
- [104] D. Lemire, O. Kaser, and E. Gutarra. Reordering rows for better compression: Beyond the lexicographic order. *ACM Trans. Database Syst.*, 37(3):20:1–20:29, 2012.
- [105] D. Lemire and C. Rupp. Upscaledb: Efficient integer-key compression in a key-value store using SIMD instructions. *Inf. Syst.*, 66:13–23, 2017.
- [106] C. Li, Z. Chen, W. Zheng, Y. Wu, and J. Cao. BAH: A bitmap index compression algorithm for fast data retrieval. In *41st IEEE Conference on Local Computer Networks, LCN 2016, Dubai, United Arab Emirates, November 7-10, 2016*, pages 697–705, 2016.
- [107] D. Li, H. Cui, Y. Hu, Y. Xia, and X. Wang. Scalable data center multicast using multi-class bloom filter. In *Proceedings of the 19th annual IEEE International Conference on Network Protocols, ICNP 2011, Vancouver, BC, Canada, October 17-20, 2011*, pages 266–275, 2011.
- [108] D. Li, Y. Li, J. Wu, S. Su, and J. Yu. ESM: efficient and scalable data center multicast routing. *IEEE/ACM Trans. Netw.*, 20(3):944–955, 2012.
- [109] Y. Li, C. Chasseur, and J. M. Patel. A padded encoding scheme to accelerate scans by leveraging skew. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 1509–1524, 2015.
- [110] 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.
- [111] Y. Liu, W. Tome, L. Zhang, D. R. Choffnes, D. Levin, B. M. Maggs, A. Mislove, A. Schulman, and C. Wilson. An end-to-end measurement of

- certificate revocation in the web's PKI. In *Proceedings of the 2015 ACM Internet Measurement Conference, IMC 2015, Tokyo, Japan, October 28-30, 2015*, pages 183–196, 2015.
- [112] L. F. Mackert and G. M. Lohman. R* optimizer validation and performance evaluation for distributed queries. In *VLDB'86 Twelfth International Conference on Very Large Data Bases, August 25-28, 1986, Kyoto, Japan, Proceedings*, pages 149–159, 1986.
- [113] R. MacNicol and B. French. Sybase IQ multiplex - designed for analytics. In *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, VLDB 2004, Toronto, Canada, August 31 - September 3 2004*, pages 1227–1230, 2004.
- [114] S. Manegold, P. A. Boncz, and M. L. Kersten. Optimizing main-memory join on modern hardware. *IEEE Trans. Knowl. Data Eng.*, 14(4):709–730, 2002.
- [115] P. Menon, A. Pavlo, and T. C. Mowry. Relaxed operator fusion for in-memory databases: Making compilation, vectorization, and prefetching work together at last. *PVLDB*, 11(1):1–13, 2017.
- [116] M. Mitzenmacher. Compressed bloom filters. *IEEE/ACM Trans. Netw.*, 10(5):604–612, 2002.
- [117] M. Mitzenmacher and E. Upfal. *Probability and computing - randomized algorithms and probabilistic analysis*. Cambridge University Press, 2005.
- [118] 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.
- [119] T. Mühlbauer, W. Rödiger, R. Seilbeck, A. Reiser, A. Kemper, and T. Neumann. Instant loading for main memory databases. *PVLDB*, 6(14):1702–1713, 2013.
- [120] J. K. Mullin. Optimal semijoins for distributed database systems. *IEEE Trans. Software Eng.*, 16(5):558–560, 1990.
- [121] J. K. Mullin. Estimating the size of a relational join. *Inf. Syst.*, 18(3):189–196, 1993.
- [122] J. Munro and V. Raman. Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing*, 31(3):762–776, 2001.
- [123] J. I. Munro and V. Raman. Succinct representation of balanced parentheses, static trees and planar graphs. In *38th Annual Symposium on Foundations of Computer Science, FOCS '97, Miami Beach, Florida, USA, October 19-22, 1997*, pages 118–126, 1997.

- [124] J. I. Munro, V. Raman, and A. J. Storm. Representing dynamic binary trees succinctly. In *Proceedings of the Twelfth Annual Symposium on Discrete Algorithms, January 7-9, 2001, Washington, DC, USA.*, pages 529–536, 2001.
- [125] P. Nagarkar, K. S. Candan, and A. Bhat. Compressed spatial hierarchical bitmap (cshb) indexes for efficiently processing spatial range query workloads. *PVLDB*, 8(12):1382–1393, 2015.
- [126] G. Navarro and E. Provedel. Fast, small, simple rank/select on bitmaps. In *Experimental Algorithms - 11th International Symposium, SEA 2012, Bordeaux, France, June 7-9, 2012. Proceedings*, pages 295–306, 2012.
- [127] T. Neumann. Cuckoo filters with arbitrarily sized tables. <http://databasearchitects.blogspot.com/2019/07/cuckoo-filters-with-arbitrarily-sized.html>. [Online; accessed 27-Feb-2020].
- [128] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4(9):539–550, 2011.
- [129] M. Nowakiewicz, E. Boutin, E. Hanson, R. Walzer, and A. Katipally. Bipie: Fast selection and aggregation on encoded data using operator specialization. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD '18*, pages 1447–1459, New York, NY, USA, 2018. ACM.
- [130] P. O’Neil and G. Graefe. Multi-table joins through bitmapped join indices. *SIGMOD Rec.*, 24(3):8–11, Sept. 1995.
- [131] P. E. O’Neil. Model 204 architecture and performance. In *High Performance Transaction Systems, 2nd International Workshop, Asilomar Conference Center, Pacific Grove, California, USA, September 28-30, 1987, Proceedings*, pages 40–59, 1987.
- [132] P. E. O’Neil, E. Cheng, D. Gawlick, and E. J. O’Neil. The log-structured merge-tree (lsm-tree). *Acta Inf.*, 33(4):351–385, 1996.
- [133] P. E. O’Neil and D. Quass. Improved query performance with variant indexes. In *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA.*, pages 38–49, 1997.
- [134] Oracle Corporation. Bitmap Index vs. B-tree Index: Which and When? <https://www.oracle.com/technetwork/articles/sharma-indexes-093638.html>, 2005. [Online; accessed 14-Jun-2019].
- [135] P. E. O’Neil, E. J. O’Neil, and X. Chen. The star schema benchmark (ssb). *Pat*, 200(0):50, 2007.

- [136] R. Pagh and F. F. Rodler. Cuckoo hashing. *J. Algorithms*, 51(2):122–144, 2004.
- [137] 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.
- [138] T. B. Pedersen and C. S. Jensen. Research issues in clinical data warehousing. In *10th International Conference on Scientific and Statistical Database Management, Proceedings, Capri, Italy, July 1-3, 1998*, pages 43–52, 1998.
- [139] A. Pinar, T. Tao, and H. Ferhatosmanoglu. Compressing bitmap indices by data reorganization. In *Proceedings of the 21st International Conference on Data Engineering, ICDE 2005, 5-8 April 2005, Tokyo, Japan*, pages 310–321, 2005.
- [140] O. Polychroniou, A. Raghavan, and K. A. Ross. Rethinking SIMD vectorization for in-memory databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 1493–1508, 2015.
- [141] O. Polychroniou and K. A. Ross. High throughput heavy hitter aggregation for modern SIMD processors. In *Proceedings of the Ninth International Workshop on Data Management on New Hardware, DaMoN 1013, New York, NY, USA, June 24, 2013*, page 6, 2013.
- [142] 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.
- [143] 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.
- [144] 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.
- [145] R. Raman, V. Raman, and S. R. Satti. Succinct indexable dictionaries with applications to encoding k -ary trees, prefix sums and multisets. *ACM Trans. Algorithms*, 3(4):43, 2007.
- [146] B. Ren, G. Agrawal, J. R. Larus, T. Mytkowicz, T. Poutanen, and W. Schulte. SIMD parallelization of applications that traverse irregular data structures. In *Proceedings of the 2013 IEEE/ACM International*

Symposium on Code Generation and Optimization, CGO 2013, Shenzhen, China, February 23-27, 2013, pages 20:1–20:10, 2013.

- [147] D. Rinfret, P. E. O’Neil, and E. J. O’Neil. Bit-sliced index arithmetic. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data, Santa Barbara, CA, USA, May 21-24, 2001*, pages 47–57, 2001.
- [148] K. A. Ross. Efficient hash probes on modern processors. In *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15-20, 2007*, pages 1297–1301, 2007.
- [149] A. Rousskov and D. Wessels. Cache digests. *Computer Networks*, 30(22-23):2155–2168, 1998.
- [150] P. Roy, J. Teubner, and R. Gemulla. Low-latency handshake join. *Proc. VLDB Endow.*, 7(9):709–720, 2014.
- [151] R. Sears and R. Ramakrishnan. bLSM: a general purpose log structured merge tree. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 217–228, 2012.
- [152] A. Shatdal, C. Kant, and J. F. Naughton. Cache conscious algorithms for relational query processing. In *VLDB’94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, pages 510–521, 1994.
- [153] 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.
- [154] L. Sidirourgos and H. Mühleisen. Scaling column imprints using advanced vectorization. In *Proceedings of the 13th International Workshop on Data Management on New Hardware, DaMoN 2017, Chicago, IL, USA, May 15, 2017*, pages 4:1–4:8, 2017.
- [155] R. R. Sinha, S. Mitra, and M. Winslett. Bitmap indexes for large scientific data sets: a case study. In *20th International Parallel and Distributed Processing Symposium (IPDPS 2006), Proceedings, 25-29 April 2006, Rhodes Island, Greece*, 2006.
- [156] R. R. Sinha and M. Winslett. Multi-resolution bitmap indexes for scientific data. *ACM Trans. Database Syst.*, 32(3):16, 2007.

- [157] E. A. Sitaridi, O. Polychroniou, and K. A. Ross. Simd-accelerated regular expression matching. In *Proceedings of the 12th International Workshop on Data Management on New Hardware, DaMoN 2016, San Francisco, CA, USA, June 27, 2016*, pages 8:1–8:7, 2016.
- [158] H. Song, S. Dharmapurikar, J. S. Turner, and J. W. Lockwood. Fast hash table lookup using extended bloom filter: an aid to network processing. In *Proceedings of the ACM SIGCOMM 2005 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, Philadelphia, Pennsylvania, USA, August 22-26, 2005*, pages 181–192, 2005.
- [159] M. Stabno and R. Wrembel. RLH: bitmap compression technique based on run-length and huffman encoding. In *DOLAP 2007, ACM 10th International Workshop on Data Warehousing and OLAP, Lisbon, Portugal, November 9, 2007, Proceedings*, pages 41–48, 2007.
- [160] K. Stockinger. Design and implementation of bitmap indices for scientific data. In *International Database Engineering & Applications Symposium, IDEAS '01, July 16-18, 2001, Grenoble, France, Proceedings*, pages 47–57, 2001.
- [161] K. Stockinger. Bitmap indices for speeding up high-dimensional data analysis. In *Database and Expert Systems Applications, 13th International Conference, DEXA 2002, Aix-en-Provence, France, September 2-6, 2002, Proceedings*, pages 881–890, 2002.
- [162] K. Stockinger and K. Wu. Bitmap indices for data warehouses. In *Data Warehouses and OLAP: Concepts, Architectures and Solutions*, pages 157–178. IGI Global, 2007.
- [163] K. Stockinger, K. Wu, and A. Shoshani. Evaluation strategies for bitmap indices with binning. In *Database and Expert Systems Applications, 15th International Conference, DEXA 2004 Zaragoza, Spain, August 30-September 3, 2004, Proceedings*, pages 120–129, 2004.
- [164] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. J. O’Neil, P. E. O’Neil, A. Rasin, N. Tran, and S. B. Zdonik. C-store: A column-oriented DBMS. In *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005*, pages 553–564, 2005.
- [165] S. Tarkoma, C. E. Rothenberg, and E. Lagerspetz. Theory and practice of bloom filters for distributed systems. *IEEE Communications Surveys and Tutorials*, 14(1):131–155, 2012.

- [166] J. Teubner and R. Müller. How soccer players would do stream joins. In T. K. Sellis, R. J. Miller, A. Kementsietsidis, and Y. Velegrakis, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*, pages 625–636. ACM, 2011.
- [167] P. Valduriez and G. Gardarin. Join and semijoin algorithms for a multi-processor database machine. *ACM Trans. Database Syst.*, 9(1):133–161, 1984.
- [168] S. J. van Schaik and O. de Moor. A memory efficient reachability data structure through bit vector compression. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*, pages 913–924, 2011.
- [169] S. Vigna. Broadword implementation of rank/select queries. In *Experimental Algorithms, 7th International Workshop, WEA 2008, Provincetown, MA, USA, May 30-June 1, 2008, Proceedings*, pages 154–168, 2008.
- [170] B. Wang, H. Litz, and D. R. Cheriton. HICAMP bitmap: space-efficient updatable bitmap index for in-memory databases. In *Tenth International Workshop on Data Management on New Hardware, DaMoN 2014, Snowbird, UT, USA, June 23, 2014*, pages 7:1–7:7, 2014.
- [171] J. Wang, C. Lin, Y. Papakonstantinou, and S. Swanson. An experimental study of bitmap compression vs. inverted list compression. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 993–1008, 2017.
- [172] M. Wang, M. Zhou, S. Shi, and C. Qian. Vacuum filters: More space-efficient and faster replacement for bloom and cuckoo filters. *PVLDB*, 13(2):197–210, 2019.
- [173] S. Wang, D. Maier, and B. C. Ooi. Lightweight indexing of observational data in log-structured storage. *PVLDB*, 7(7):529–540, 2014.
- [174] J. W. J. Williams. Algorithm 232: Heapsort. *Communications of the ACM*, 7(6):347–348, 1964.
- [175] K. Wu. Notes on design and implementation of compressed bit vectors. 2001.
- [176] K. Wu, S. Ahern, E. W. Bethel, J. Chen, H. Childs, E. Cormier-Michel, C. Geddes, J. Gu, H. Hagen, B. Hamann, et al. FastBit: interactively searching massive data. In *Journal of Physics: Conference Series*, volume 180, page 012053. IOP Publishing, 2009.

- [177] K. Wu, E. Otoo, and A. Shoshani. Compressed bitmap indices for efficient query processing. *Rep. LBNL-47807 Lawrence Berkeley National Laboratory Berkeley, CA*, 2001.
- [178] K. Wu, E. J. Otoo, and A. Shoshani. On the performance of bitmap indices for high cardinality attributes. In *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, VLDB 2004, Toronto, Canada, August 31 - September 3 2004*, pages 24–35, 2004.
- [179] K. Wu, E. J. Otoo, and A. Shoshani. Optimizing bitmap indices with efficient compression. *ACM Trans. Database Syst.*, 31(1):1–38, 2006.
- [180] K. Wu, A. Shoshani, and K. Stockinger. Analyses of multi-level and multi-component compressed bitmap indexes. *ACM Trans. Database Syst.*, 35(1):2:1–2:52, 2010.
- [181] K. Wu, K. Stockinger, and A. Shoshani. Performances of multi-level and multi-component compressed bitmap indices. 2007.
- [182] K. Wu and P. S. Yu. Range-based bitmap indexing for high cardinality attributes with skew. In *COMPSAC '98 - 22nd International Computer Software and Applications Conference, August 19-21, 1998, Vienna, Austria*, pages 61–67, 1998.
- [183] M. Wu and A. P. Buchmann. Encoded bitmap indexing for data warehouses. In *Proceedings of the Fourteenth International Conference on Data Engineering, Orlando, Florida, USA, February 23-27, 1998*, pages 220–230, 1998.
- [184] J. Yu and M. Sarwat. Two birds, one stone: A fast, yet lightweight, indexing scheme for modern database systems. *PVLDB*, 10(4):385–396, 2016.
- [185] M. Yu, A. Fabrikant, and J. Rexford. BUFFALO: bloom filter forwarding architecture for large organizations. In *Proceedings of the 2009 ACM Conference on Emerging Networking Experiments and Technology, CoNEXT 2009, Rome, Italy, December 1-4, 2009*, pages 313–324, 2009.
- [186] Y. Yu, C. Qian, and X. Li. Distributed and collaborative traffic monitoring in software defined networks. In *Proceedings of the third workshop on Hot topics in software defined networking, HotSDN '14, Chicago, Illinois, USA, August 22, 2014*, pages 85–90, 2014.
- [187] E. T. Zacharatou, F. Tauheed, T. Heinis, and A. Ailamaki. RUBIK: efficient threshold queries on massive time series. In *Proceedings of the 27th International Conference on Scientific and Statistical Database Management, SSDBM '15, La Jolla, CA, USA, June 29 - July 1, 2015*, pages 18:1–18:12, 2015.

- [188] H. Zhang, H. Lim, V. Leis, D. G. Andersen, M. Kaminsky, K. Keeton, and A. Pavlo. SuRF: Practical range query filtering with Fast Succinct Tries. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 323–336, 2018.
- [189] W. X. Zhao, X. Zhang, D. Lemire, D. Shan, J. Nie, H. Yan, and J. Wen. A general simd-based approach to accelerating compression algorithms. *ACM Trans. Inf. Syst.*, 33(3):15:1–15:28, 2015.
- [190] D. Zhou, D. G. Andersen, and M. Kaminsky. Space-efficient, high-performance rank and select structures on uncompressed bit sequences. In *Experimental Algorithms, 12th International Symposium, SEA 2013, Rome, Italy, June 5-7, 2013. Proceedings*, pages 151–163, 2013.
- [191] J. Zhou and K. A. Ross. Implementing database operations using SIMD instructions. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, June 3-6, 2002*, pages 145–156, 2002.
- [192] 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.

Appendix

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**

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-nd/4.0/>. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.

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

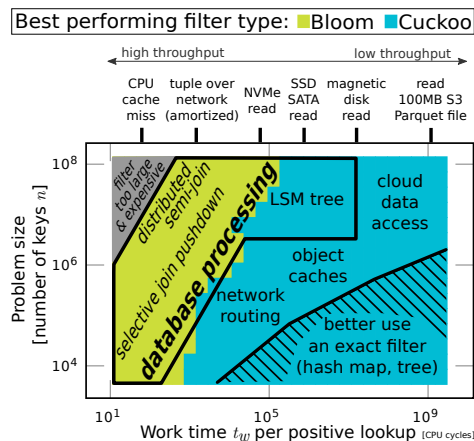


Figure 1: Performance-optimal filter types for different problem sizes n and potential work savings t_w . Example reference points for t_w values are shown above the plot and applications inside the plot.

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.

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

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 $\sigma=1.0$ (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 $1-\sigma$ 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 $(1-\sigma)$.

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 bit-wise 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 structures¹.

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.

¹Source 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 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:

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

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: $\rho(F^{opt}) < (1 - \sigma) * t_w$.

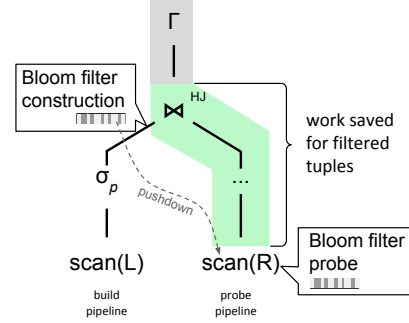


Figure 2: Bloom filters with selective joins. Tuples without a join partner are filtered before entering the pipeline.

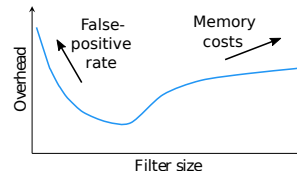


Figure 3: Overhead ρ as a function in m for fixed configuration F , n and t_w .

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

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

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:

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

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

```
// 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.

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.

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

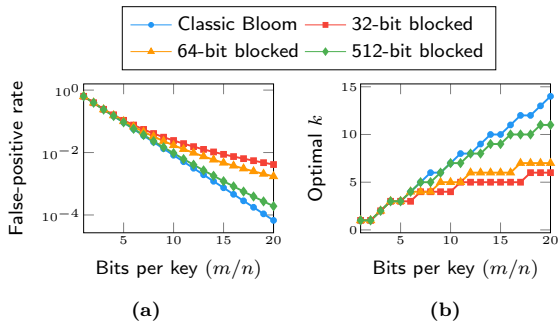


Figure 4: Impact of blocking on the false-positive-rate and the hashing efforts.

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

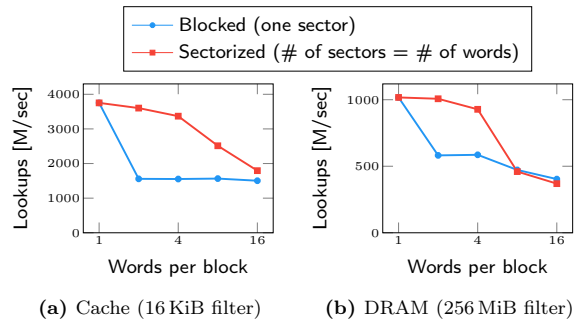


Figure 5: Performance impact of sectorization for varying block sizes.

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

²The 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:

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

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

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:

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

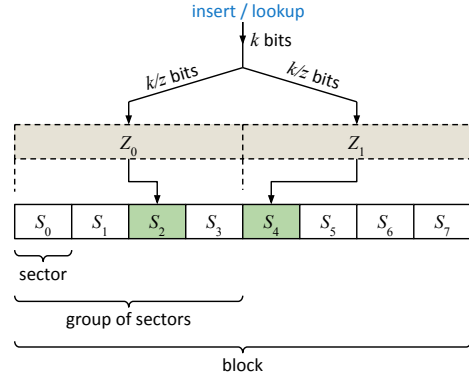


Figure 6: Block partitioning scheme of cache-sectorized Bloom filters: hashing sets bits concentrated in z words spread over a cache line.

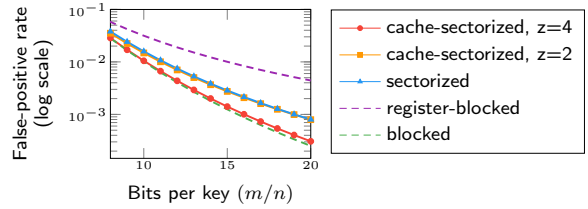


Figure 7: Comparison of the false-positive rate of sectorized and cache-sectorized Bloom filters, with $k = 8$.

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

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

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 ($\sim 50\%$) 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

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

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

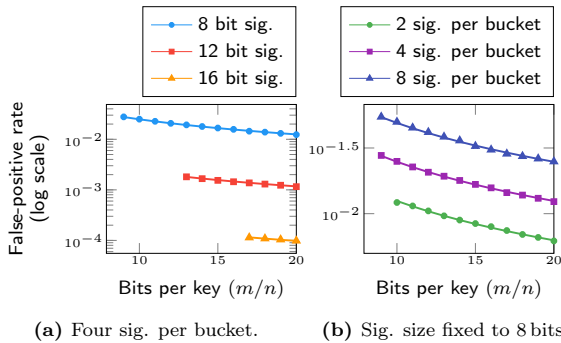


Figure 8: The false-positive rate of Cuckoo filters for different signature lengths and bucket sizes.

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 than 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 ANDs (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 $\text{hash}(\text{key}) \bmod m$, which involves an integer division, is several times slower than using $\text{hash}(\text{key}) \& \text{mask}$ (with $\text{mask} := (1 \ll \log_2(m)) - 1$). 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 $6 \times$ 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 $i = \text{hash}(\text{key}) \bmod C$ is thereby replaced by

$$h = \text{hash}(\text{key})$$

$$i = h - (\text{mulhi_u32}(h, \text{magicNo}) \gg \text{shiftAmount}) * h, \quad (9)$$

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 block³ of the Bloom filter and a bucket in the Cuckoo filter, respectively. The *actual* filter size is therefore

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

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:

$$i_1 = \text{hash}(x) \text{ magicMod } C$$

$$i_2 = -(i_1 + \text{hash}(x\text{'s signature})) \text{ magicMod } C \quad (11)$$

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.

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

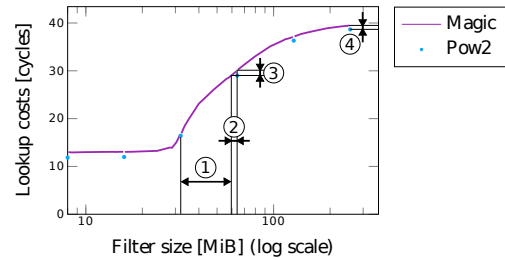


Figure 9: Lookup performance of a cache-sectorized Bloom filter for varying filter sizes.

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 valid⁴ 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

⁴Please 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_w s 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_w s, 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 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 f s 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_w s.

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 k s, 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 .

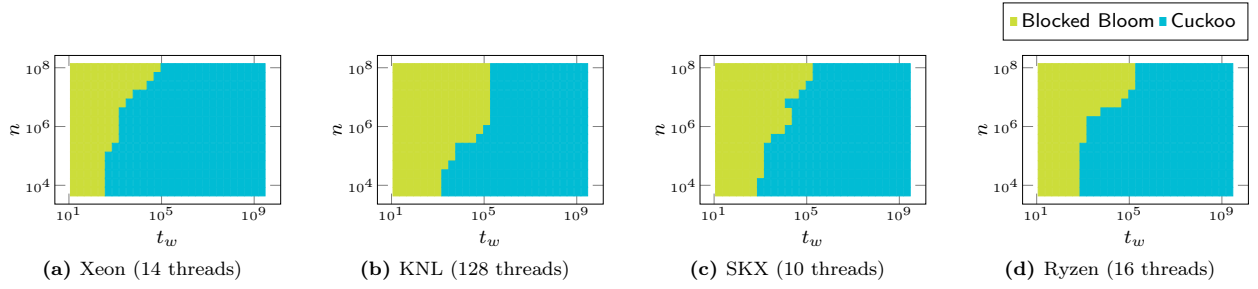


Figure 10: Skyline of performance-optimal filters for varying n and t_w .

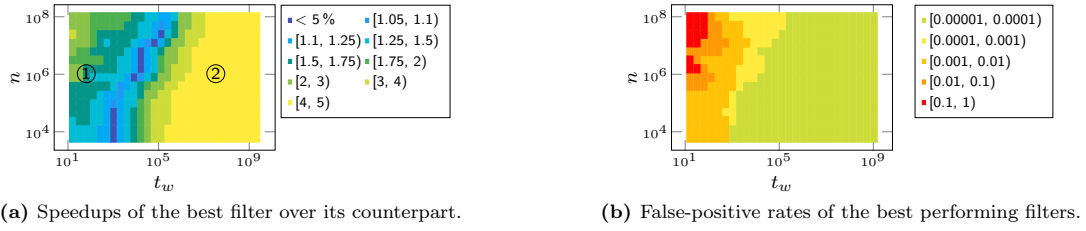


Figure 11: Performance comparison of Bloom and Cuckoo filters (a) and the corresponding false-positive rates (b).

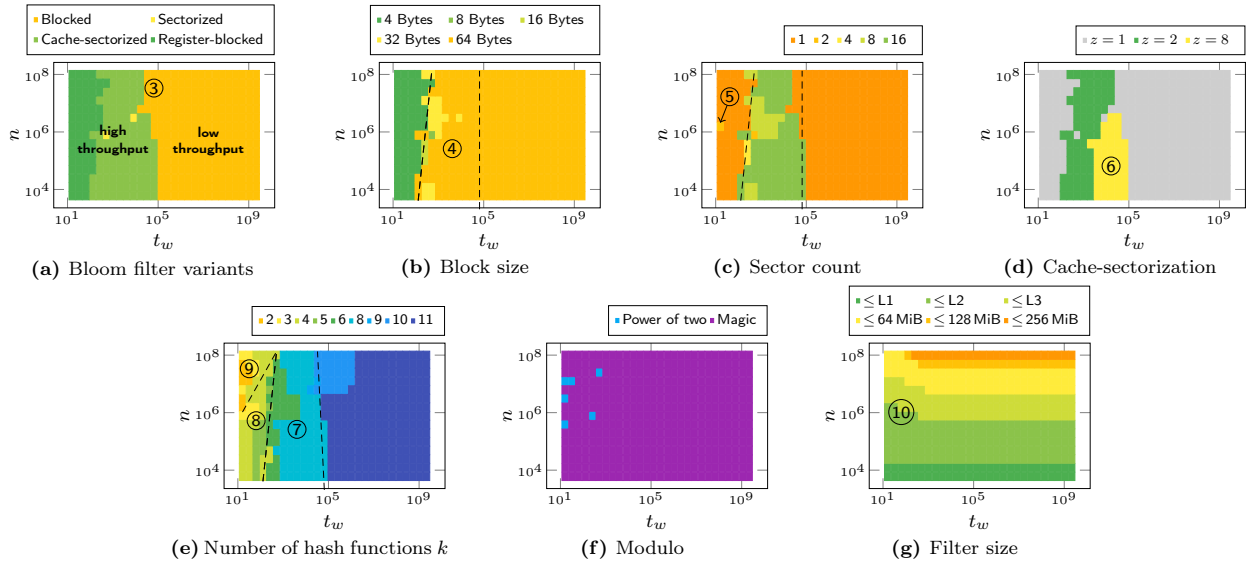


Figure 12: Skyline of configurations of the best performing blocked Bloom filters (on SKX).

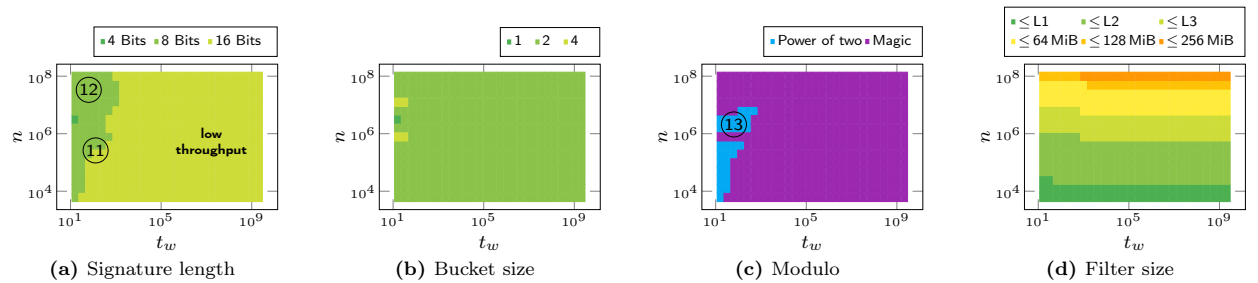


Figure 13: Skyline of configurations of the best performing Cuckoo filters (on SKX).

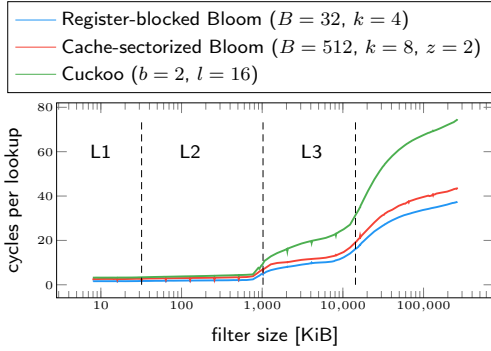


Figure 14: Lookup time for varying filter sizes (on SKX).

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

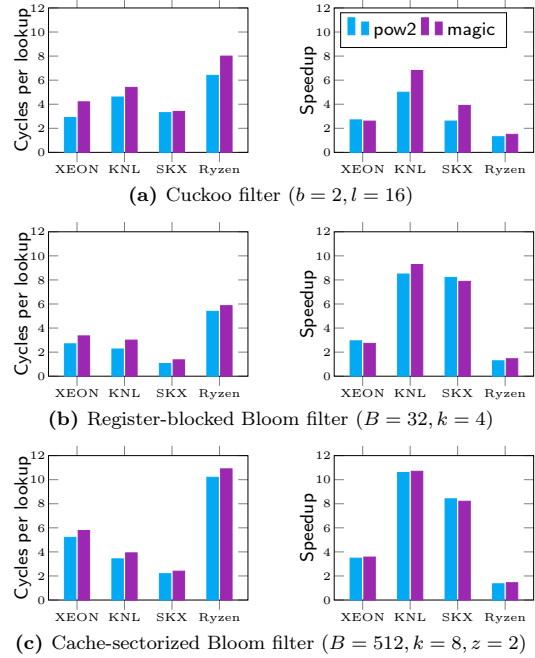


Figure 15: Performance of our SIMD-optimized filter implementations.

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 $t_{m,s}$). 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-512BW (Byte Word) 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 numbers⁵ on similar hardware (Ryzen Threadripper 1950X), showing that our implementation provides the same query performance with large filters ($\approx 200\text{MB}$); 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.

⁵At 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. Kuzmaul, 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. Bobrovitsky, 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. Sidiourgos 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.



Make the most out of your SIMD investments: counter control flow divergence in compiled query pipelines

Harald Lang¹ · Linnea Passing¹ · Andreas Kipf¹ · Peter Boncz² · Thomas Neumann¹ · Alfons Kemper¹

Received: 14 December 2018 / Revised: 23 April 2019 / Accepted: 13 June 2019 / Published online: 16 July 2019
© The Author(s) 2019

Abstract

Increasing single instruction multiple data (SIMD) capabilities in modern hardware allows for the compilation of data-parallel query pipelines. This means GPU-alike challenges arise: *control flow divergence* causes the underutilization of vector-processing units. In this paper, we present efficient algorithms for the AVX-512 architecture to address this issue. These algorithms allow for the fine-grained assignment of new tuples to idle SIMD lanes. Furthermore, we present strategies for their integration with compiled query pipelines so that tuples are never evicted from registers. We evaluate our approach with three query types: (i) a table scan query based on TPC-H Query 1, that performs up to 34% faster when addressing underutilization, (ii) a hashjoin query, where we observe up to 25% higher performance, and (iii) an approximate geospatial join query, which shows performance improvements of up to 30%.

Keywords Control flow divergence · Database systems · Query execution · Query compilation · SIMD · Vectorization · AVX-512

1 Introduction

Integrating SIMD processing with database systems has been studied for more than a decade [28]. Several operations, such as selection [12,23], join [2,3,10,26], partitioning [20], sorting [5], CSV parsing [17], regular expression matching [25],

and (de-)compression [15,23,27] have been accelerated using the SIMD capabilities of the x86 architectures. In more recent iterations of hardware evolution, SIMD instruction sets have become even more popular in the field of database systems. Wider registers, higher degrees of data-parallelism, and comprehensive support for integer data has increased the interest in SIMD and led to the development of many novel algorithms.

SIMD is mostly used in interpreting database systems [9] that use the *column-at-a-time* or *vector-at-a-time* execution model [4]. Compiling database systems [9] like HyPer [8] barely use it due to their data-centric *tuple-at-a-time* execution model [18]. In such systems, therefore, SIMD is primarily used in scan operators [12] and in string processing [17].

With the increasing vector-processing capabilities for database workloads in modern hardware, especially with the advent of the AVX-512 instruction set, query compilers can now vectorize entire query execution pipelines and benefit from the high degree of data-parallelism [6]. With AVX-512, the width of vector registers increased to 512 bit, allowing for the processing of an entire cache line in a single instruction. Depending on the bit-width of the attribute values, data elements from up to 64 tuples can be packed into a single register.

An excerpt of this invited paper in the special issue “Best of DaMoN” appeared in DaMoN 2018.

Harald Lang
harald.lang@in.tum.de

Linnea Passing
linnea.passing@tum.de

Andreas Kipf
andreas.kipf@in.tum.de

Peter Boncz
boncz@cwi.nl

Thomas Neumann
thomas.neumann@in.tum.de

Alfons Kemper
alfons.kemper@in.tum.de

¹ Technical University of Munich, Boltzmannstr. 3, 85748 Garching, Germany

² Centrum Wiskunde & Informatica, Science Park 123, 1098 XG Amsterdam, The Netherlands

Vectorizing entire query pipelines raises new challenges. One such challenge is keeping all SIMD lanes busy during query evaluation, as not all in-flight tuples follow the same control flow. For instance, some might be disqualified during predicate evaluation, while others may not find a join partner later on and get discarded. Whenever a tuple gets disqualified, the corresponding SIMD lane is affected. A scalar (non-vectorized) pipeline would take a branch and thereby return the control flow to a tuple producing operator to fetch the next tuple. In a vectorized pipeline, this is only possible iff **all in-flight tuples have been disqualified**. If this is not the case, the query of the subsequent operator still needs to be executed. Ignoring SIMD lanes containing disqualified tuples is the easiest way to deal with this situation, as it does not introduce branching logic and only requires a small amount of bookkeeping. A small bitmap is sufficient to keep track of disqualified elements. The bitmap is used at the pipeline sink, when the (intermediate) result is materialized, making sure that disqualified elements are not written to the query result set. The downside of this approach is, that within the pipeline, all instructions are performed on all SIMD lanes regardless of whether the SIMD lane contains an active or an inactive element. All operations that are performed on inactive elements can be considered overhead, as they do not contribute to the result. In other words, not all SIMD lanes perform useful work and if lanes contain disqualified elements, the vector-processing units (VPUs) can be considered **underutilized**. Therefore, efficient algorithms are required to counter the underutilization of vector-processing units. In [16], this issue was addressed by introducing (memory) materialization points immediately after each vectorized operator. However, with respect to the more strict definition of pipeline breakers given in [18], materialization points can be considered as pipeline breakers because tuples are evicted from registers to slower (cache) memory. In this work, we present alternative algorithms and strategies that do not break pipelines. Further, our approach can be applied at the intra-operator level as well as at operator boundaries.

The remainder of this paper is organized as follows. In Sect. 2, we briefly describe the relevant AVX-512 instructions that we use in our algorithms. The potential performance degradation caused by underutilization in holistically vectorized pipelines is discussed in Sect. 3. In Sect. 4, we introduce efficient algorithms to counter underutilization, and in Sect. 5, we present strategies for integrating these algorithms with compiled query pipelines. The experimental evaluation of the proposed algorithms using a table scan query, a hashjoin query, and an approximate geospatial join query is given in Sect. 6. The experimental results are summarized and discussed in Sect. 7, followed by our conclusions in Sect. 8.

2 Background

In this section, we briefly describe the key features of the AVX-512 instruction set that we use in our algorithms in Sect. 4. In particular, we cover the basics of vector predication as well as the *permute* and the *compress/expand* instructions. **Mask instructions:** Almost all AVX-512 instructions support *predication*. These instructions allow to perform a vector operation only on those vector components (or lanes) specified by a given bitmask, where the i th bit in the bitmask corresponds to the i th lane. For example, an *add* instruction in its simplest form requires two (vector) operands and a destination register that receives the result. In AVX-512, the instruction exists in two additional variants:

1. **Merge masking:** The instruction takes two additional arguments, a *mask* and a source register, for example, `dst = mask_add(src, mask, a, b)`. The addition is performed on the vector components in *a* and *b* specified by the *mask*. The remaining elements, where the mask bits are 0, are copied from *src* to *dst* at their corresponding positions.
2. **Zero masking:** The functionality is basically the same as that of merge masking, but instead of specifying an additional source vector, all elements in *dst* are set to zero if the corresponding bit in the mask is not set. Zero masking is, therefore, (logically) equivalent to merge masking with *src* set to zero: `maskz_add(mask, a, b) ≡ mask_add(0, mask, a, b)`. Thus, zero masking is a special case of merge masking.

Masked instructions can be used to prevent individual vector components from being altered, e.g., `x = mask_add(x, mask, a, b)`.

Typically, masks are created using comparison instructions and stored in special mask registers, which is a significant improvement over earlier SIMD instruction sets, in which these masks were stored in 256-bit vector registers.

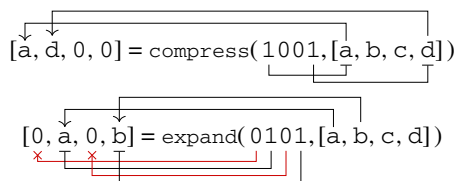
Permute: The *permute* instruction shuffles elements within a vector register according to a given index vector:

$$\underbrace{[d, a, \bar{d}, b]}_{\text{result vector}} = \text{permute}(\underbrace{[3, 0, 3, 1]}_{\text{index vector}}, \underbrace{[a, b, c, \bar{d}]}_{\text{input vector}}).$$

It is noteworthy, that the *permute* instruction has already been available in earlier instruction sets. But due to the doubled register size, twice as many elements can now be processed at once. Further, in our application, we achieve four times higher throughput compared to the earlier AVX2 instruction set. The reason is, that assigning new elements to idle SIMD lanes is basically a *merge* operation of the content of two vector registers. In combination with merge masking, this operation can be performed using a single instruction,

whereas with AVX2, two instructions need to be issued, (i) a `permute` to move the elements into their desired SIMD lanes and (ii) a `blend` to select the desired lanes from two source registers and merge them into a destination register.

Compress/Expand: Typically, before a `permute` instruction can be issued, an algorithm needs to determine the aforementioned index vector, which used to be a tedious task that often induced significant overheads, such as additional accesses into predefined lookup tables [7,12,16,22]. The key instructions introduced with AVX-512 to efficiently solve these types of problems, are called `compress` and `expand`. `Compress` stores the active elements (indicated by a bitmask) contiguously into a target register, and `expand` stores the contiguous elements of an input at certain positions (specified by a *write mask*) in a target register:



Both instructions come in two flavors: (i) read/write from/to memory and (ii) directly operate on registers.

Our algorithms in general require both, `permute` and `compress/expand` instructions. There is only one special case, where a `permute` suffices, which we describe in the later Sect. 4.

3 Vectorized pipelines

As mentioned in the introduction, the major difference between a scalar (i.e., non-vectorized) pipeline, as pioneered by HyPer [8], and a vectorized pipeline is that in the latter, multiple tuples are pushed through the pipeline at once. This impacts the *control flow* within the query pipeline. In a scalar pipeline, whenever the control flow reaches any operator, it is guaranteed that there is *exactly one* tuple to process (tuple-at-a-time). By contrast, in a vectorized pipeline, there are several tuples to process. However, because the control flow is not necessarily the same for all tuples, some SIMD lanes may become inactive when a conditional branch is taken. Such a branch is only taken if *at least one* element satisfies the branch condition. This implies that a vector of length n may contain up to $n - 1$ *inactive* elements, as depicted in Fig. 1. The figure shows a simplified control flow graph (CFG) for an example query pipeline that consists of a table scan, a selection, and a join operator. The directed edges represent the branching logic. For instance, the *no match* edges are taken if a tuple is disqualified in the selection or the join operator. The *index traversal* (self-)edge is taken when an index lookup is performed. For instance, a hash table or tree lookup might require one to follow multiple bucket pointers

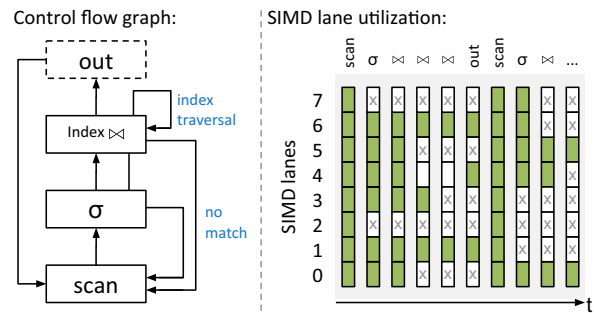


Fig. 1 During query processing, individual SIMD lanes may (temporarily) become inactive due to different control flows. The resulting underutilization of vector-processing units causes performance degradations. We propose efficient algorithms and strategies to fill these gaps

until a join partner for the current tuples is found. The right-hand side of Fig. 1 visualizes the SIMD lane utilization over time. Initially, in the scan operator, all SIMD lanes are active (green color). Inside the select or join operator, elements are disqualified (marked with X), but the *no match* branch is not taken, because some elements are still active. Lane 4 represents a different situation, where a SIMD lane becomes **temporarily inactive**. In that example, the element in lane 4 finds its join partner in the very first iteration of the index lookup. However, lanes 1 and 6 need three iterations until the index lookup terminates. During that time, lane 4 is idle and afterward, it becomes active again.

In general, all conditional branches within the query pipeline are potential sources of control flow *divergence* and, therefore, a source of the underutilization of VPUs, whereas, disqualified elements cause underutilization in all subsequent operators and lookups in index structures cause *intra-operator* underutilization. The latter is an inherent problem when traversing irregular pointer-based data structures in a SIMD fashion [24]. To avoid underutilization through divergence, we need to dynamically assign new tuples to idle SIMD lanes, possibly at multiple “points of divergence” within the query pipeline. We refer to this process as *pipeline refill*.

4 Refill algorithms

In this section, we present our refill algorithms for AVX-512, which we later integrate into compiled query pipelines (cf., Sect. 5). These algorithms essentially copy new elements to *desired positions* in a destination register. In this context, these desired positions are the lanes that contain inactive elements. The active lanes are identified by a small bitmask (or simply *mask*), where the i th bit corresponds to the i th SIMD lane. An SIMD lane is active if the corresponding bit is set, and vice versa. Thus, the bitwise complement of the given mask refers to the inactive lanes and, therefore, to the

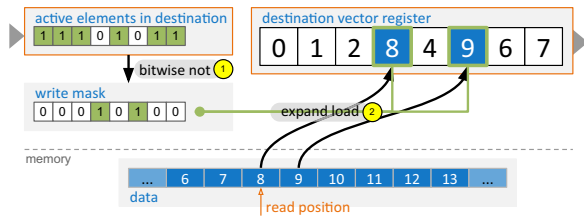


Fig. 2 Refilling empty SIMD lanes from memory using the AVX-512 expand load instruction

write positions of new elements. We distinguish between two cases as follows: (i) where new elements are copied from a source memory address and (ii) where elements are already in vector registers.

In the following, we frequently use various constant values, which we write in capital letters. For instance, **ZERO** and **ALL** refer to constant values where all bits are zero or one, respectively. The vector constant **SEQUENCE** contains an integer sequence starting at 0 and **LANE_CNT** refers to the number of SIMD lanes.

4.1 Memory to register

Refilling from memory typically occurs in the table scan operator, where contiguous elements are loaded from memory (assuming a columnar storage layout). AVX-512 offers the convenient `expand load` instruction that loads contiguous values from memory directly into the desired SIMD lanes (cf., Fig. 2). One mask instruction (`bitwise not`) is required to determine the `write mask` and one vector instruction (`expand load`) to execute the actual load. Overall, the simple case of refilling from memory is supported by AVX-512 directly out of the box.

The table scan operator typically produces an additional output vector containing the tuple identifiers (TIDs) of the newly loaded attribute values. The TIDs are derived from the current read position and are used, for example, to (lazily) load attribute values of a different column later on or to reconstruct the tuple order. Figure 3 illustrates, how the content of the TID vector register is updated, using the read position and write mask from Fig. 2.

4.2 Register to register

Moving data between vector registers is more involved. In the most general case, we have a source and a destination register that contain both active and inactive elements at *random* positions. The goal is to move as many elements as possible from the source to the destination. This can be achieved using a single masked `permute` instruction. But before the permutation instruction can be issued, the `permutation indices` need to be computed, based on the positions of active elements in

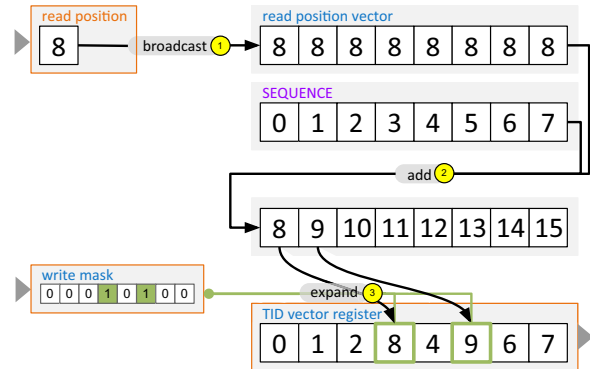


Fig. 3 TIDs are derived from the current read position and assigned to a TID vector register

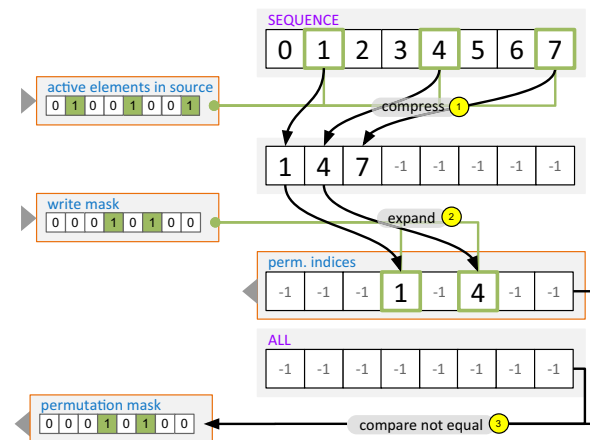


Fig. 4 Computation of the permutation indices and the permutation mask based on positions of the active elements in the source register and the inactive elements in the destination register

the source and the destination vector registers. This is illustrated in Fig. 4, where, as in the previous examples, the `write mask` refers to the inactive lanes in the destination register. In total, three vector instructions are required to compute the permutation indices and an additional permutation mask. The latter is required in case the number of active elements in the source is smaller than the number of empty lanes in the destination vector. In that case, the destination register still contains some inactive lanes, and the corresponding bitmask must be updated accordingly.

Once the permutation indices are computed, elements can be moved between registers accordingly. Notably, the algorithm can be adapted to move elements directly instead of computing the permutation indices first. However, if elements need to be moved between more than one source/destination vector pair, the additional cost of computing the permutation amortizes immediately with the second pair. In practice, the permutation is typically applied multiple times, for example, when multiple attributes are pushed through the pipeline or to keep track of the TIDs.

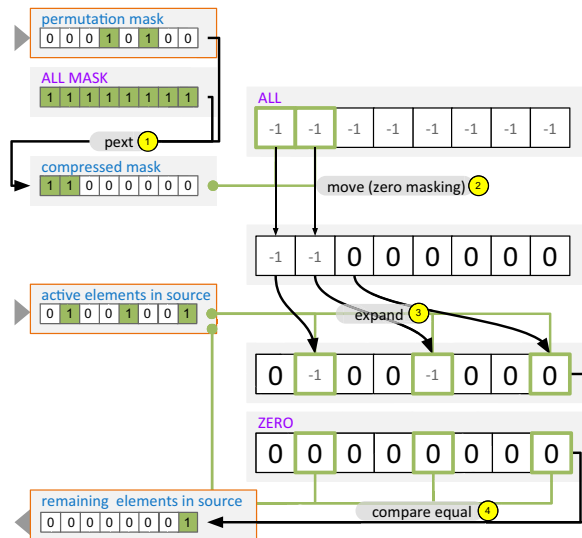


Fig. 5 If not all elements could be moved from the source to the destination register, the source mask needs to be updated accordingly

In the general case, there are no guarantees about the number of (active) elements nor their positions within the vector register. For example, the elements in the source may not be entirely consumed or the destination vector may still contain inactive elements. Thus, it is necessary to update source and destination masks accordingly. Updating the destination mask is straightforward by using a bitwise or with the previously computed permutation mask. Updating the source mask is less obvious as illustrated in Fig. 5. As the figure shows, updating the source mask is as expensive as preparing the permutation. However, if it is guaranteed that all source elements fit into the destination vector, this phase of the algorithm can be skipped altogether. Listing 1 shows the full algorithm formulated in C++.

In summary, a typical refill looks as follows:

```
[...]
//Prepare the refill.
fill_rr r(src_mask, dst_mask);
//Copy elements from src to dst.
r.apply(src_tid, dst_tid);
r.apply(src_attr_a, dst_attr_a);
r.apply(src_attr_b, dst_attr_b);
r.apply(..., ...);
//Update the destination mask,
r.update_dst_mask(dst_mask);
//and optionally the source mask.
r.update_src_mask(src_mask);
[...]
```

4.3 Variants

Depending on the position of the elements, cheaper algorithms can be used. Especially when the vectors are in a compressed state, meaning that the active elements are stored

contiguously, it is considerably cheaper to prepare the permutation (compare Listing 1 and 2). Compared to the first algorithm, which can permute elements from/to random positions, the second algorithm does not need any bit masks to refer to the active lanes. Instead, it is sufficient to pass in the number of active elements. In Listing 2, we refer to these numbers as `src_cnt` and `dst_cnt`. Based on these, the permutation indices, as well as the permutation mask, can be computed without any crosslane operations, such as compress/expand. A noteworthy property of the second SIMD algorithm is that the source vector remains in a compressed state even if not all elements fit into the destination vector.

Listing 1 Generic refill algorithm

```
struct fill_rr {
    __mmask8 permutation_mask;
    __m512i permutation_idxs;

    //Prepare the permutation.
    fill_rr(const __mmask8 src_mask,
            const __mmask8 dst_mask) {
        __m512i src_idxs = _mm512_mask_compress_epi64(
            ALL, src_mask, SEQUENCE);
        __mmask8 write_mask = _mm512_knot(dst_mask);
        permutation_idxs = _mm512_mask_expand_epi64(
            ALL, write_mask, src_idxs);
        permutation_mask = _mm512_mask_cmpneq_epu64_mask(
            write_mask, permutation_idxs, ALL);
    }

    //Move elements from 'src' to 'dst'.
    void apply(const __m512i src, __m512i& dst) const {
        dst = _mm512_mask_permutexvar_epi64(
            dst, permutation_mask, permutation_idxs, src);
    }

    void update_src_mask(__mmask8& src_mask) const {
        __mmask8 compressed_mask =
            _pext_u32(~0u, permutation_mask);
        __m512i a =
            _mm512_maskz_mov_epi64(compressed_mask, ALL);
        __m512i b =
            _mm512_maskz_expand_epi64(src_mask, a);
        src_mask =
            _mm512_mask_cmpeq_epu64_mask(src_mask, b, ZERO);
    }

    void update_dst_mask(__mmask8& dst_mask) const {
        dst_mask =
            _mm512_kor(dst_mask, permutation_mask);
    }
};
```

These two foundational SIMD algorithms cover the extreme cases where (i) active elements are stored at random positions and (ii) active elements are stored contiguously. Based on these cases, the algorithms can easily be adapted so that only one vector needs to be compressed, which is useful when vector registers are used as tiny buffers because those should always be in a compressed state to achieve the best performance. In total, there are four different algorithms. Each algorithm has two different flavors: (i) where all elements from the source register are guaranteed to fit into the destination register or (ii) where not all elements can be moved and therefore elements remain in the source register. We do

not show all variants here, but have released the C++ source code¹ under the BSD license.

Listing 2 Refill algorithm for compressed vectors

```

struct fill_cc {
  __mmask8 permutation_mask;
  __m512i permutation_idxxs;
  uint32_t cnt;

  //Prepare the permutation.
  fill_cc(const uint32_t src_cnt,
          const uint32_t dst_cnt) {
    const auto src_empty_cnt = LANE_CNT - src_cnt;
    const auto dst_empty_cnt = LANE_CNT - dst_cnt;
    //Determine the number of elements to be moved.
    cnt = std::min(src_empty_cnt, dst_empty_cnt);
    bool all_fit = (dst_empty_cnt >= src_cnt);
    auto d = all_fit ? dst_cnt : src_empty_cnt;
    const __m512i d_vec = _mm512_set1_epi64(d);
    //Note: No compress/expand instructions required
    permutation_idxxs =
      _mm512_sub_epi64(SEQUENCE, d_vec);
    permutation_mask = ((1u << cnt) - 1) << dst_cnt;
  }

  //Move elements from 'src' to 'dst'.
  void apply(const __m512i src, __m512i& dst) const {
    dst = _mm512_mask_permutexvar_epi64(
      dst, permutation_mask, permutation_idxxs, src);
  }

  void update_src_cnt(uint32_t& src_cnt) const {
    src_cnt -= cnt;
  }

  void update_dst_cnt(uint32_t& dst_cnt) const {
    dst_cnt += cnt;
  }
};

```

5 Refill strategies

We discuss the integration of these refill algorithms in data-centric *compiled* query pipelines. Such pipelines turn a query operator pipeline into a for-loop, and the code generated by the various operators is nested bottom-up in the body of such a loop [18]. Relational operators in this model generate code in two methods, namely, `consume()` and `produce()`, which are called in a depth-first traversal of the query tree: `produce()` code is generated before generating the code for the children, and `consume()` afterward.

The main idea of data-centric execution with SIMD is to insert checks for each operator that control the number of tuples in play, i.e., if-statements nesting the rest of the body. Such an if-statement ensures that its body only gets executed if the SIMD registers are sufficiently full. Generally speaking, operator code processes input SIMD data computed by the outer operator and *refills* the registers it works with and the ones it outputs.

We identify two *base strategies* for applying this refilling.

¹ Source code: https://github.com/harald-lang/simd_divergence.

5.1 Consume everything

The consume everything strategy allocates additional vector registers that are used to *buffer* tuples. In the case of under-utilization, the operator *defers* the processing of these tuples. This means the body will not be executed in this iteration (if-condition not satisfied) but instead (else) the active tuples will be moved to these buffer registers. It uses the refill algorithms from the previous section both to move data to the buffer and to emit buffered tuples into the unused lanes in a subsequent iteration. Listing 3 shows the code skeleton as it would be generated by such a *buffering* operator. The `THRESHOLD` parameter specifies when a refill is triggered during query execution. Depending on the situation, the costs for refilling might not amortize if only a few lanes contain inactive elements. But if the remaining pipeline is very expensive, setting the threshold to the number of SIMD lanes could be the best option. The important thing to note here is that all SIMD lanes are empty when the control flow returns to the previous operator, thus we call it *consume everything*.

Compared to a scalar pipeline, this strategy only requires a minor change to the push model: handling a special case when the pipeline execution is about to terminate, flushing the buffer(s). The essence is that buffering only takes place in SIMD registers and it specifically does not cause extra in-memory materialization.

Figure 6a, b illustrates the effects of applying a refill strategy to a query pipeline by visualizing the SIMD lane utilization over time. The structure of the query is similar to the one shown in Fig. 1 and consists of a scan, a selection, a join, and a sink to where the output is written. The *stage* indicator on top of the plot refers to the node in the control flow graph in Fig. 1. In Fig. 6a, the query is executed without divergence handling, and the white areas refer to under-utilization. Figure 6b visualizes the same workload with in-register buffering, following consume everything semantics. The purple and black vertical lines indicate that tuples are written to the buffers, or read from the buffer, respectively. Compared to the divergent implementation, the lane utilization has significantly increased, and the overall execution time has reduced. In this example, we require the utilization to be at least 75% (six out of eight lanes need to be active). Underutilization is observed only when the execution is about to finish, which triggers a pipeline flush, where all (potentially) buffered tuples need to be processed regardless of the minimum utilization threshold.

5.2 Partial consume

As the name suggests, the second base strategy no longer expects the `consume()` code to process the entire input. The consume code can decide to defer execution by returning the control flow to the previous operator and leave the active ele-

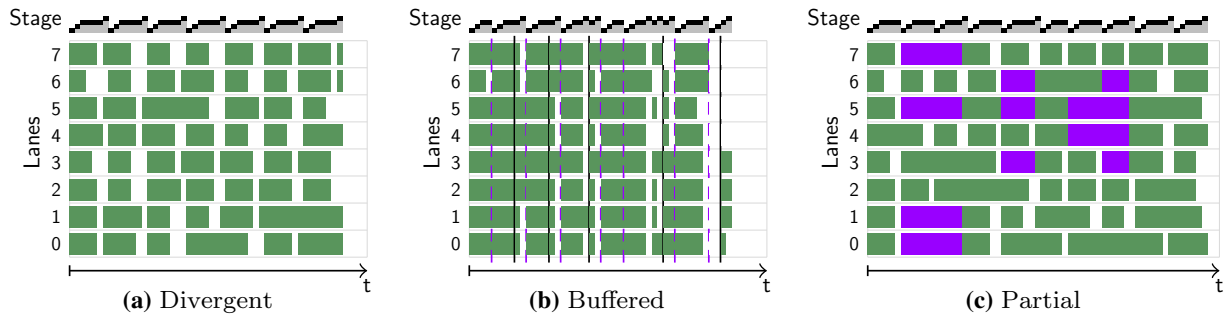


Fig. 6 SIMD lane utilization using different strategies. In **a**, no refilling is performed to visualize the divergence. **b** Uses the consume everything strategy, which performs refills when the utilization falls below 75%. The dashed purple lines indicate a write to buffer registers, black lines

a read. **c** shows a partial consume throughout the entire pipeline with the minimum required utilization set to 50%. Lanes colored in purple are protected (color figure online)

Listing 3 Code skeleton of a buffering operator.

```
[...]
auto active_lane_cnt = popcount(mask);
if (active_lane_cnt + buffer_cnt < THRESHOLD
    && !flush_pipeline) {
    [...]//Buffer the input.
}
else {
    const auto bail_out_threshold =
        flush_pipeline ? 0
            : THRESHOLD;
    while (active_lane_cnt + buffer_cnt >
        bail_out_threshold) {
        if (active_lane_cnt < THRESHOLD) {
            [...]//Refill lanes with buffered elements.
        }
        //=====//
        //The actual operator code and
        //consume code of subsequent operators.
        [...]
        //=====//
        active_lane_cnt = popcount(mask);
    }
    if (likely(active_lane_cnt != 0)) {
        [...]//Buffer the remaining elements.
    }
}
//All lanes empty (consume everything semantics).
mask = 0;
[...]
```

Listing 4 Code skeleton of a partial consume operator.

```
[...]
auto active_lane_cnt = popcount(mask);
if (active_lane_cnt < THRESHOLD && !flush_pipeline){
    //Take ownership of newly arrived elements.
    this_stage_mask = mask ^ later_stage_mask;
}
else {
    //=====//
    //The actual operator code and
    //consume code of subsequent operators.
    [...]
    //The later_stage_mask is set by the
    //consumer.
    //=====//
}
//Protect lanes in the preceding operator.
mask = this_stage_mask | later_stage_mask;
[...]
```

ments in the vector registers. New tuples are assigned only to inactive lanes by one of the preceding operators, typically a table scan. Naturally, the active lanes, that contain deferred tuples, must not be overwritten or modified by other operators. We refer to these elements (or to their corresponding lanes) as being *protected*. Another way of looking at a protected lane is that the lane is *owned* by a different operator. When an *owning* operator completes the processing of a tuple, it transfers the ownership to the subsequent operator. Alternatively, if the tuple is disqualified, it gives up ownership to allow a tuple producing operator to assign a new tuple to the corresponding lane.

Lane protection requires additional bookkeeping on a per operator basis. Each operator must be able to distin-

guish between tuples that (i) have just arrived, (ii) have been protected by the operator itself in an earlier iteration and (iii) tuples that have already advanced to later stages in the pipeline. To do so, an operator maintains two masks, one that identifies the lanes that are owned by the current operator and another one that identifies lanes that are owned by a later operator. Listing 4 shows the structure of such an operator, where `this_stage_mask` and `later_stage_mask` are part of the operator's state and `mask` is used to communicate which lanes contain active elements (regardless of their stage).

Figure 6c shows how the partial consume strategy affects the lane utilization with the minimum lane utilization threshold set to 50%. The lanes colored in purple are in a protected state. Compared to the divergent implementation, the lane utilization has increased. However, if we take protected lanes into account and consider them as idle, the overall utilization decreases. Thus, the example workload, used in Fig. 6, reveals an important drawback. If the lanes become protected in later stages of the pipeline, these lanes can cause signifi-

cant underutilization in the preceding operators. We discuss this issue, among other things, in the following section.

5.3 Discussion and implications

The two strategies are not mutually exclusive. Within a single pipeline, both strategies can be applied to individual operators as long as buffering operators are aware of protected lanes (*mixed* strategy). Moreover, the query compiler might decide to *not* apply any refill strategy to certain operators. Especially, when a sequence of operators is quite cheap, divergence might be acceptable as long as the costs for refill operations are not amortized. Naturally, this is a physical query optimization problem that we will leave for future work. Nevertheless, we briefly discuss the advantages and disadvantages, as this is the first work in which we present the basic principles of vector-processing in compiled query pipelines.

As mentioned above, *consume everything* requires additional registers, which increases the register pressure and may lead to spilling. *partial consume* allocates additional registers as well, but these are restricted to (smaller) mask registers. Therefore, it is unlikely to be affected by (potential) performance degradation due to spilling.

The second major difference lies in the cost of refilling empty lanes. In a pipeline that follows the partial consume strategy, the very first operator, that is, the pipeline source, is responsible for refilling empty lanes. If other operators experience underutilization, they return the control flow to the previous operator while retaining ownership of the active lanes. This cascades downward until the source operator is reached, as shown in Fig. 6c. All operators between the pipeline source and the operator that returned the control flow may be subject to underutilization because all lanes in later stages are protected. The costs of refilling, therefore, depend on the length of the pipeline and the costs of the preceding operators. In general, the costs increase in the later stages. Nevertheless, partial consume can improve query performance if it is applied only to the very first operators. By contrast, the refilling costs of buffering operators do not depend on the pipeline length. Instead, the crucial factor governing these costs is the number of required buffer registers. The greater the number of buffers, the greater the number of *permute* instructions that need to be executed, whereas the number of required buffers depends on (i) the number of attributes passed along the pipeline and optionally on (ii) the number of registers required to save the internal state of the operator (e.g., a pointer to the current tree node).

6 Evaluation

We evaluate our approach with two major sources of control flow divergence, (i) predicate evaluation as part of a

Table 1 Hardware platforms

	Intel Knights landing (KNL)	Intel Skylake-X (SKX)
Model	Phi 7210	i9-7900X
Cores (SMT)	64 (\times 4)	10 (\times 2)
SIMD [bit]	2×512	2×512
Max. clock rate [GHz]	1.5	4.5
L1 cache	64 KiB	32 KiB
L2 cache	1 MiB	1 MiB
L3 cache	–	14 MiB

table scan and (ii) a hash join. Additionally, we experiment with a more complex operator, an approximate geospatial join. The experiments were conducted on an Intel Skylake-X (SKX) and an Intel Knights Landing (KNL) processor (cf., Table 1). The experiments were implemented in C++ and compiled with GCC 5.4.0 at optimization level three (`-O3`) and the target architecture set to `knl`. If not stated otherwise, we ran the experiments in parallel using two threads per core.² We dispatched the work in batches to the individual threads using batch sizes between 2^{16} and 2^{20} tuples. On the KNL platform, we placed the data in high-bandwidth memory (HBM); otherwise, the experiments would have been dominated by memory stalls. To measure the throughputs, we let each experiment run for at least three seconds, possibly consuming the input data multiple times.

6.1 Table scan

To evaluate the effects of divergence handling in table scans, we integrate our refill algorithms into the AVX-512 implementation of TPC-H Query 1 of Gubner et al. [6]. Additionally, we implemented and integrated the *materialization* approach as proposed by Menon et al. in [16].

From a high-level perspective, TPC-H Query 1 (or short Q1) is a structurally simple query that operates on a single fact table (*lineitem*) with a single scan predicate. It involves several fixed-point arithmetic operations in the aggregation based on the group by clause. In total, five additional attributes are accessed to compute eight aggregated values per group. Almost all tuples survive the selection (i.e., selectivity ≈ 0.98). Therefore, in its original form, Q1 does not suffer from control flow divergence. To simulate control flow divergence and the resulting underutilization of SIMD

² Please note that throughout our (multi-threaded) experiments, we did not observe any performance penalties through downclocking. Both processors KNL and SKX run stable at 1.4GHz and 4.0GHz, respectively.

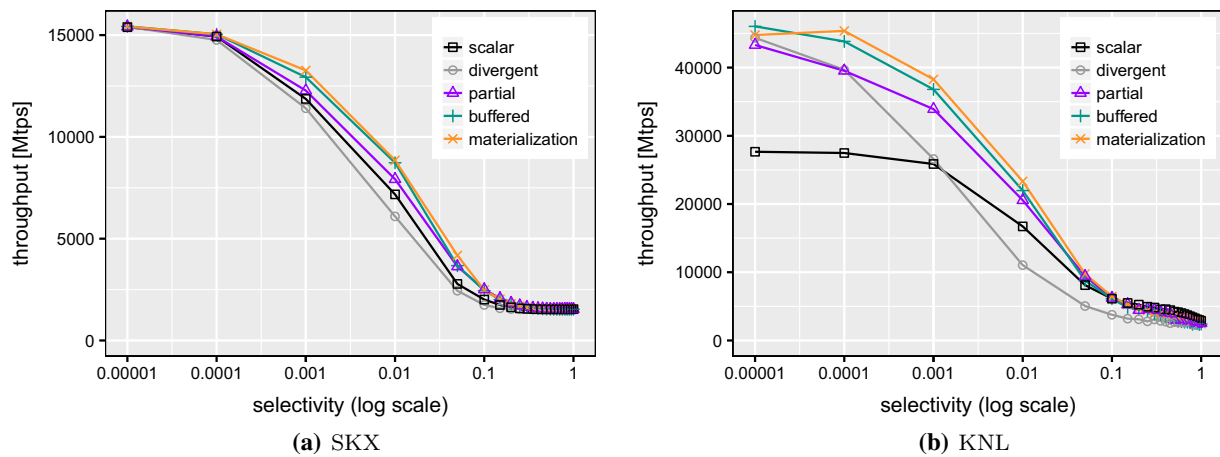


Fig. 7 Performance of TPC-H Q1 with varying selectivities

lanes, we vary the selectivity of the scan predicate on the `shipdate` attribute.

We evaluate and compare a scalar³ non-SIMD) implementation with four AVX-512 implementations:

Divergent: The divergent implementation refers to the implementation published by the authors of [6], with a minor modification. In the original version, all tuples are pushed through the query pipeline and disqualified elements are ignored in the final aggregation by setting the lane bitmask accordingly. For our experiments, we introduced a branch behind the predicate evaluation code, which allows to return the control flow to the scan operator iff all SIMD lanes contain disqualified elements. In the case of Q1, the predicate is evaluated on 16 elements in parallel.

Partial/Buffered: The *partial* and *buffered* implementations make use of our refill algorithms. A major difference to the *divergent* implementation is that it can no longer make use of aligned SIMD loads. Instead, it relies on the `gather` instruction to load subsequent attribute values. The select operator, therefore, produces a tuple identifier (TID) list that identifies the qualifying tuples. The subsequent operators use the TIDs to compute the offset from where to load the additional attributes. Both implementations are parameterized with the *minimum lane utilization threshold*, which limits the degree of underutilization.

Materialization: The *materialization* implementation makes use of small (memory) buffers to consecutively store the output. Similarly to our approach, the select operator produces a TID list. The code of the subsequent operator(s) is executed when the buffer is (almost) full.

³ Scalar refers to an implementation which does not use any SIMD instructions. We verified, that the compiler did not auto-vectorize the query pipelines.

The buffered TID list is then consumed (scanned) similarly to a table scan in the subsequent operator. Notably, the output contains only TIDs that belong to qualifying tuples, which is in contrast to our approach, where SIMD lanes may contain non-qualifying tuples, depending on the chosen threshold.

Figure 7a shows the performance results for varying selectivities (between 0.00001 and 1.0) on SKX. In the extreme cases, all implementations perform similarly. Interestingly, this includes the scalar implementation, which indicates that the SKX processor performs extremely well with respect to IPC, branch prediction, and out of order execution. With intermediate selectivities, divergence handling can make a significant difference. For instance, with $sel = 0.01$ the difference between the divergent and materialization implementation is 2.6 billion tuples per second (1.5 billion over scalar). The graph also shows that the materialization dominates over almost the entire range. Our approach (buffered) can compete, but is slightly slower in most cases. On KNL (Fig. 7b), we observed similar effects. The most important difference is that the divergent SIMD implementation is significantly slower than the scalar implementation with selectivities larger than 0.0001. Divergence handling extends the range in which SIMD optimizations become beneficial.

For this experiment, we varied the utilization threshold for partial and buffered as well as the buffer size for materialization and we reported only the best performing variant. In the following, we investigate the impact of these parameters. Figure 8a shows the performance of the materialization approach for varying buffer sizes and a fixed selectivity ($sel = 0.01$). Peak performance for Q1 is achieved with a memory buffer of size 1024 elements or larger.

In Fig. 8b, we vary the SIMD lane utilization threshold for our approaches. The performance of the buffered imple-

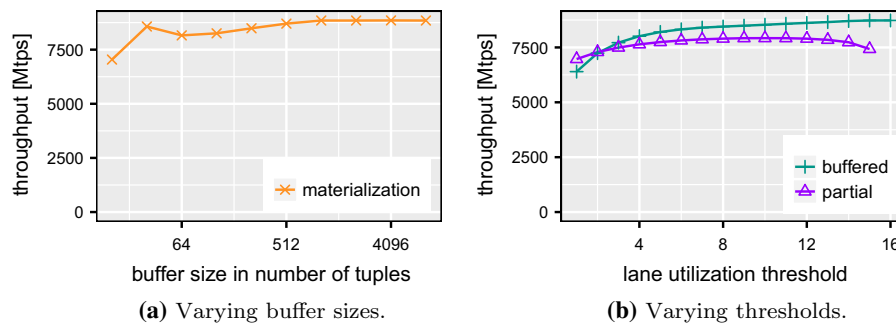


Fig. 8 Performance of TPC-H Q1 performance on SKX when varying algorithm parameters

mentation increases with the threshold. Peak performance is reached when only qualifying tuples pass the select operator (threshold = 16). But the performance only gradually increases for a threshold ≥ 6 . The reason for this behavior is that non-qualifying tuples only cause computational overhead in the remaining pipeline but no memory accesses, which would be significantly more expensive. On the other hand, the partial consume strategy favors a threshold that is approximately half the number of SIMD lanes. If the threshold is too low (left-hand side), many non-qualifying tuples pass the filter, and if it is set too high (right-hand side), the control flow is often returned to the scan code to fetch (a few) more values.

6.2 Hashjoin

Probing a hash table is a search operation in a pointer-based data structure and therefore a prime source of control flow divergence. Here, we evaluate the very common foreign-key join of two relations followed by (scalar) aggregations. The primary key relation constitutes the build size in such a way that the join is non-expanding, i.e., for a probe tuple, at most one join partner exists. The two input relations each have two 8-byte integer attributes: a key and a value. The relations are joined using the keys. Afterward, three aggregations are computed on the join result: the number of tuples, the sum of the values from the left input relation, and the sum of the values from the right input relation. Our hash table implementation stores the first key-value pair per hash bucket in the hash table dictionary. In case of collisions, additional key-value pairs are stored in a linked list per hash bucket.

We evaluate and compare a scalar (non-SIMD) implementation with four AVX-512 implementations:

Divergent: This SIMD implementation handles eight tuples in parallel. The lane bitmask is used to keep track of disqualified tuples, such that they can be ignored at the end of the pipeline. As in the table scan evaluation, we add a branch to allow for an early return to the beginning of the pipeline iff all SIMD lanes contain disqualified

tuples. We introduce this branch after the first hash table lookup, i.e., it is triggered when all probe tuples fall into empty hash buckets.

Partial/Buffered: These implementations make use of our in-register refill algorithms. In contrast to the table scan discussed in 6.1, the hash table example uses only few relation attributes. Therefore, instead of loading the additional attributes using `gather`, here all attributes (i.e., key and value) are passed through the pipeline. If the number of active SIMD lanes drops below the *minimum lane utilization threshold*, a refill is performed.

Materialization: Menon et al. [16] propose operator fusion, which introduces buffers between operators to compact the stream of tuples flowing through a pipeline. Here, we introduce an *intra-operator* buffer to further densify the stream of tuples. At the beginning of the pipeline, we load key-value pairs from the probe side input, and compute the hash value and the pointer to the hash table dictionary. We store these key-value pairs and pointers in an input buffer. From this buffer, we then lookup eight pointers in the hash table in parallel, and determine if (i) we found a match, (ii) we need to follow a chain (further), or (iii) there is no match. Unfinished tuples (case (ii)) are written back into the input buffer with an updated pointer; matching tuples (case (i)) are directly pushed to the subsequent aggregation operator without further buffering, which is not in line with [16], where materialization happens on operator boundaries. We also implemented a “fully” materialized version where the matches are first stored in an output buffer before the aggregation code is executed. However, our experiments have shown that two memory materializations are more expensive.

Figures 9 and 10 show the performance results for varying hash table sizes (between 10 KiB and 45 MiB). The hash table size is chosen depending on the build input size. We size the hash table dictionary so that it has the same number of buckets as there are build tuples. Among all evaluated approaches, as

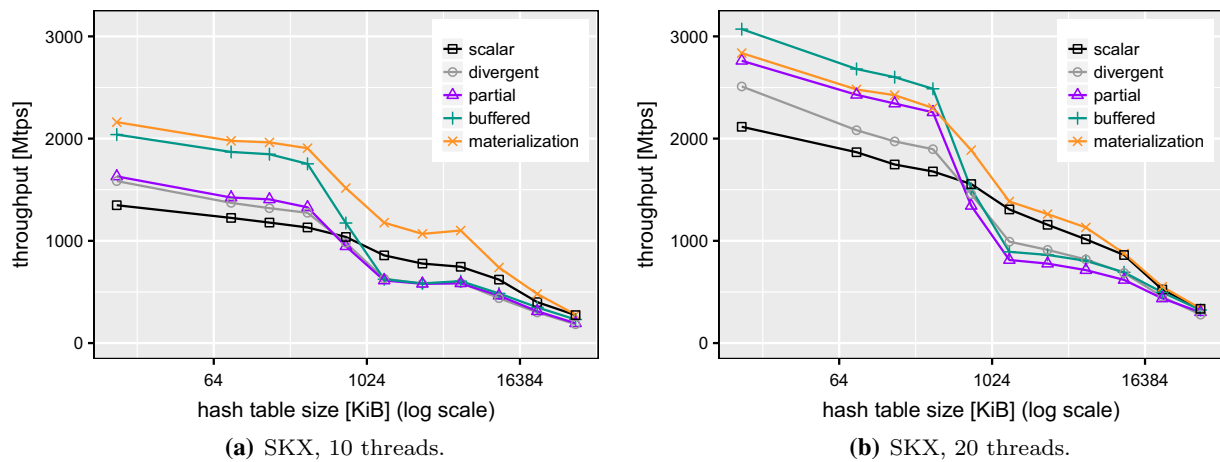


Fig. 9 Hashjoin performance when varying build sizes. SKX

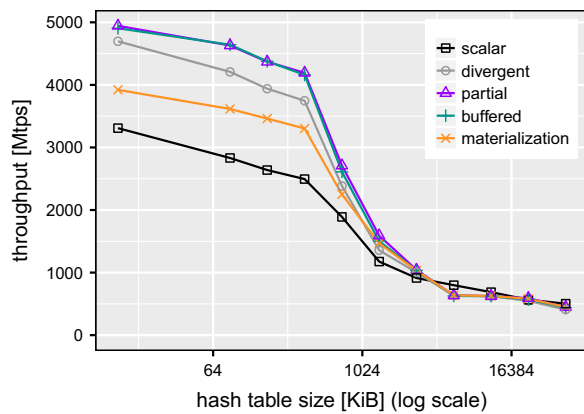


Fig. 10 Hashjoin performance when varying build sizes. KNL, 128 threads

well as both platforms, the throughput shrinks with growing hash table sizes. The overall throughput on Knights Landing is about twice as high as on Skylake-X, even though the performance of Skylake-X can be increased by 50% by using Hyper-Threading. On Skylake-X (Fig. 9a, b), as expected, a sharp performance decrease happens when the hash table grows beyond the size of the L2 cache at around 1 MiB, and at around 10 MiB when it exceeds the L3 cache. For large hash tables, that do not fit into the cache, all approaches converge. This has also been observed in earlier work, for instance, by Polychroniou et al. [21] and by Kersten et al. [9]. In these cases, partitioning the hash table might help (cf. the radix partitioning join proposed by Kim et al. [10]), but this is out of scope for this paper.

When the hash table is small enough to fit into the L1 or L2 cache, all SIMD approaches outperform the scalar baseline: Irrespective of the SIMD divergence handling deployed by the individual approaches, they all reach a higher through-

put than the scalar approach. For larger hash tables, SIMD divergence no longer dominates the performance, and thus the scalar approach reaches similar throughput levels (using 10 threads) or even higher throughput (using 20 threads) than some SIMD variants. For the whole evaluated range of hash table sizes, the partial and buffered approaches that make use of the introduced refill strategies outperform or are on par with the divergent SIMD approach. Using 20 threads, the buffered approach achieves up to 32% higher throughput than the divergent approach, while the partial approach outperforms the divergent one by up to 19%.

When the hash table fits into the L1 cache, the buffered approach defeats the materialization approach by up to 8%. When the hash tables grow, the materialization approach dominates all other approaches. Two contradicting influences determine whether the materialization approach outperforms our divergence-handling approaches: First, the materialization approach can hide memory latencies better than the buffered and partial approaches because more memory is accessed at the same time (i.e., multiple outstanding loads). This is shown in Fig. 9a and more severely in Fig. 9b when the hash table is large, because it then resides in slower memory. Second, the materialization approach suffers from the higher number of issued instructions, i.e., load and store instructions. In particular, when the hash table fits into L1, the number of instructions can become the limiting factor. On the Knights Landing platform in particular, the materialization approach has a significantly lower throughput compared to the other SIMD variants (Fig. 10). In contrast to the table scan, which we evaluated in Sect. 6.1, the materialization buffer is read and written in the same loop multiple times—during index lookup, which exceeds the limited out-of-order execution capabilities of KNL.

In Fig. 11a, b, we vary the SIMD lane utilization threshold for the partial and buffered approaches. Hyper-Threading,

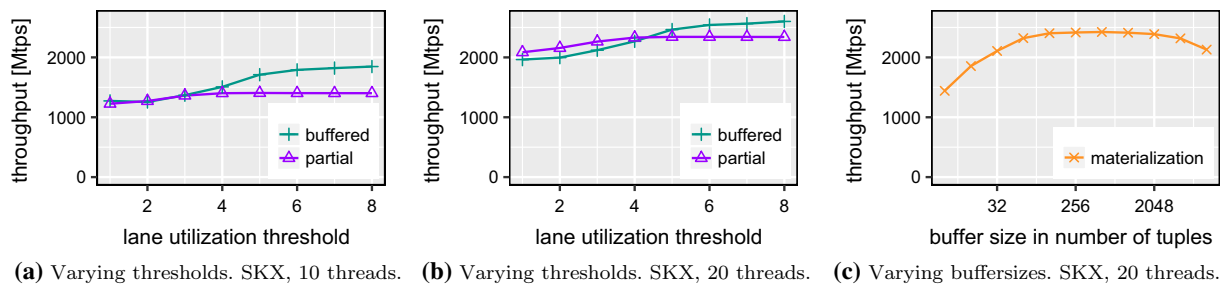


Fig. 11 Hashjoin performance when varying algorithm parameters

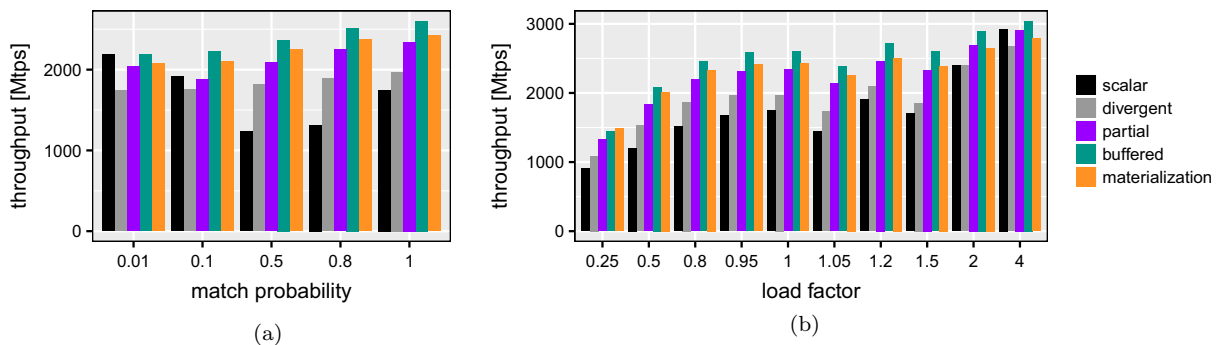


Fig. 12 Hashjoin performance for varying match probabilities (a), hash table load factors (b). SKX, 20 threads

i.e., using 20 threads instead of 10, increases throughput by about 50%. In general, a higher threshold, i.e., less inactive SIMD lanes and more refills, results in a higher throughput. There is little change in throughput when setting the threshold to six, seven or all eight tuples. This is because in most hash table lookups, only a few of the eight tuples need to be kept for additional pointer lookups in the collision chains. As a result, almost no refills are done differently when choosing six, seven or eight as the threshold. The buffered approach is more sensitive to the chosen threshold. For a low threshold, the partial approach reaches a higher throughput, but that changes at threshold 3 (using 10 threads) or 5 (using 20 threads). As mentioned, only few tuples need to be kept for additional lookups. Thus, only few tuples need to be buffered in the buffered approach, while the partial approach suffers from underutilization when frequently performing refills in the table scan.

Figure 11c focuses on the materialization approach, varying the buffer size between eight and 8192 tuples and a fixed build cardinality (hash table size ≈ 128 KiB). For the chosen configuration, the scalar approach reaches a throughput of 1747 Mtps. For small buffers, e.g., 8 tuples, the scalar approach outperforms the materialization approach. The materialization approach with an 8-tuple buffer is conceptually similar to the buffered approach with a SIMD lane utilization threshold of 1. Both use a buffer the size of one SIMD vector. In the buffered approach, this buffer lives

in registers, while the materialization approach stores it in memory. As a result, the buffered approach outperforms the materialization approach with a throughput of 1964 Mtps (i.e., about 2 billion tuples per second). A buffer size between 128 and 1024 results in the best performance of the materialization approach. The throughput shrinks gracefully when the buffer size is further increased. This is an effect of the chosen workload, especially the number of attributes beside the join attribute.

Two additional parameters affect throughput in the hashjoin evaluation: First, the *match probability* describes how likely a tuple from the probe side finds a join partner in the hash table. We vary this probability between 0.01 and 1. A low match probability, therefore, results in more disqualified tuples, which—depending on the approach—in turn leads to more ignored SIMD lanes, more refills, or a worse VPU utilization. Figure 12a shows that the buffered approach, using the proposed refill strategies, outperforms both pre-existing approaches, scalar and divergent, irrespective of the match probability. The scalar approach is competitive with the SIMD approaches for low match probabilities. With few matches, the scalar approach can often exit the pipeline early, which leads to the high throughput rates we observed. The divergent approach, on the other hand, suffers from extreme underutilization because frequently only few SIMD lanes stay active due to the low match probability. With a match probability of 50%, branches are mispredicted in the scalar

approach, and its throughput subsequently tanks. When the match probability approaches 100%, almost all probe tuples find a non-empty hash bucket that then needs to be inspected further. Furthermore, the final aggregation becomes more expensive as more tuples make it into the join result. The scalar approach, therefore, performs best for low match probabilities and worst for a match probability of around 50%, and cannot fully recover its throughput even for a match probability of 100%. When looking at the SIMD approaches, we observe that the throughput difference between the divergent approach and our novel refill approaches increases with the match probability. A higher match probability comes along with more active SIMD lanes after the first lookup in the hash dictionary. Then, more divergence happens because these tuples will have to traverse collision chains of different lengths. The divergence-handling buffered and partial approaches can, therefore, outperform the divergent approach for high match probabilities.

Second, we define the hash table's *load factor* as the number of buckets in the hash table divided by the number of keys stored in the hash table. While the load factor has been kept constant in all previous experiments ($= 1.0$), in real scenarios, the hash table size is not only determined by the size of the build side input, but also by the set load factor. With a low load factor, more collisions in the hash table occur, resulting in longer chains. With longer chains, the variance of the number of pointers that need to be followed to perform the hash table probe increases. This variance directly translates to higher SIMD divergence. A low load factor, therefore, leads to worse VPU utilization in the divergent approach, which can then be mitigated by applying the proposed in-register refill strategies. Figure 12b shows how the load factor affects the throughputs reached by the different approaches. The hash table for load factor 4 is 16 times as big as the hash table for load factor 0.25. Over all approaches, the throughput of the bigger hash table is about three times as high as for the smaller one. For high load factors, the scalar approach performs well. This is because for high load factors, fewer and shorter collision chains exist. When zero of the eight tuples in a vector need to follow a chain, there is not SIMD divergence. Subsequently, for especially high load factors like 4, there is little difference between all approaches.

6.3 Approximate geospatial join

In the following, we evaluate and compare our approach with a modern and more complex operator, an approximate geospatial point-polygon join. Our approximate geospatial join [11] uses a quadtree-based hierarchical grid to approximate polygons. Figure 13 shows such an approximation for the neighborhoods in New York City (NYC). The grid cells are encoded as 64-bit integers and are stored in a specialized radix tree, where the cell size corresponds to the level within



Fig. 13 Quadtree-based cell-approximation of neighborhood polygons in NYC

the tree structure (larger cells are stored closer to the root node and vice versa). During join processing, we perform (prefix) lookups on the radix tree. Each lookup is separated into two stages: First, we check for a *common prefix* of the query point and the indexed cells. The common prefix allows for the fast *filtering* of query points. If the query point does not share the common prefix, there are no join partners. The actual tree traversal takes place in the second stage. We traverse the tree starting from the root node until we hit a leaf node (which contains a reference to the matching polygon).

An important property of our approximate geospatial join operator is that it can be configured to guarantee a certain precision. In the experiments, we used 60-, 15-, and 4-meter precision (as in [11]). The higher the precision guarantee, the smaller are the cells at the polygon boundaries, which in turn increases the total number of cells and, more importantly, the height of the radix tree. In general, the probability of control flow divergence during index lookups increases with the tree height. Throughout our experiments, the tree height is ≤ 6 .

In our experiments, we join the boroughs, neighborhoods, and census blocks polygons of NYC⁴ with randomly generated points, uniformly distributed within the minimum bounding box of the corresponding polygonal dataset. The datasets vary in terms of the total number of polygons and complexity (with respect to the number of vertices).

Table 2 summarizes the relevant metrics of the polygon datasets, and Table 3 summarizes the metrics of the corresponding radix tree, including the probability distribution of the number of search steps during the tree traversal.

⁴ The polygons of NYC are available at:

- <https://data.cityofnewyork.us/City-Government/Borough-Boundaries/tqmj-j8zm>
- <https://data.cityofnewyork.us/City-Government/Neighborhood-Tabulation-Areas/cpf4-rkhq>
- <https://data.cityofnewyork.us/City-Government/2010-Census-Blocks/v2h8-6mxf>.

Table 2 Polygon datasets

	Number of polygons	Avg. number of vertices
Boroughs	5	662.2
Neighborhoods	289	29.6
Census	39,184	12.5

6.3.1 Query pipeline

The query pipeline of our experiments (point-polygon join) consists of four stages:

- (1) Scan point data (source)
- (2) Prefix check
- (3) Tree traversal
- (4) Output point-polygon pairs (sink)

Stages (2) and (3) are subject to control flow divergence, with (3) being significantly costlier than (2). For simplicity, the produced output (point-polygon pairs) is not further processed. We compile the pipeline in three different flavors:

Divergent: Refers to the baseline pipeline without divergence handling, thus the pipeline follows consume everything semantics. The code of subsequent operators is executed if at least one lane is active.

Partial: The partial consume strategy is applied to stages (2) and (3), which also affects the scan operator because it needs to be aware of protected lanes.

Buffered: Follows consume everything semantics with register buffers in stage (3). We check the lane utilization after each traversal step. Divergence in stage (2) is not handled at all.

Materialization: The integration of memory materialization is similar to the one used with the hash join operator (cf., Sect. 6.2).

6.3.2 Results

Figure 14 shows the performance results in million tuples per second on KNL using 128 threads. We observe that refilling from register buffers improves the overall throughput by up to 20% (= 870 mtps) when joining with the boroughs or neighborhood polygons. The effect of divergence handling falls below 10% with the census blocks polygons where the index structure is more than 1 GiB in size. In that case, the memory subsystem is the limiting factor.

As expected, the partial consume strategy exacerbates the divergence issue in most cases (cf., Sect. 5.3), resulting in a 53% performance degradation in the worst case.

The materialization approach performs poorly on KNL. The throughput is similar to the scalar implementation, thus canceling out all SIMD optimizations. As in previous benchmarks, we observed a significantly better performance on SKX. Here, the materialization approach is on par with the buffered pipeline: in case of small index structures

Table 3 Metrics of radix tree

Polygons	Boroughs			Neighborhoods			Census		
	60	15	4	60	15	4	60	15	4
Precision [m]	60	15	4	60	15	4	60	15	4
# of cells [M]	0.08	1.27	20.7	0.11	0.79	13.2	6.08	6.52	34.6
Tree size [MiB]	1.39	168	168	25.3	139	139	1162	1205	1205
Tree traversal depth									

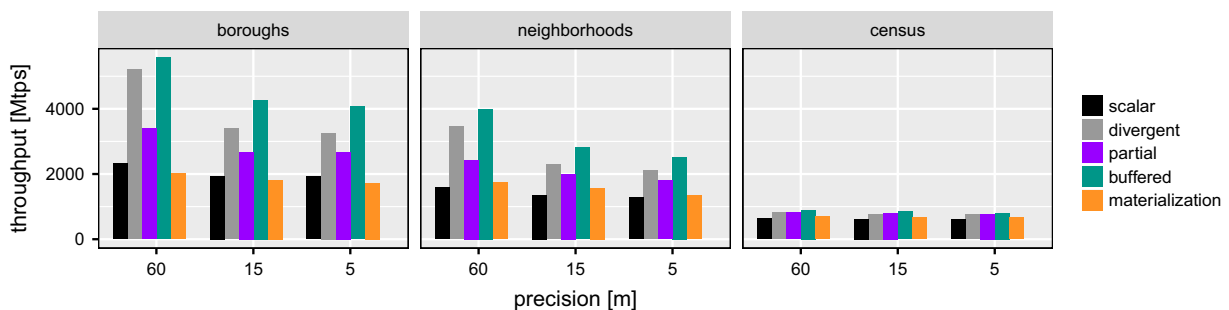


Fig. 14 Geospatial join performance for varying workloads and precisions

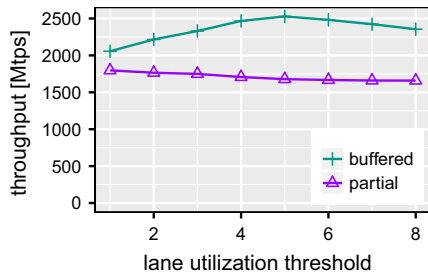


Fig. 15 Varying thresholds. KNL, 128 threads

(boroughs) slightly worse, and with large indexes (census) slightly better. In the latter case, the materialization approach helps to hide memory latencies through out-of-order execution.

Unlike the previous experiments, the optimal lane utilization threshold for the buffered approach is less than the number of SIMD lanes (cf., Fig. 15), which is due to the higher refilling costs involved in the geojoin operator. During the radix tree traversal, refilling affects five vector registers, whereas in the hash join experiment, refilling affects three registers; and only one in the table scan experiment. The optimal threshold for the partial approach is 1, indicating that a refill from the pipeline source is not efficient.

In the experiment above, all points pass the prefix check stage (2) and therefore cause a radix tree traversal. In the following, we also apply divergence handling on the second stage of the pipeline and we changed the workload so that a certain amount of points are disqualified in that stage. We compiled the query pipeline with several combinations of the different approaches. We refer to it using the first letter of the approach (**D**ivergent, **B**uffered, **P**artial, and **M**aterialization). For instance, **PB** refers to the pipeline that uses partial consume in stage two and in-register buffering in the third stage, and **BB** uses buffering in both stages. Figure 16 shows the results for the neighborhood/4 meter precision workload with varying selectivities. We observe an 8% performance decrease when the buffered approach is applied to stages 2 and 3, and the selectivity remains at 1.0. In contrast, the materialization approach adds a significantly larger overhead (35% decrease). If materialization is applied in both pipeline stages, the performance is worse compared to the pipeline, where it is applied only in the tree traversal stage. Overall, the performance difference for lower selectivities is relatively small with the partial and buffered approaches: +5% with buffering applied in both stages, -7% when partial consume is applied in stage 2 and buffering in stage 3. Compared to the divergent pipeline, lane refilling increases the throughput of the neighborhood workload by up to 30% with lower selectivities.

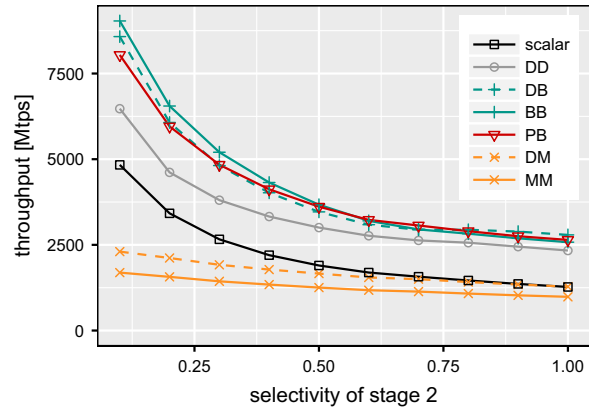


Fig. 16 2-Way divergence handling

6.4 Overhead

In our final experiment, we evaluate the overhead of divergence handling with a varying number of attributes. To quantify the overhead, we use a very simplistic query that consists of a simple selection and a scalar aggregation (`select sum(a1), sum(a2), ..., sum(aN) from...`). Divergence is handled immediately after the selection and before the aggregation. In that scenario, we expect the divergent pipeline to perform best, as the remainder of the pipeline only consists of a single addition and thus the benefits of refilling are close to zero.

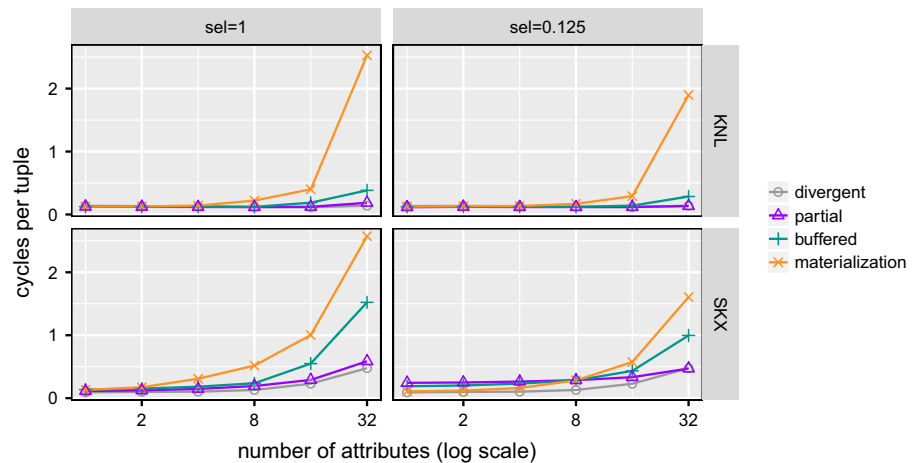
In the following, we consider two different selectivities:

sel = 1: For in-register buffering, this situation is the one with the lowest overhead, as the tuples are passed through to the subsequent operator and the buffer registers are not used altogether (cf. Listing 3). Thus, the overhead is rather small, as it effectively consists of a `popcount` to determine the number of active lanes and a branch instruction. The same applies for partial consume pipelines.

sel = 0.125 = 1/LANE_CNT: A selectivity of `1/LANE_CNT` results in one active lane per iteration (on average) and thus represents the most write-intensive case for in-register buffering. That is, the refill algorithm, which moves active elements to the buffer registers, is executed in almost every iteration. The partial consume strategy, on the other hand, suffers from lane underutilization caused by lane protection, and thus, the lower part of the pipeline is executed more frequently.

Throughout all experiments, the pipelines are 8-way data-parallel and we set the minimum lane utilization threshold to 6 for buffered and 4 for partial; the size of memory buffers are

Fig. 17 Overhead of divergence handling for varying number of attributes



fixed to 1024 elements (= 8 KiB). The number of attributes is varied within the range [1, 32].

Figure 17 summarizes the results for both evaluation platforms. On KNL, all approaches perform similarly with up to four attributes and the overhead, i.e., the performance difference to the divergent pipeline is barely measurable. The materialization approach degrades significantly when the number of attributes increases (2.5 CPU cycles per tuple per thread compared to 0.14 cycles for divergent). The throughput of the buffered approach degrades as well, which is also attributed to memory materializations. The high register file pressure forces the compiler to evict values to memory. Even though the buffer registers are not used in the case of $sel = 1$, register allocation is static and happens at query compilation time when the actual selectivity is not known. Therefore, a performance degradation can be observed even if register buffers are not used at query runtime. In contrast, the partial consume pipelines are on par with the divergent pipelines.

On the SKX platform, the performance degrades more steeply with an increasing number of attributes. In case of $sel = 1$, the throughput of the materialization approach decreases linearly with the number of attributes. Compared to KNL, the number of attributes has a higher impact on the overall performance on SKX. For instance, in-register buffering is $4 \times$ faster on KNL with $sel = 1$ and $3.6 \times$ faster with $sel = 0.125$. For $sel = 0.125$ and a single projected attribute, we measure an overhead of approximately 0.1 cycles per tuple for buffered and 0.15 cycles per tuple for partial, which is significantly higher than with the materialization approach (0.02 cycles). However, the per attribute overhead of buffered and partial decreases with more projected attributes, whereas the materialization approach shows an increasing overhead with an increasing number of attributes. The crossover point is reached with 8 projected attributes. Afterward, our approaches are consistently faster.

In general, the partial consume approach shows no performance impact when the number of projected attributes increases, which is an expected result, because the bookkeeping overhead about protected lanes is constant, irrespective from the number of projected attributes. The actual overhead of the partial consume strategy depends on the pipeline costs, more precisely on the pipeline fragment before divergence handling (see Sect. 5.3).

7 Summary and discussion

The partial consume strategy shows performance improvements for relatively simple workloads. With more complex workloads, like the geospatial join, we observe severe performance degradations. The reason for that is twofold. (i) Protected lanes inherently cause the underutilization of VPU (as described in Sect. 5) and (ii) they result in a suboptimal memory access pattern at the pipeline source where the refill happens. In contrast to the consume everything strategy, wherein every iteration exact `LANE_CNT` elements are read from memory, a partial consume scan reads *at most* `LANE_CNT` elements. This circumstance reduces the degree of data-parallelism (fewer elements are loaded per instruction) and also leads to unaligned SIMD loads. Even though the access pattern is still sequential, the alignment issues can reduce the load throughput by up to 25% (on our evaluation platforms), which could severely reduce the overall performance of scan-heavy workloads.

We found that the materialization approach is very sensitive to the underlying hardware, in particular, on KNL, the approach performs poorly when the buffer is read *and* written within a tight loop (intra-operator), an effect that could not be observed on SKX. On the other hand, if materialization is applied at operator boundaries and thus written and read only once, it performs similarly or better than in-register

buffering, as it benefits from out-of-order execution, which allows the materialization approach to hide memory latencies. Memory access latencies play an important role when the data that is randomly accessed (like a hash table) does not fit into the *L1/L2* cache. In contrast, when the data fits into cache or the workload is more compute-heavy, the in-register buffering approach dominates because the buffers provide much faster access.

The SIMD lane utilization threshold (refill more often vs. VPU underutilization) has a big impact on the buffered approach and less impact on partial. As buffered shows better performance in general, this parameter is important. Choosing the highest possible threshold shows the best results in simple workloads, so going back down the pipeline to refill the vector is always better than having inactive lanes, we found. So the idea of materialization, where only active (or qualifying) elements are passed along the pipeline, was right in these scenarios. The picture changes with more complex operators like the *geojoin*, where refilling affects five vector registers. In this case, refilling doesn't pay off for a single idle SIMD lane. On average, the optimal utilization threshold was 5 out of 8 among the geospatial related experiments.

It remains an open question how the optimal threshold can be predicted at query compilation time, as it depends on hardware, refilling costs, the costs incurred by underutilized lanes, and the actual input data. A possible approach to address this issue is to adaptively adjust the threshold parameter at runtime (per batch or per morsel [14]). Nevertheless, divergence handling cannot fully be disabled once the pipeline has been compiled. One can set the threshold to 1, which is equivalent to a divergent execution, but some overhead remains in the compiled code, namely the population count instruction and the branching logic. For instance, in our geospatial experiments on KNL, we observed an overhead of up to 6% over divergent with the *boroughs* workload when the utilization threshold is set to one (*neighborhoods* 3.6%, *census* 0.6%). Dynamically adjusting the threshold at query runtime provides some flexibility but due to the fact that divergence handling cannot be fully disabled, a database system needs to decide at compilation time whether to enable or disable divergence handling altogether.

Finally, we want to point out that our proposed refill algorithms and strategies are generally applicable to any data processing system that uses AVX-512 SIMD instructions. A prominent open-source representative is Apache Arrow [1] (in combination with *Gandiva*) which shares many similarities with state-of-the-art relational database systems (e.g., columnar storage, JIT-compilation, and operator fusion). Further, our approaches are also applicable if the underlying database system uses compression in its storage layer. In particular, when compression is only used on secondary storage, it does not affect query execution. However, recent systems [13,19] tend to use lightweight compression

techniques that allow for the processing of data without explicitly decompressing it. This implies that the degree of data-parallelism can be increased, as more attributes can be packed into a single vector register. Currently, our buffered approach is limited to 16-way data-parallelism on the KNL and SKX platforms, but it can be easily extended to 64-way data-parallelism for upcoming processors with the AVX-512/VBMI2 instruction set.

8 Conclusions

In this work, we presented efficient refill algorithms for vector registers by using the latest SIMD instruction set, AVX-512. Further, we identified and presented two basic strategies for applying refilling to compiled query pipelines for preventing the underutilization of VPUs. Our experimental evaluation showed that our strategies can efficiently handle control flow divergence. In particular, query pipelines that involve traversing irregular pointer-based data structures, like hash tables or radix trees, can significantly benefit from divergence handling. Especially when the workload is compute-intense or fits into fast caches, our novel approach shows better performance than existing approaches that rely on memory buffers.

Nevertheless, our research also showed that SIMD still cannot live up to the high expectations set by the promising features of the latest hardware, i.e., providing *n*-way data-parallelism. In practice, SIMD speedups are only a fraction of the advertised degree of data-parallelism, for many reasons, including underutilization. Our refill algorithms address this important reason, yet merely achieve a $2\times$ speedup over scalar code.

Acknowledgements This work has been 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. Furthermore, this work is part of the TUM Living Lab Connected Mobility (TUM LLCM) Project and has been partially funded by the Bavarian Ministry of Economic Affairs, Energy and Technology (StMWi) through the Center Digitization.Bavaria, an initiative of the Bavarian State Government.

Open Access This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

References

1. Apache Arrow. <https://arrow.apache.org/>
2. Balkesen, C., Alonso, G., Teubner, J., Özsu, M.T.: Multi-core, main-memory joins: sort vs. hash revisited. *PVLDB* 7(1), 85–96 (2013)

3. Balkesen, C., Teubner, J., Alonso, G., Özsu, M.T.: Main-memory hash joins on multi-core CPUs: tuning to the underlying hardware. In: 29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8–12, 2013, pp. 362–373 (2013). <https://doi.org/10.1109/ICDE.2013.6544839>
4. Boncz, P.A., Zukowski, M., Nes, N.: MonetDB/X100: hyper-pipelining query execution. In: CIDR 2005, 2nd Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4–7, 2005, Online Proceedings, pp. 225–237 (2005). <http://cidrdb.org/cidr2005/papers/P19.pdf>
5. Chhugani, J., Nguyen, A.D., Lee, V.W., Macy, W., Hagog, M., Chen, Y., Baransi, A., Kumar, S., Dubey, P.: Efficient implementation of sorting on multi-core SIMD CPU architecture. PVLDB 1(2), 1313–1324 (2008)
6. Gubner, T., Boncz, P.: Exploring query compilation strategies for JIT, vectorization and SIMD. In: 8th International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures, ADMS 2017, Munich, Germany, September 1, 2017 (2017). <https://stackoverflow.com/questions/36932240/avx2-what-is-the-most-efficient-way-to-pack-left-based-on-a-mask> (2016)
8. Kemper, A., Neumann, T.: Hyper: a hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In: Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11–16, 2011, Hannover, Germany, pp. 195–206 (2011). <https://doi.org/10.1109/ICDE.2011.5767867>
9. Kersten, T., Leis, V., Kemper, A., Neumann, T., Pavlo, A., Boncz, P.A.: Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. PVLDB 11(13), 2209–2222 (2018). <https://doi.org/10.14778/3275366.3275370>
10. Kim, C., Sedlar, E., Chhugani, J., Kaldewey, T., Nguyen, A.D., Blas, A.D., Lee, V.W., Satish, N., Dubey, P.: Sort vs. hash revisited: fast join implementation on modern multi-core CPUs. PVLDB 2(2), 1378–1389 (2009)
11. Kipf, A., Lang, H., Pandey, V., Persa, R.A., Boncz, P., Neumann, T., Kemper, A.: Approximate geospatial joins with precision guarantees. In: 34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16–19, 2018 (2018)
12. Lang, H., Mühlbauer, T., Funke, F., Boncz, P.A., Neumann, T., Kemper, A.: 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, pp. 311–326 (2016). <https://doi.org/10.1145/2882903.2882925>
13. Lang, H., Mühlbauer, T., Funke, F., Boncz, P.A., Neumann, T., Kemper, A.: Data blocks: hybrid OLTP and OLAP on compressed storage using both vectorization and compilation. In: Özcan, F., Koutrika, G., Madden, S., (eds.) Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26–July 01, 2016, pp. 311–326. ACM, San Francisco (2016). <https://doi.org/10.1145/2882903.2882925>
14. Leis, V., Boncz, P.A., Kemper, A., Neumann, T.: Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In: Dyreson, C.E., Li, F., Özsu, M.T. (eds.) International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22–27, 2014, pp. 743–754. ACM, New York (2014). <https://doi.org/10.1145/2588555.2610507>
15. Lemire, D., Rupp, C.: Upscaledb: efficient integer-key compression in a key-value store using SIMD instructions. Inf. Syst. 66, 13–23 (2017). <https://doi.org/10.1016/j.is.2017.01.002>
16. Menon, P., Pavlo, A., Mowry, T.C.: Relaxed operator fusion for in-memory databases: making compilation, vectorization, and prefetching work together at last. PVLDB 11(1), 1–13 (2017)
17. Mühlbauer, T., Rödiger, W., Seilbeck, R., Reiser, A., Kemper, A., Neumann, T.: Instant loading for main memory databases. PVLDB 6(14), 1702–1713 (2013)
18. Neumann, T.: Efficiently compiling efficient query plans for modern hardware. PVLDB 4(9), 539–550 (2011)
19. Nowakiewicz, M., Boutin, E., Hanson, E., Walzer, R., Katipally, A.: BIPie: fast selection and aggregation on encoded data using operator specialization. In: Proceedings of the 2018 International Conference on Management of Data, SIGMOD'18, pp. 1447–1459. ACM, New York (2018). <https://doi.org/10.1145/3183713.3190658>
20. Polychroniou, O., Raghavan, A., Ross, K.A.: Rethinking SIMD vectorization for in-memory databases. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31–June 4, 2015, pp. 1493–1508 (2015). <https://doi.org/10.1145/2723372.2747645>
21. Polychroniou, O., Raghavan, A., Ross, K.A.: Rethinking SIMD vectorization for in-memory databases. In: Proceedings of SIGMOD, pp. 1493–1508 (2015). <https://doi.org/10.1145/2723372.2747645>
22. Polychroniou, O., Ross, K.A.: Vectorized bloom filters for advanced SIMD processors. In: 10th International Workshop on Data Management on New Hardware, DaMoN 2014, Snowbird, UT, USA, June 23, 2014, pp. 6:1–6:6 (2014). <https://doi.org/10.1145/2619228.2619234>
23. Polychroniou, O., Ross, K.A.: 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, pp. 9:1–9:6 (2015). <https://doi.org/10.1145/2771937.2771943>
24. Ren, B., Agrawal, G., Larus, J.R., Mytkowicz, T., Poutanen, T., Schulte, W.: SIMD parallelization of applications that traverse irregular data structures. In: Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2013, Shenzhen, China, February 23–27, 2013, pp. 20:1–20:10 (2013). <https://doi.org/10.1109/CGO.2013.6494989>
25. Sitaridi, E.A., Polychroniou, O., Ross, K.A.: Simd-accelerated regular expression matching. In: Proceedings of the 12th International Workshop on Data Management on New Hardware, DaMoN2016, San Francisco, CA, USA, June 27, 2016, pp. 8:1–8:7 (2016). <https://doi.org/10.1145/2933349.2933357>
26. Teubner, J., Müller, R.: How soccer players would do stream joins. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12–16, 2011, pp. 625–636 (2011). <https://doi.org/10.1145/1989323.1989389>
27. Zhao, W.X., Zhang, X., Lemire, D., Shan, D., Nie, J., Yan, H., Wen, J.: A general simd-based approach to accelerating compression algorithms. ACM Trans. Inf. Syst. 33(3), 15:1–15:28 (2015). <https://doi.org/10.1145/2735629>
28. Zhou, J., Ross, K.A.: Implementing database operations using SIMD instructions. In: Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, June 3–6, 2002, pp. 145–156 (2002)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Tree-Encoded Bitmaps

Harald Lang

Technical University of Munich
harald.lang@tum.de

Alexander Beischl

Technical University of Munich
beischl@tum.de

Viktor Leis

Friedrich Schiller University Jena
viktor.leis@uni-jena.de

Peter Boncz

Centrum Wiskunde & Informatica
boncz@cwi.nl

Thomas Neumann

Technical University of Munich
thomas.neumann@in.tum.de

Alfons Kemper

Technical University of Munich
alfons.kemper@in.tum.de

ABSTRACT

We propose a novel method to represent compressed bitmaps. Similarly to existing bitmap compression schemes, we exploit the compression potential of bitmaps populated with consecutive identical bits, i.e., 0-runs and 1-runs. But in contrast to prior work, our approach employs a binary tree structure to represent runs of various lengths. Leaf nodes in the upper tree levels thereby represent longer runs, and vice versa. The tree-based representation results in high compression ratios and enables efficient random access, which in turn allows for the fast intersection of bitmaps. Our experimental analysis with randomly generated bitmaps shows that our approach significantly improves over state-of-the-art compression techniques when bitmaps are dense and/or only barely clustered. Further, we evaluate our approach with real-world data sets, showing that our tree-encoded bitmaps can save up to one third of the space over existing techniques.

ACM Reference Format:

Harald Lang, Alexander Beischl, Viktor Leis, Peter Boncz, Thomas Neumann, and Alfons Kemper. 2020. Tree-Encoded Bitmaps. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20)*, June 14–19, 2020, Portland, OR, USA. ACM, New York, NY, USA, 17 pages. <https://doi.org/10.1145/3318464.3380588>

1 INTRODUCTION

Bitmap indexes have a long history in database systems and information retrieval [8, 11, 16, 37, 45, 53, 57]. They

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGMOD'20, June 14–19, 2020, Portland, OR, USA
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-6735-6/20/06...\$15.00

<https://doi.org/10.1145/3318464.3380588>

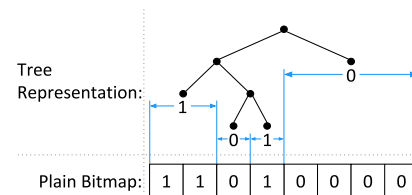


Figure 1: The key idea is to represent bitmaps as full binary trees. Longer runs are mapped to tree nodes closer to the root, and vice versa.

have many applications, such as efficiently evaluating predicates [42, 45, 46] and have been used to accelerate join [44] and aggregation [9, 46] queries. For medium or high cardinality columns, bitmap indexes consist of many individual bitmaps that are sparsely populated with 1-bits. Therefore, plain bitmaps consume large amounts of space, and compression is essential.

Consider the case of a bitmap index on an attribute A consisting of $|A|$ individual bitmaps of length n , where $|A|$ is the number of distinct values of A and n the number of tuples in the corresponding relation. The total number of 1-bits in the index is also n , whereas each bitmap receives $\frac{n}{|A|}$ 1-bits on average. A high number of distinct values, or the presence of skew, results in bitmap indexes with many sparsely populated bitmaps. Sparsity implies that these bitmaps mostly consist of consecutive 0-bits, i.e., 0-runs. Having long runs of identical bits offers great compression potential, which all existing bitmap compression schemes try to exploit.

One simple, but fairly effective bitmap compression scheme is the Word-Aligned Hybrid [63] (WAH) approach, whose compression is based on run-length encoding (RLE). A WAH-compressed bitmap is a sequence of machine words, typically 32 or 64 bits in size. Each word either encodes a run or represents a small part of the original bitmap as is. The first is called a *fill word* and the latter a *literal word*. While WAH offers significantly better performance than its predecessor the Byte-Aligned Bitmap Compression [2] (BBC), its compression effectiveness suffers from two major weaknesses: (i) runs need to be rather long for the RLE-based compression to be effective and (ii) WAH has linear space overhead (one bit

per word) for distinguishing between fill and literal words. In particular, the first weak point impairs compression when some random bits (also called *dirty bits* or *odd bits*) disrupt long runs. Over the years, several extensions to WAH have been proposed to solve this issue, i.e., PLWAH [18], Concise [15], VAL-WAH [22], EWAH [33], and SBH [27].

All the aforementioned compression techniques are based on RLE and therefore share another disadvantage, namely the linear time complexity of random access. Supporting efficient random access directly affects the efficiency of logical operations like bitwise AND, which are common operations in analytical queries.

Chambi et al. identified this problem and proposed the Roaring Bitmap format [10]. In contrast to the aforementioned compression techniques, Roaring Bitmap does not rely on RLE. Instead it partitions the input bitmap into equally sized chunks of length 2^{16} bits, where each chunk is physically stored in a separate container, as illustrated in Figure 2. Roaring implements three different container types and each container type represents the corresponding part of the bitmap differently. Depending on the number of bits set and on the presence of 1-runs, Roaring chooses the container type that consumes the smallest amount of memory. More precisely, if the number of 1-bits is less than or equal to 4096, an *array container* is used that stores a sorted list of 16-bit integers, one for each set bit. The integer values correspond to the positions of those bits within the current partition. If the number of set bits exceeds 4096, Roaring either employs a plain *bitmap container* or a *run container* [32]. A bitmap container stores the partition as is. A run container on the other hand stores the 1-runs as a list of 16-bit integer pairs $\langle a, b \rangle$, where $[a, b]$ is the range spanned by the 1-run.

Overall, Roaring is a very lightweight approach in terms of compression, as it only relies on integer arrays to represent bitmaps. Integer values are thereby truncated to 16 bits as every container encodes 2^{16} bits of the bitmap. Nevertheless, it results in significantly lower space consumption compared to RLE-based techniques in most scenarios. Due to the fact that the bit positions, the runs, and the containers themselves are sorted, a random access can be performed in logarithmic time, which significantly improves the performance of bitwise operation and thus of analytical queries [9].

It is worth mentioning that in principle Roaring is an extendable format, as it could employ any bitmap compression technique at the container level; including the tree-encoded bitmaps, we present in this work.

At the time of writing, Roaring was available in 11 programming languages and was widely used in Apache projects like Druid, Hive, Kylin, Lucence, Spark, and other systems¹. This shows that today’s applications not only demand high

¹We refer the reader to the official web site [31] for more details.

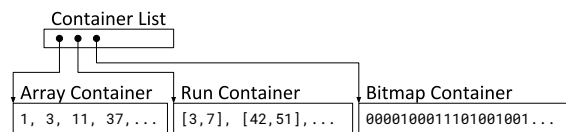


Figure 2: Roaring partitions the bitmap and stores each partition using the best suitable container type.

compression ratios but also efficient logical operations on compressed bitmaps. Further, we see a trend in database systems towards denser bitmaps—in particular, when bitmap indexes use histogram-based binning or are constructed to support range queries [11, 12]. In both cases, the resulting bitmaps exhibit higher bit densities compared to simple bitmap indexes as described at the beginning of this section².

With this work, we contribute a novel method to compress bitmaps. The compressed representation, which we call a *tree-encoded bitmap*, provides high compression ratios paired with logarithmic access time. Its primary strengths are the abilities (i) to compress both long and short runs and (ii) to significantly improve the compression ratios with denser bitmaps over existing techniques. The major conceptual difference compared to other compressed bitmap formats is that our approach employs a binary tree to represent bit runs of various lengths as illustrated in Figure 1. Tree nodes in the upper tree levels (closer to the root) thereby correspond to longer runs, and tree nodes in the lower levels to shorter runs. The low space requirement is achieved by using a succinct tree encoding and additional space optimizations that truncate balanced parts of the tree structure from the compressed representation. A key insight is that although our approach initially triples the size of a given bitmap to establish the tree structure, it does not only amortize this overhead, but also ultimately offers overall better compression ratios than RLE-based compression methods or the state-of-the-art Roaring Bitmap in a wide spectrum of moderately populated and clustered bitmaps. Using a collection of real-world data sets, we empirically found that tree-encoded bitmaps offer the best compression in 7 out of 8 cases, saving up to $1/3$ space in comparison with the second best solution.

Notation. Throughout the paper, we let n denote the length of a bitmap. Further, since compression heavily depends on the data distribution, we use the following two metrics to characterize individual bitmaps: (i) The *bit density* denoted as d refers to the fraction of bits set to 1, where $0 \leq d \leq 1$. The total number of set bits in a bitmap is therefore $d \cdot n$. (ii) The *clustering factor* denoted as f , with $1 \leq f \leq n$, indicates the degree of clustering of the 1-bits in a bitmap, i.e., how likely a 1-bit is followed by another 1-bit. Formally, it is defined as the average length of the 1-runs in a bitmap [63]. For

²In Section 5 we give a brief overview on the design space of bitmap indexes.

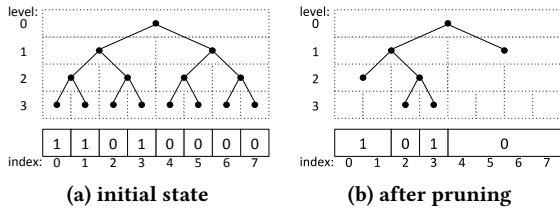


Figure 3: A bitmap represented as a binary tree. Initially, each leaf node is assigned a single bit (label). Sibling leaf nodes with identical labels are then pruned and the label is assigned to their parent. After pruning, the prior parent node becomes a leaf and represents multiple consecutive bits, a 0-run or a 1-run.

instance, the bitmap 01110010 (with $d = 0.5$) contains two 1-runs, one of length 3 and one of length 1. The clustering factor f therefore equals to 2. As both d and f refer to the set bits, they are dependent and the following restrictions apply: The clustering factor cannot exceed the total number of bits set ($f \leq d \cdot n$). Further, when the bit density exceeds 50%, the smallest possible value for f increases as well. E.g., given the bitmap 01010101 with $d = 0.5$ and $f = 1$; when the leftmost 0-bit is toggled (11010101), d increases to 0.625 and f to 1.25. In that particular case, 1.25 is the smallest possible clustering for a bitmap of length $n = 8$ and $d = 0.625$. In the general case, the smallest possible clustering is $\max(1, d/(1-d))$. Clustered bitmaps can be synthetically generated using a two-state Markov process, which we describe in the evaluation section.

2 TREE-ENCODED BITMAPS

In this section, we present our Tree-Encoded Bitmaps (TEB). The key idea behind TEB is to represent bitmaps as binary trees, which enables efficient navigation and therefore fast random access. The data structure is best explained by describing the construction algorithm. We therefore first present the tree-based compression algorithm. Later in this section, we describe how the tree is encoded space efficiently.

2.1 Compression

A TEB is constructed in two phases. In the first phase, a perfect binary tree is established on top of a given bitmap, as shown in Figure 3a. Each bit in the bitmap is associated with a single leaf node of the binary tree. Only leaf nodes carry a payload, which we refer to as *labels*. A label can either be a 0-bit or a 1-bit.

In the second construction phase, the binary tree is pruned bottom-up. Thereby, the algorithm removes all sibling leaf nodes with identical labels l , and the label l is assigned to the parent node. The pruning process stops when all pairs of sibling leaf nodes have different labels. Figure 3b depicts a fully pruned tree. The important thing to note here is that

the newly created leaf nodes in the upper tree levels no longer represent individual bits of the bitmap; instead they represent consecutive bits that form either a 0-run or a 1-run. For instance, the leftmost node in Figure 3b represents a 1-run of length 2, starting at index 0 and the rightmost node represents a 0-run of length 4, starting at index 4.

With every single pruning step, two nodes are eliminated from the tree structure and one bit from the labels. Bottom-up pruning can therefore be considered a *lossless compression* method. Compressing the tree structure is a crucial part of TEB because the space overhead of the tree structure needs to be amortized. The tree initially consists of $2n - 1$ nodes, assuming n is a power of two. When the tree structure is encoded using one bit per node, then the space consumption of a TEB, including the labels, is initially, and in worst case, $3n - 1$ bits. Even though the worst case space consumption is relatively high, we will show that our tree-based representation of bitmaps often achieves significantly lower space usage than other compression schemes.

2.2 Encoding

An important part of TEB is the space-efficient way the tree structure is stored. We employ a *level-order binary marked* representation [24], which requires one bit per tree node. The encoded tree itself therefore is a sequence of bits (a bitmap).

We have to differentiate between the tree data structure that is used during compression and the *encoded* tree that is eventually stored in a TEB. For the tree-based compression, we temporarily make use of an implicit data structure [59] that allows for fast modifications, but occupies a constant amount of space – constant in the sense that its size does not change when nodes are removed. The level-order binary marked representation, on the other hand, is static but requires less space once the tree has been pruned. Thus, *encoding* is the process with which we transform the pruned tree into a more compact form.

To encode the pruned tree structure we traverse it in breadth-first left-to-right order (or level-order) and for each visited node a single bit is emitted, a 1-bit for inner nodes and a 0-bit for leaf nodes. These bits are appended to the bit sequence that represents the encoded tree, denoted as T . The labels of the leaf nodes are stored as a separate bit sequence to which we refer as L . When a leaf node is observed during traversal, its label bit is appended to L . For instance, the tree in Figure 3b is encoded as $T = 1100100$, $L = 0101$.

To support efficient random access and bitwise operations, it is necessary to traverse the tree. Internally, the most important primitive operation is to determine the two child nodes of some given tree node, i.e., navigating downwards the tree. Within the encoded tree, each tree node is identified by its position in the bit sequence T . The sequence starts with the

root node at position 0. For any given tree node i , the child nodes can then be determined as follows [24]:

$$\begin{aligned} \text{left-child}(i) &:= \text{right-child}(i) - 1 \\ \text{right-child}(i) &:= 2 \cdot \text{rank}(i) \end{aligned}$$

where $\text{rank}(i)$ refers to the number of 1-bits (inner nodes) in T within the range $[0, i]$.

Computing the rank of a node is a linear-time operation, and navigating from the root to any leaf node is therefore an $O(n \cdot \log n)$ operation. However, the rank operation can be turned into an $O(1)$ operation at the cost of additional space consumption [24]. TEB uses an implementation similar to the one used in [68], which pre-computes the rank on 512-bit block granularity and stores the values in an auxiliary integer array; which results in a 6.25% increased memory footprint. The rank is then computed as

$$\text{rank}(i) := R[\lfloor i/512 \rfloor] + \text{popcount}(T, \lfloor i/512 \rfloor \cdot 512, i)$$

where R refers to the array with the pre-computed values at block level and popcount counts the 1-bits in the last block up to index i .

Using an additional integer array populated with pre-computed ranks (a lookup table) is a common approach [20, 21, 43, 69] and changing the granularity of the lookup table offers a space/time trade-off. The more coarse-grained the lookup table is, the lower its space requirement and the higher the costs for counting the 1-bits within the last block; and vice versa. For TEB, we empirically determined that a granularity of 512 bits offers competitive performance at a reasonable space overhead. On a reasonably modern 64-bit hardware, a navigational operation in the tree therefore requires at most eight population count instructions (four on average) and one array lookup.

Besides the downward navigation, the rank of a tree node is further required to determine the node's label. If the node i is a leaf, then the position of the label within L is equal to the number of 1-bits in T preceding node i , which corresponds to the non-inclusive rank of i . However, because only leaf nodes have labels, we can use the inclusive³ rank from above, because $T[\text{rank}(i)]$ is guaranteed to be a 0-bit. In summary, a label is accessed as follows:

$$\text{label}(i) := L[i - \text{rank}(i)]$$

Let us close by mentioning that the chosen encoding requires the tree structure to be a full binary tree, i.e., each node has either zero or two child nodes. It is easy to show that this holds for the tree structure of a TEB: Since the initial binary tree is perfect, and pruning always affects two sibling leaf nodes, the resulting tree structure remains full binary.

³We chose the inclusive rank as it results in fewer arithmetic instructions.

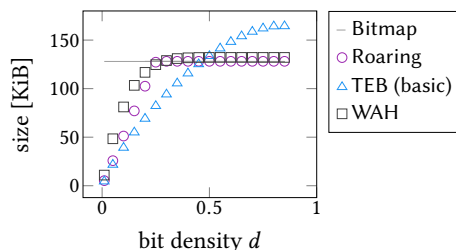


Figure 4: Size comparison for varying bit densities and a fixed clustering factor of 8.

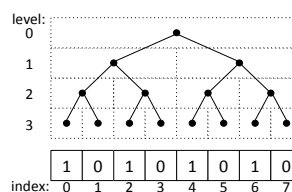


Figure 5: In worst case, the tree cannot be pruned (compressed) and the resulting TEB consumes approximately three times the space of the original bitmap.

2.3 Optimizations

The basic idea of TEB we have presented so far already shows promising results with regard to compression ratios. For instance, Figure 4 shows a space comparison of the TEB approach with two state-of-the-art bitmap compression techniques, Roaring and WAH. The compressed size (y-axis) depends on the ratio of 1-bits in the original bitmap (x-axis). Sparsely populated bitmaps offer higher compression potentials than densely populated bitmaps. In that particular case, if more than $\sim 25\%$ of the bitmap is populated with 1-bits, Roaring and WAH do not offer any compression at all. Both fall back to an uncompressed (literal) representation. TEB, on the other hand, is able to compress bitmaps with a bit density of up to $\sim 45\%$.

The downside of the basic TEB approach is that in corner cases it can significantly exceed the size of the plain bitmap. In contrast to Roaring and WAH, our approach does not support an alternative representation to which it could fall back. In the following, we show that it is in fact not necessary to switch between different representations to address the high space consumption of TEB in unfavorable cases. It just requires a few minor modifications to the data structure and the compression algorithm, which we discuss in the following.

Implicit Tree Nodes. We motivate our first space optimization by considering the worst-case scenario for TEB. Figure 5 illustrates such a case. The depicted alternating bit sequence does not offer any compression potential. All pairs of sibling leaf nodes have different labels and therefore bottom-up

pruning cannot remove any tree nodes. The resulting TEB would consist of $n-1$ 1-bits for the inner nodes, followed by n 0-bits for the leaf nodes, and n label bits. In this extreme case, the label bits in L are identical to the uncompressed bitmap. Thus, storing the encoded tree structure is pure overhead.

Our first space optimization is to omit the leading 1-bits as well as the trailing 0-bits of the encoded tree structure. Only the intermediate bits of the tree structure are stored in the physical representation of a TEB. We refer to the omitted nodes as *implicit* tree nodes, and to the remaining as *explicit* tree nodes.

With regard to the worst case, this simple modification allows for the elimination of the entire tree encoding from the physical representation. Only the n label bits remain:

$$T = \underbrace{1111111}_{\text{leading 1-bits}} \underbrace{00000000}_{\text{trailing 0-bits}}, \quad L = 10101010$$

As mentioned before, the labels in L are identical to the original bitmap, i.e., the TEB degraded into an uncompressed bitmap. Thus, the size of the TEB is equal to the size of the plain bitmap, except for a small overhead that is caused by metadata.

However, further optimizations are needed, as this minor modification only mitigates the high space consumption of TEBs when the plain bitmap is poorly compressible. The TEB size may still significantly exceed the size of the uncompressed bitmap, i.e., the worst case has shifted. The modification, however, has two important implications:

- (i) The encoded tree structure T is an optional part of the physical TEB data structure, as the entire tree may be implicit.
- (ii) The space minimal TEB instance does not necessarily contain a fully pruned tree.

We give an example for (ii) in Figure 6a. The depicted TEB consists of three explicit tree nodes and four labels. Thus the space requirement is $3 \cdot 1.0625 + 4 = 7.1875$ bits, where the factor 1.0625 is to incorporate the space consumption of the rank helper structure (cf. Section 2.2). Figure 6b shows the TEB instance with the minimum size. The difference between the two TEB instances is that in Figure 6a the tree is fully pruned, whereas in 6b the two sibling leaves in the high-lighted subtree have been preserved. The second instance therefore comprises a larger tree, but even though the total number of tree nodes and labels are higher, the second instance occupies less space ($2 \cdot 1.0625 + 5 = 7.125$ bits), as fewer tree nodes need to be stored explicitly. The circumstance that a fully pruned tree, in general, no longer corresponds to the smallest TEB instance requires a modification to the bottom-up pruning algorithm: Instead of returning the fully pruned tree, the algorithm needs to return the smallest tree instance observed during pruning, where the size is computed based

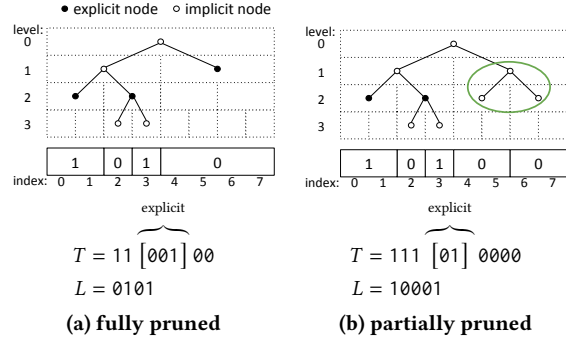


Figure 6: Two different tree representations of the bitmap 11010000. The fully pruned tree (a) occupies more space than the partially pruned tree (b), as more tree nodes need to be stored explicitly.

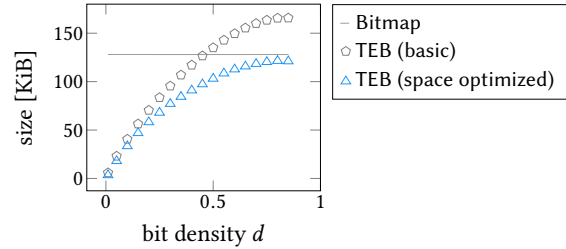


Figure 7: Size comparison of basic and space optimized TEBs using a clustering factor of 8.

on the number of explicit nodes, rather than the total number of nodes.

Implicit Labels. Our second modification is to omit leading and trailing 0-labels in the physical TEB representation, similarly to implicit tree nodes. Omitting the leading 0-labels reduces the space consumption in particular with very sparse bitmaps. The tree representation of a sparse bitmap typically consists of a few leaf nodes with 1-labels at the deepest tree level $\log_2(n)$. But most of the leaf nodes with 0-labels can be found in the tree levels 1 to $\log_2(n) - 1$. Due to the tree being encoded in level order, the label bit sequence L tends to start with a long run of 0-labels, which we do not need to store explicitly. Trailing 0-labels on the other hand can occur when the length of the input bitmap is not a power of two. In that case, a TEB internally rounds up to the next power of two and fills the range $[n, 2^{\lceil \log_2(n) \rceil})$ with 0-bits. Omitting these trailing 0-bits ensures that the number of stored labels never exceeds the length of the original bitmap.

The presented modifications reduce the overall space usage, as shown in Figure 7. In particular, the worst-case space consumption reduced from $3n-1$ to n bits, excluding the

(small) metadata. We observe that in an optimized TEB the fraction of space occupied by the tree, the rank helper structure, and the labels is no longer fixed; compare Figures 8a and 8b. With sparse bitmaps, the labels occupy significantly less space. With denser bitmaps, on the other hand, we see that the fraction of space occupied by the tree structure decreases. Figure 9 shows how the implicit tree nodes and the implicit labels optimizations contribute to the space savings. The implicit labels optimization is most effective with sparse bitmaps and the implicit tree node optimization, on the other hand, favors denser bitmaps.

An important implication is that the space optimizations balance the upper part of the tree structure, as the example in Figure 6 has shown. The partially pruned tree in Figure 6b is perfectly balanced until level two, whereas the fully pruned tree in Figure 6a is only perfectly balanced until level one. Thus in general, the tree can be split into an upper balanced and a lower imbalanced part. This property allows for the reduction of the cost of navigational operations. We exploit the fact that within a perfect binary tree we can directly address the individual tree nodes, i.e., without computing ranks. If the number of the upper *perfect levels* is known, these levels of the tree can be logically cut off, and only the remaining sub-trees need to be considered. In our case, we can directly compute the number of perfect levels u based on the number of implicit inner nodes c that are already known when the space optimizations have been applied: $u := \lfloor \log_2(c+1) \rfloor + 1$. The corresponding node IDs for the last perfect level are within the range $[t_{\text{begin}}, t_{\text{end}})$, with $t_{\text{begin}} := 2^{u-1} - 1$ and $t_{\text{end}} := 2^u - 1$. Each of these nodes, or the sub-trees rooted at these nodes, respectively, span a range of length $2^{\log_2(n)-u-1}$ in the original bitmap. Thus, it can be considered as a uniform partitioning scheme, similar to the one used in Roaring Bitmaps, but with the major difference that the partition size is chosen adaptively.

The number of perfect tree levels is correlated with the effectiveness of the tree-based compression. The less effective the compression, the larger the number of perfect levels, and vice versa. In worst case, the entire tree is implicit and the number of perfect levels corresponds to the tree height. In other words, TEBs gradually degrade into literal bitmaps, but unlike Roaring and WAH, TEBs remain homogeneous and do not need to switch between different encodings or representations.

3 OPERATIONS

In this section, we describe the operations supported by TEB. Fundamentally, a TEB supports two access methods: (i) a point lookup and (ii) a 1-run iterator. High-level functionalities, like decompressing a bitmap or logical operations are implemented on top of the 1-run iterator.

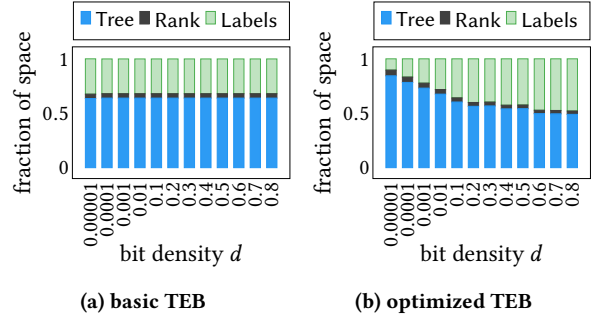


Figure 8: The fraction of space occupied by the tree, the rank helper structure, and the labels.

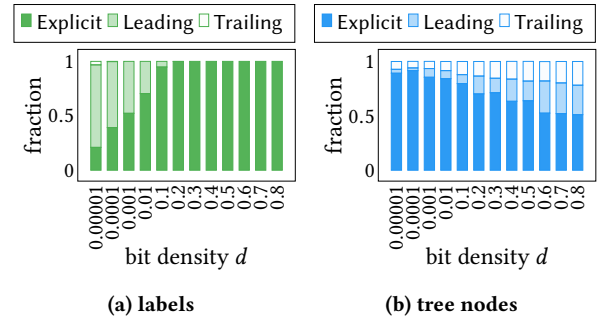


Figure 9: The fraction of explicitly stored labels (a) and tree nodes (b).

Algorithm 1: Point lookup

Input : The bit index k to test
Returns : true if the k^{th} bit is set, false otherwise

// Determine the tree node at the last perfect level.
 $t_{\text{offset}} \leftarrow k \gg (\text{tree_height} - \text{perfect_levels} - 1)$
 $i \leftarrow t_{\text{begin}} + t_{\text{offset}}$
 $j \leftarrow \text{tree_height} - 1 - \text{perfect_levels} - 1$
// Navigate downwards until a leaf node is observed.
while i is an inner node **do**
 $\text{direction} \leftarrow \text{extract } j^{\text{th}} \text{ bit from } k$
 $i \leftarrow \text{left-child}(i) + \text{direction}$
 $j \leftarrow j - 1$
end while
return label(i)

3.1 Point Lookup

A point lookup is a straightforward operation that navigates downward the tree until a leaf node is reached. The index k of the bit to look up thereby specifies the path to take within the tree. For performance reasons, the downward navigation starts at the last perfect tree level rather than at the root node. The details are shown in Algorithm 1.

3.2 Run Iterator

The iterator interface allows for efficient iteration over a TEB. Unlike the iterators implemented in Roaring and WAH, the TEB iterator does not iterate over the individual 1-bits, instead it iterates over the 1-runs of a bitmap. A 1-run is thereby represented as two integer values $\langle begin, end \rangle$, pointing to the position of the first 1-bit and to the position one past the last 1-bit. The iterator traverses the tree in depth-first left-to-right order. To navigate down the tree, the functions `left-child()` and `right-child()` are used, as described in Section 2.2. To navigate upwards, the iterator makes use of a small stack that is populated during downward navigation. Other data structures like SuRF [68] implement upwards navigation using the `select` primitive, the counterpart to rank. For TEB, we prefer a classic stack-based approach as it is significantly faster in practice and saves space.

During tree traversal, the iterator needs to keep track of its position (and level) within the tree structure. This information is required to determine the start index and length of a 1-run when the iterator reaches a leaf node with label 1 and thus needs to produce an output. The iterator therefore maintains a *path* variable that encodes the path from the root to the current node using a single integer. The initial (and minimum) value of the path variable p is 1. During downwards navigation, a 0-bit is shifted in when navigating to the left child $p := (p \ll 1)$ and a 1-bit when navigating to the right child $p := (p \ll 1) | 1$. The index of the most significant 1-bit (the *sentinel bit*) indicates the level of the corresponding tree node:

$$\text{level}(p) := \text{sizeof}(p) \cdot 8 - 1 - \text{lzcount}(p)$$

where `sizeof(p)` refers to the size of the variable p in bytes and `lzcount(p)` to the number of leading zeros in p . A tree node that is identified by its path p then represents a run that starts at position

$$\text{pos}(p) := (p \oplus (1 \ll \text{level}(p))) \ll (tree_height - \text{level}(p))$$

with length

$$\text{length}(p) := n \gg \text{level}(p) \hat{=} 2^{\log_2(n) - \text{level}(p)}.$$

Similarly to the point lookup access method, the upper perfect levels of the tree are skipped. The iterator only considers the sub-trees rooted in $[t_{\text{begin}}, t_{\text{end}})$, as described in Section 2.3. Algorithm 2 shows how the iterator is forwarded to the next 1-run.

As mentioned earlier, a time-critical operation is to *fast-forward* the iterator to a desired position, thereby skipping all set bits in between. Thanks to the navigable tree structure, the operation can be performed in logarithmic time. Nevertheless, to achieve competitive performance in practice,

Algorithm 2: Forward the iterator to the next 1-run.

```

while  $t < t_{\text{end}}$  do
  while stack is not empty do
    // Pop tree node  $i$  and its path  $p$  from the stack.
     $\langle i, p \rangle \leftarrow \text{stack.pop}()$ 
    while  $i$  is an inner node do
      // Push right child on stack and go to left child.
       $i \leftarrow \text{left-child}(i)$ 
       $p \leftarrow p \ll 1$ 
      stack.push( $\langle i + 1, p | 1 \rangle$ )
    end
    // Reached a leaf node.
    if  $\text{label}(i) = 0$  then continue
    // Found a 1-run. Update the iterator state.
     $\text{level} \leftarrow \text{sizeof}(p) \cdot 8 - 1 - \text{lzcount}(p)$ 
     $\text{begin} \leftarrow (p \oplus (1 \ll \text{level})) \ll (tree\_height - \text{level})$ 
     $\text{end} \leftarrow \text{begin} + (n \gg \text{level})$ 
    return
  end
   $t \leftarrow t + 1$ 
   $p \leftarrow (t - t_{\text{begin}}) | (1 \ll (\text{perfect\_levels} - 1))$ 
  stack.push( $\langle t, p \rangle$ )
end
 $\text{begin} \leftarrow \text{end} \leftarrow n$  // Reached the end.
return

```

we optimize the skip operation so that unnecessary navigation steps are avoided. The primary decision that is to be made is whether to (i) navigate up the tree to the common ancestor of the current and the destination node, and then downwards in the right sub-tree to the desired position, or (ii) start at the root node (or at the corresponding tree node in the last perfect level) and navigate only downwards until the desired position has been reached. Depending on the source and destination nodes, one option might be more efficient than the other. The two options may differ in the number of required navigation steps. But we also need to consider that navigating upwards is less costly in terms of issued CPU instructions than navigating downwards. The asymmetrical costs are mostly caused by the rank primitive, which is significantly more costly than accessing the stack. We experimentally determined that a downward step is approximately $9 \times$ more expensive than an upward step (~ 55 cycles vs. ~ 6 cycles).

Our decision logic works as follows: We start with a fast test to determine whether the destination position is outside of the current sub-tree:

$$pos \gg (h - u - 1) \neq to_pos \gg (h - u - 1)$$

where h refers to the tree height and u to the number of perfect levels. If the expression evaluates to true, we can directly go to the corresponding node at the last perfect level and navigate downwards until the desired position has been reached. Otherwise, if the destination node is within

the current sub-tree, we (i) determine the common ancestor node (ii), estimate the navigational costs for both options, and (iii) pick the cheaper path.

It is worth mentioning that an iterator with skip support is not the most efficient way to decompress (rather than intersect) a TEB. For these cases we provide an alternative iterator to which we refer to as *scan iterator*. Unlike the regular iterator, the scan iterator’s seek function operates in $O(n)$, but it offers a significantly higher read throughput, as it (i) decodes the tree in batches and (ii) does not rely on the rank primitive to traverse the tree.

3.3 Tree Scan

In this section, we present a tree traversal algorithm that is optimized for modern x86 hardware. The algorithm takes a level-order encoded binary tree and iterates over all leaf nodes in left-to-right order. We refer to the algorithm as *tree scan*. The tree scan is the basic building block for the TEB scan iterator.

Generally speaking, navigating from one leaf node to the next one is a 3-step process: (i) navigate up the tree until a left child is observed, (ii) go to its right sibling, and (iii) walk down the tree to the leftmost leaf node. The key idea behind our solution is to have multiple lightweight bit iterators for the encoded tree structure T , one iterator per tree level, and then scan the bit sequence T in parallel. We denote the bit iterators in T as t_l with $0 \leq l < h$. Initially, all iterators point to the first bit in T at their corresponding level l . We expose the values each iterator points to as an integer value, denoted as α . The bits in $\alpha := b_{h-1} \dots b_1 b_0$ are populated with $b_l = *t_l$, where $*$ denotes the dereference operator. A path variable p identifies the position and the level within the tree, as described earlier. Initially, p points to the leftmost leaf node. Using the two values α and p , we can efficiently iterate over all leaf nodes in left-to-right order, cf., Algorithm 3. Thereby, p determines the number of upward steps and α determines the number of downward steps to perform in each iteration.

The bit iterators are implemented using the AVX-512 SIMD instruction set as follows. We use a 512-bit SIMD register to buffer the tree structure. The register is interpreted as 32×16 -bit integers, i.e., the register is split into 32 lanes. Thereby, each SIMD lane corresponds to a tree level. For each level we load up to 16 bits from the encoded tree T . For instance, Figure 10 illustrates a buffer that contains the tree from Figure 6b. To consume the buffered tree bit by bit, we use a second SIMD register to which we refer as *read mask*. The read mask again consists of 32 lanes, and a single bit is set within each lane. Initially, the least significant bit is set to 1. The position of that bit represents the current read position in the corresponding buffer lane. Thus, the read mask represents the state of all (up to) 32 lightweight bit

Algorithm 3: Tree scan

```

p // The current path. Initially points to the leftmost leaf.
do
  // Produce an output, if the label of the current node is 1.
  ...
  // Walk upwards until a left child is found.
  up_steps ← tzcount(~p)
  last ← level(p) + 1
  p ← p >> up_steps
  p ← p | 1 // Go to the right sibling.
  first ← level(p)
  increment the iterators  $t_{first}$  to  $t_{last}$  and update  $\alpha$ 
  // Walk downwards to the leftmost leaf in that sub-tree.
  down_steps ← tzcount(~( $\alpha$  >> level(p)))
  p ← p << down_steps
while not done

```

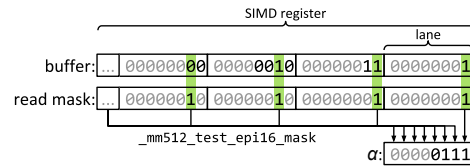


Figure 10: AVX-512 allows for the instantiation of up to 32 lightweight bit iterators (one for each tree level) using only two SIMD registers: The first is used to buffer the encoded tree level by level and the second represents the iterators’ read positions.

iterators. The increment of an iterator is then implemented as a left shift of the corresponding lane. The implementation has the advantages that we can work with multiple iterators in parallel and that the most important operations can be performed in a single instruction. For instance, multiple iterators can be incremented using a single masked shift instruction (`_mm512_mask_slli_epi164`) and all iterators can be dereferenced in parallel to retrieve the aforementioned α value; cf., Figure 10.

The presented algorithm is used in the TEB scan iterator that is supposed to be used when efficient skip support is not required, e.g., when decompressing an entire TEB. With regard to performance, the scan iterator benefits from the predictable memory access pattern, as well as from the reduced number of memory loads, due to buffering. However, a problem not mentioned above is that we need to know the start offset in T for each tree level. Unfortunately, determining these offsets is a linear time operation. Therefore, we store the offsets as part of the TEB metadata, which now is logarithmic in size. For brevity we have omitted some of the

⁴We refer the reader to the Intel Intrinsics Guide for more details on the SIMD instruction set architectures: <https://software.intel.com/sites/landingpage/IntrinsicsGuide>

Algorithm 4: Next function of the AND iterator.

```

Input: Run iterators  $a$  and  $b$ .
while  $!(a.begin != n \parallel b.begin != n)$  do
   $begin\_max \leftarrow \max(a.begin, b.begin)$ 
   $end\_min \leftarrow \min(a.end, b.end)$ 
   $overlap \leftarrow begin\_max < end\_min$ 
  if  $overlap$  then
    if  $a.end \leq b.end$  then  $a.next()$ 
    if  $b.end \leq a.end$  then  $b.next()$ 
     $begin \leftarrow begin\_max$  // Update the iterator state.
     $end \leftarrow end\_min$ 
    return
  else
    if  $a.end \leq b.end$  then  $a.skip\_to(b.begin)$ 
    else  $b.skip\_to(a.begin)$ 
  end
end
 $begin \leftarrow end \leftarrow n$  // Reached the end.

```

implementation details, such as how buffers are refilled and how labels are buffered and accessed; which works similarly to the buffering of the tree structure. We invite the interested reader to examine the source code of TEB⁵.

3.4 Logical Operations

As mentioned earlier, high-level functionality is implemented on top of the 1-run iterator. Operations like a bitwise AND are themselves implemented as iterators and can therefore be arbitrarily chained and combined to evaluate complex expressions. Algorithm 4, for instance, shows how two bitmaps are intersected using the iterator API. In contrast to the implementations in Roaring and WAH, the iterator approach does not produce a compressed bitmap. We think this is not a disadvantage because producing compressed intermediate results when evaluating complex compressions could harm performance. For instance, when bitmap indexes are used to evaluate multi-dimensional selection predicates, it is sufficient to identify the ranges (or pages) that contain qualifying tuples; an intermediate bitmap would be discarded afterwards anyhow.

3.5 Updates

Data structure design in general is a trade-off between read, update, and memory overheads. The RUM conjecture [3] states that when optimizing (reducing) two of these overheads, it impairs the third one. TEBs are optimized for efficient read access and low memory consumption, and similarly to existing RLE-based compression schemes, the static nature of TEBs does not allow for in-place updates. In the following, we discuss various approaches that can be combined with TEBs to achieve updatability.

⁵TEB source code: <https://db.in.tum.de/research/publications/#teb>

	WAH	EWAH	Concise	Roaring	TEB
CENSUS INCOME	3.4	3.3	2.9	2.6	2.1
CENSUS INCOME (sorted)	0.66	0.64	0.55	0.6	0.36
CENSUS 1881	34.4	33.8	25.6	15.1	12.6
CENSUS 1881 (sorted)	3.0	2.9	2.5	2.1	1.5
WEATHER	6.8	6.7	5.9	5.4	4.2
WEATHER (sorted)	0.55	0.54	0.43	0.34	0.26
WIKILEAKS	11.1	10.9	10.2	5.9	5.4
WIKILEAKS (sorted)	2.9	2.7	2.2	1.7	1.7

Table 1: Space usage in bits per attribute value.

The naïve and costly way to support random updates is to decompress the bitmap, perform the update on the uncompressed representation, and (re-)compress it again afterwards. Prior work [4] proposed to reduce the update costs by staging updates in an auxiliary differential data structure and to apply these pending updates in batches, rather than one-by-one. Thereby, another compressed bitmap is used as a differential data structure. While this approach greatly reduces the number of decompression/compression cycles, it also causes redundancies (slightly higher memory consumption) and requires the differential data structure to be consulted (XORed) during read access.

Roaring bitmap applies a different strategy. Due to the fixed size partitioning, an update affects only a single container, rather than the entire bitmap. Thus, in worst-case, 2^{16} bits need to be re-compressed during updates. Updates can therefore be performed in constant time⁶, even though the constant is quite large. Nevertheless, the partition size has been chosen sufficiently small to fit in an L1 cache to enable efficient decompression/compression cycles.

Both approaches can be used with TEBs. Partitioning could further be combined with differential updates so that a separate diff is maintained per partition. We will show in the later evaluation section that the combined approach offers the highest throughput regarding updates, with minor compromises regarding reads.

4 EXPERIMENTAL ANALYSIS

In the following, we evaluate our approach with regard to its compression ratio and performance. We begin by using a number of real-world data sets before performing a detailed evaluation using synthetic data.

4.1 Real-World Data

We evaluate TEBs with bitmaps from bitmap indexes constructed from four real-world data sets that have been previously used in the experimental evaluation of Roaring Bitmaps [32]. The data sets, namely CENSUS INCOME, CENSUS 1881,

⁶Assuming the corresponding containers reside in heap memory. Modifications to the serialized format would still be in linear time.

WEATHER, and WIKILEAKS, come in two flavors: *as is* and *sorted*. The latter relies on a-priori sorting of the raw input data, which leads to significantly better compression ratios [33, 34, 47]. Following the prior work, we compress the individual bitmaps, 200 per data set, and report the average number of bits per attribute value. We compare TEB with Concise, EWAH, Roaring, and WAH. The results for Concise and EWAH are taken from [32]. We reproduced the results for Roaring with very minor differences with the sorted CENSUS 1881 and WIKILEAKS data. But we observed a higher discrepancy for WAH. Among our experiments, we observed a slightly higher space usage than reported earlier, except for CENSUS 1881 where we observed a significantly better compression ratio (34.4 vs. 43.8 bits per element). We attribute these discrepancies to the fact that we use a different implementation [60] of WAH. Please note that EWAH and WAH use 32-bit words; we omit the results for the 64-bit implementations, as those have a higher space consumption among all tested workloads.

Table 1 summarizes the experimental results. TEB offers the best compression ratios, except for the sorted WIKILEAKS data, where Roaring is slightly better (1.667 vs. 1.677 bits per element). TEB saves up to 22% space on unsorted data and up to 34.6% on sorted data compared to the second best compression technique, which in most cases is Roaring.

The rank lookup table (LuT) thereby accounts for 2.2% to 4.4% of the TEB size (3.7% geo. mean, among all real-world data sets). As mentioned earlier, changing the resolution of the LuT offers a space/time trade-off. A fine-grained LuT with one entry per 64 bit offers the best performance. We observe a 30% lower execution time for computing bitmap intersections. The memory overhead of the LuT thereby increases significantly to up to 27%, which almost cancels out the improvements in compression. Decreasing the LuT resolution to 2048 bits on the other hand reduces the TEB size by up to 2.8% but also causes the intersection time to increase by up to 10%. Table 2 shows how the space consumption of TEBs changes for varying rank resolutions compared to Roaring. Throughout our experiments, we found that a 512-bit resolution offers a reasonable space/time trade-off, which we use as our default setting in the following. Nevertheless, it is noteworthy that the rank LuT could be omitted when TEBs are written to persistent storage, and could be recomputed on-the-fly when TEBs are loaded back into main memory, allowing one to save additional disk space and I/O (cf., rightmost column in Table 2).

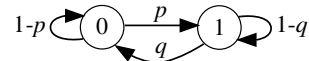
4.2 Synthetic Data

For an in-depth analysis we generate random bitmaps, where the individual 1-bits are either uniformly distributed or clustered. *Uniform* random bitmaps are random bitmaps where each bit is independently generated following an identical

	Rank LuT resolution [bits]					no
	64	128	256	512	2048	LuT
CENSUS 1881	1.10	0.95	0.87	0.83	0.81	0.80
CENSUS 1881 (sorted)	0.87	0.76	0.71	0.69	0.67	0.66
CENSUS INC.	0.93	0.86	0.82	0.81	0.79	0.79
CENSUS INC. (sorted)	0.76	0.66	0.62	0.60	0.58	0.58
WEATHER	0.93	0.84	0.80	0.77	0.76	0.75
WEATHER (sorted)	0.97	0.84	0.79	0.76	0.74	0.73
WIKILEAKS	1.18	1.02	0.95	0.91	0.89	0.88
WIKILEAKS (sorted)	1.25	1.11	1.04	1.01	0.98	0.98

Table 2: Relative size of TEB compared to Roaring ($\frac{\text{TEB size}}{\text{Roaring size}}$) for varying rank resolutions.

probability distribution [63], i.e. each bit is set with probability d . *Clustered* random bitmaps on the other hand are generated using a two-state Markov process



with the transition probabilities p and q set to

$$p := \frac{d}{(1-d) \cdot f}, \text{ and } q := \frac{1}{f}$$

with $0 < d < 1$ and $1 \leq f \leq n$. We make a minor change over the definition given in [63]; which is that we choose the initial state randomly with a probability of 0.5, whereas in [63] the initial state is ①, meaning that a randomly generated bitmap would always start with a 1-run.

We generate bitmaps of length $n = 2^{20}$ and report the averaged results over 10 independent experiments. We compare TEB with WAH [63], which is the most popular RLE-based bitmap compression scheme, and with Roaring Bitmap [32], which is the state-of-the-art with regard to performance and compression ratio. The thorough study of Wang et al. [57] found Roaring to be superior over other bitmap compression techniques such as Concise [15], WAH, EWAH [33], VALWAH [22], PLWAH [18], and SBH [27]. We therefore limit our evaluation to Roaring and WAH.

For the experiments we use FastBit [60, 61] v2.0.3, which provides a C++ implementation of WAH, and CRoaring [5] v0.2.60 (unity build), the official C/C++ implementation of Roaring Bitmap. The `dynamic_bitset` from the Boost C++ libraries [7] v1.67.0 is used for uncompressed bitmaps. We compile with GCC v8.3.0 (`-O3 -march=native`) and execute on an Intel Core i9-7900X CPU @ 4 GHz.

4.2.1 Compression.

Uniform Bitmaps. In the following, we examine the compression ratios with uniform random bitmaps with varying bit densities. The results in Figure 11 show that TEB and Roaring are on par in the case of sparse bitmaps ($d < 0.005$). With an increasing d , TEB shows the lowest space usage.

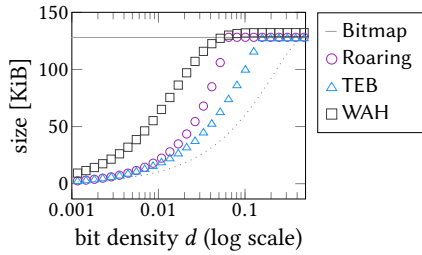


Figure 11: Size of uniform random bitmaps with varying bit densities. The dotted line refers to the information theoretic minimum.

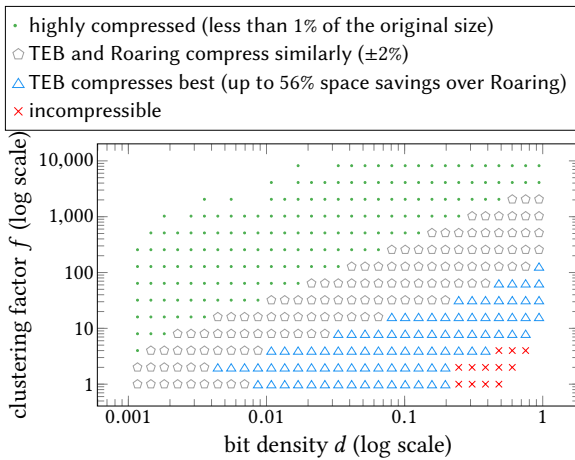


Figure 12: Summary of our findings when compressing clustered bitmaps.

When more than 13% of the bitmap is populated, TEB is no longer able to compress; Roaring and WAH already stop at 5%. With dense bitmaps ($0.5 < d \leq 1$) we observed symmetrical results for TEB and WAH, only Roaring requires a density of more than 97% for the compression to work again (rather than 95%). This is attributed to the different containers being used in Roaring, and the fact that Roaring encodes 0-runs and 1-runs differently, which is in contrast to TEB and WAH.

Clustered Bitmaps. With our third experiment, we examine the compression ratios with clustered bitmaps, using varying bit densities d and clustering factors f . We start with an exploration of the space spanned by d and f . Thereby, we consider the ranges $0.0001 \leq d < 1$ and $1 \leq f \leq n$. We make the following observations:

- When the input bitmaps are very sparsely populated or exhibit a strong clustering, all bitmap compression techniques under test perform well. In the dotted area (\cdot) in Figure 12, the compressed bitmaps occupy less than 1% of

the space of the uncompressed bitmap, irrespective from the employed compression scheme.

- TEB offers better compression ratios than WAH throughout all measurements; and only in some rare cases does WAH compress slightly better than Roaring.
- When comparing TEB and Roaring, TEB does not always offer the best compression ratios. However, in these cases, the differences in size are marginal. The largest difference in size we observed throughout all experiments is 1.6% of the original bitmap size. In the area marked with \circ in Figure 12, TEB and Roaring perform similarly.
- TEB in contrast, shows significantly higher compression ratios with denser bitmaps and bitmaps with lower clustering, cf. the area marked with \triangle in Figure 12. In comparison to Roaring, we observed a difference in size of up to 56% of the plain bitmap size, in favor of TEB. Figure 13 shows a qualitative side-by-side comparison.

Figure 14 gives a detailed view on how the size of the compressed bitmaps change for varying d and fixed f . Figure 14a shows that the TEB approach is able to exploit short 1-runs in sparse bitmaps, resulting in up to $\sim 50\%$ space savings over Roaring. With a moderate clustering, as shown in Figure 14b, our approach is also able to compress dense bitmaps. Figure 14c, on the other hand, reveals that our approach has a slightly higher space usage than Roaring with strongly clustered bitmaps, which implies that Roaring can encode longer runs more space efficiently.

Figure 15 illustrates how f affects the compression ratios. Figures 15a and 15b show that already a slight clustering can lead to significant space savings with TEB. Roaring requires a significantly higher clustering to be competitive. With sparser bitmaps, TEB falls slightly behind Roaring (see Figure 15c), whereas WAH cannot compete.

4.2.2 Performance.

In the following, we evaluate the read and update performance of TEB, and show how it compares to Roaring and WAH.

Read Access. We first investigate the read (or decompression) throughput. We thereby iterate over all 1-runs of a bitmap and measure the duration in wall-clock time. In our initial performance experiment, we again explore the space spanned by d and f . Thereby we observe that an uncompressed bitmap performs better than the compressed formats when $16 \leq f \leq 128$ and $0.01 \leq d < 1$. It should be noted that the `dynamic_bitset` implementation, which we use for uncompressed bitmaps, is very straightforward and does not include any hardware specific optimizations. Thus, we expect a performance-optimized implementation to dominate an even larger space. When we consider only the performance of compressed bitmaps, we observe that the

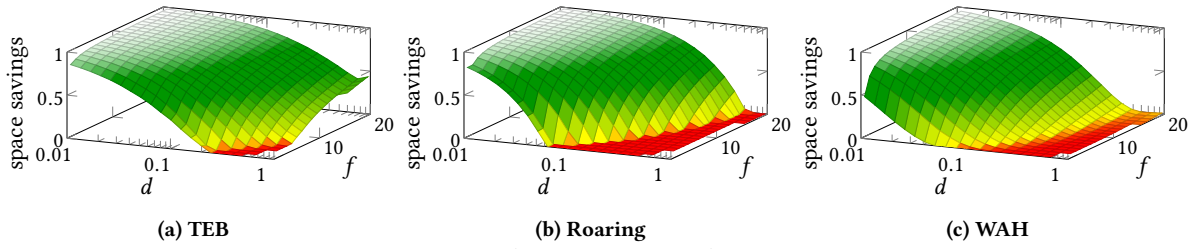


Figure 13: Space savings $\left(1 - \frac{\text{compressed size}}{\text{uncompressed size}}\right)$ for varying d and f .

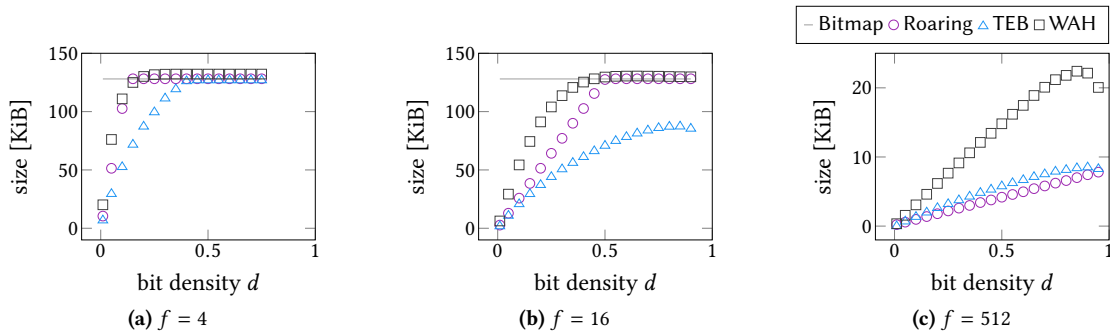


Figure 14: Compressed bitmap size for varying bit densities and fixed clustering factors.

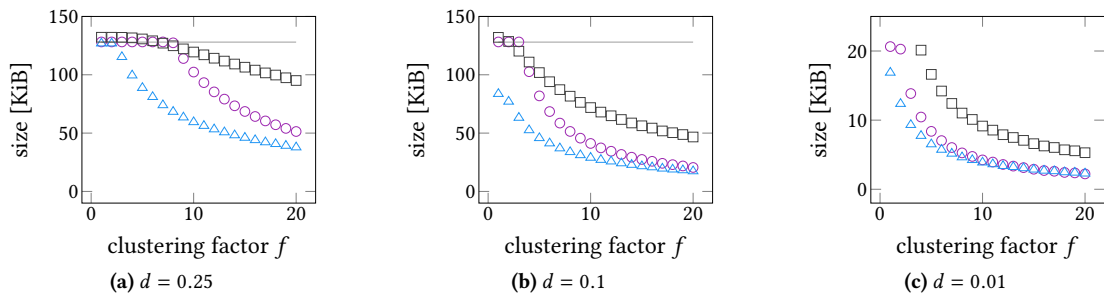


Figure 15: Compressed bitmap size for varying clustering factors and fixed bit densities.

clustering mostly determines the best performing compression technique: Roaring is dominant when $f \leq 16$, followed by WAH until f is approximately 128. TEB requires an evenly higher clustering ($f > 128$) to outperform Roaring and WAH.

In Figure 16, we compare the performance for reasonable values of d and f , which we expect to occur in practice. We fixed d to $\{0.25, 0.1, 0.01\}$ and varied f within the range $[1, 20]$. We observe that the time to read the bitmap decreases with an increasing f , which is due to the smaller size of the input and due to less branching; the higher f is, the lower the number of 1-runs to iterate over. Figure 16a, with d set to 0.25, shows that TEB offers a similar performance as WAH, and that both are close to the performance of Roaring. Still, a plain bitmap performs best in most cases. The outliers at

$f = 1$ and $f = 2$ are due to specialized code paths that are taken when the bitmaps are not compressed (or just barely compressed). In the Figures 16b and 16c, with bit densities reduced to 0.1 and 0.01, we observe that the absolute time to read a bitmap decreases for all implementations under test (note the different y-axis scales), but also that TEB falls behind relative to Roaring and WAH, indicating that the average cost per 1-run increases with lower d . Naturally, this is an expected result, as lower bit densities result in sparse and imbalanced trees, which in turn increases the number of tree levels that need to be traversed (cf., Section 2).

In our second experiment, we evaluate the effectiveness of efficient tree navigations within logical operations. We intersect (bitwise AND) two bitmaps with different characteristics.

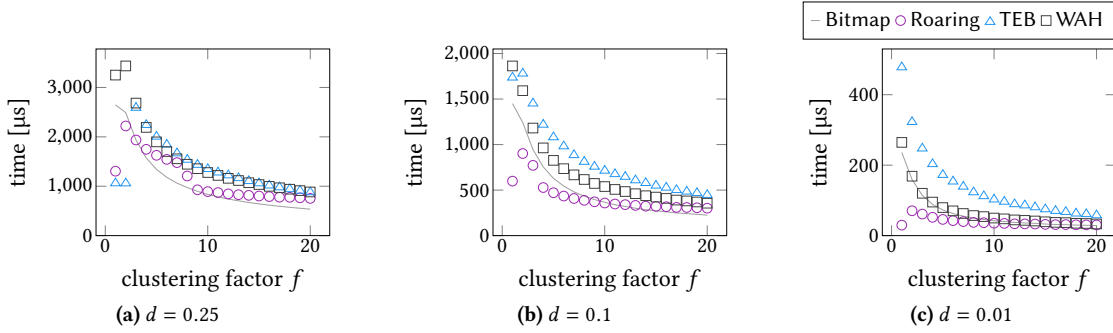


Figure 16: Read performance for varying clustering factors and fixed bit densities.

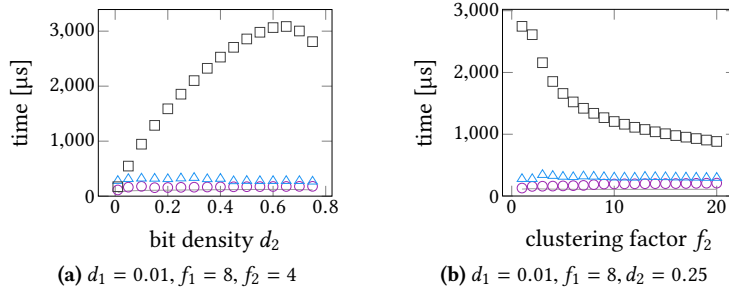


Figure 17: Intersection performance.

The density and the clustering in the first bitmap is thereby fixed to $d_1 = 0.01$ and $f_1 = 8$. In Figure 17a, we fix the clustering in the second bitmap to $f_2 = 4$ and vary the density d_2 . We observe that the density of the second bitmap only has a minor impact on the overall intersection time, except for WAH. The intersection of uncompressed bitmaps, with constant time random access, is fastest in this setting. Roaring takes $\sim 1.5\times$ the time of the plain bitmap intersection, and TEB $\sim 1.9\times$ the time of Roaring. In Figure 17b, we fix the density of the second bitmap to $d_2 = 0.25$ and vary f_2 . Again, only WAH is sensitive to the varying clustering factor and thus to the size of the second bitmap. On average, Roaring needs $\sim 1.8\times$ the time of the plain bitmap intersection, and TEB $\sim 1.6\times$ the time of Roaring.

Differential Updates. In our final experiments, we extend TEB and the other bitmap compression techniques under test by a differential data structure and evaluate the update performance. Our experiments revealed that WAH is not well suited as a differential data structure. We found that Roaring significantly outperforms WAH in that regard, because (i) the partitioned in-memory layout of Roaring offers significantly faster updates and (ii) the better compression ratios of Roaring reduce the amount of memory occupied by

Compression method	avg. time per update [ns]	
	non-partitioned	partitioned
TEB	599	218
Roaring	480* / 574	121* / 216
WAH	17634	794

* using the in-memory layout (non serialized)

Table 3: The average time to apply an update.

pending updates. We therefore use Roaring as a differential data structure in the following and omit the results for WAH.

We measure the update throughput by applying 100k point updates to a compressed bitmap (with $n=2^{20}$, $d=0.1$, $f=8$) and report the average execution time. The number of pending updates is limited to 20k; i.e., a merge is triggered when this threshold is reached. Further, we examine how partitioning affects the execution time of point updates. We partition the bitmap into chunks of 2^{16} bits, whereas each chunk has its own diff. The results in Table 3 show that TEB and Roaring are on par, whereas WAH is several times slower. WAH suffers from the linear time complexity of point lookups that are involved with updates. Data partitioning helps to reduce the access latency significantly, but the average time of an update is still $3.6\times$ higher. The performance of Roaring on the

other hand could be improved by using its in-memory layout and its specialized XOR implementations for the individual container combinations (cf., the results marked with * in Table 3). The optimization is enabled by the fact that both the value bitmap and the differential bitmap are Roaring bitmaps. In a pure in-memory setting, Roaring therefore outperforms TEB by up to 1.8× and WAH by more than 6× in terms of update latency (in the partitioned case).

Pending updates naturally impair read latency. We observed a 30% penalty for TEB and Roaring with 20k pending updates (20% with WAH), irrespective of partitioning. For more general information on the trade-offs involved with differential updates, we refer the reader to UpBit [4].

5 RELATED WORK

Throughout the paper, we already covered the related work regarding bitmap compression techniques [2, 4, 15, 18, 22, 27, 32, 33, 57, 63], except for the HICAMP bitmap [56] which is designed for a special kind of memory system [13]. In the following, we discuss other related work.

Bitmap Indexes. Bitmap indexes and bitmap compression are orthogonal topics, as bitmap indexes may also be constructed with verbatim bitmaps. However, in practice, compression is commonly used to reduce space consumption and to improve query performance. Thus, the term bitmap index often refers to a *compressed* bitmap index. Compression, however, is just one aspect of a bitmap index. Other techniques that are involved when a bitmap index is constructed are (i) *binning* [28, 65, 66] which groups multiple attribute values together and (ii) *encoding* [11, 12, 46] which translates the bins into a set of bitmaps [64]. Thereby, an encoding scheme is chosen that best supports the query workload. Common encodings are equality encoding, range encoding and interval encoding, whereas the latter two allow for arbitrary range queries by accessing at most two bitmaps. Optionally, an attribute value may be *decomposed* into multiple components that are individually assigned to bins afterwards. A single attribute value may therefore map to multiple bins. An extreme case is the bit-sliced index [46, 50], where the attribute values are decomposed bit-by-bit, and the number of bins (and bitmaps) is equal to the bit-width of the attribute.

Binning, encoding, and decomposition influence the characteristics of the individual bitmaps [64] of an index. Consequently, they affect the overall index size and eventually the query performance [25, 62]. A thorough evaluation of TEBs within the large design space of bitmap indexes is therefore beyond the scope of this work.

Succinct Data Structures. The space efficiency of TEBs is founded on the idea of mapping tree nodes to integer values [30] and the foundational work on succinctly encoded binary

trees [24] that efficiently support the necessary navigational operations using the rank and select primitives. Both primitives require a helper structure to lower the time complexity of tree navigations from linear to constant time. Several implementations have been proposed [20, 21, 43, 55, 69] to achieve the performance of pointer-based tree structures. A key to success, in terms of performance, was the introduction of the population count instruction, which unfortunately was quite late in wide-spread x86 processors (AMD 2007, Intel 2008). Over the years, other succinct tree encodings have been proposed [6, 14, 17, 39, 40, 49] that support a richer set of operations or being updatable [41]; both, however, would incur higher space consumption and/or lower performance with TEB.

Lightweight Indexing. Space-efficient secondary 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 skipping (i) blocks of tuples [1, 38, 51, 52, 54, 67], (ii) scan ranges within blocks [29], or (iii) (parts of) individual tuples [19, 23, 35, 36, 48]. Other index structures were designed to support specific kinds of queries, e.g., queries with a LIMIT clause [26], or for specific kinds of data, e.g., observational data [58]. Most of these index structures rely on lightweight statistical data that is easy to maintain and query. The more heavyweight approaches either store approximations of the indexed columns [23, 51] or even require a different storage layout [19, 36].

6 CONCLUSION

The Tree-Encoded Bitmap (TEB) is a novel approach for compressing bitmaps. Its tree-based compression algorithm maps 0- or 1-runs of various lengths to binary tree nodes, where the depth of a node implicitly determines its run length. The resulting tree structure is then encoded using a succinct physical data structure that supports logarithmic access time and therefore allows for efficient logical operations (such as intersections) on compressed data. We experimentally showed that TEB saves considerable space compared to other compressed bitmap formats—in particular at higher bit densities, i.e., those cases where memory consumption would otherwise be fairly high. In terms of access speed, TEB is quite fast for intersection operations: almost as fast as the competing approach Roaring, and much faster than WAH. In the data distributions where TEB is strongest in saving space, its raw scan performance is also close to Roaring. As such, TEB encoded chunks could also be used as a worthwhile addition to the adaptive Roaring approach, significantly improving compression in the most difficult data distributions, while preserving performance.

Acknowledgements. This work was supported by the DFG project KE401/22.

REFERENCES

- [1] Karolina Alexiou, Donald Kossmann, and Per-Åke Larson. 2013. Adaptive Range Filters for Cold Data: Avoiding Trips to Siberia. *PVLDB* 6, 14 (2013), 1714–1725. <http://www.vldb.org/pvldb/vol6/p1714-kossmann.pdf>
- [2] G. Antoshenkov. 1995. Byte-aligned bitmap compression. In *Proceedings DCC '95 Data Compression Conference*. 476–. <https://doi.org/10.1109/DCC.1995.515586>
- [3] Manos Athanassoulis, Michael S. Kester, Lukas M. Maas, Radu Stolica, Stratos Dreuos, Anastasia Ailamaki, and Mark Callaghan. 2016. Designing Access Methods: The RUM Conjecture. In *Proceedings of the 19th International Conference on Extending Database Technology, EDBT 2016, Bordeaux, France, March 15-16, 2016*. 461–466. <https://doi.org/10.5441/002/edbt.2016.42>
- [4] Manos Athanassoulis, Zheng Yan, and Stratos Dreuos. 2016. UpBit: Scalable In-Memory Updatable Bitmap Indexing. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*. 1319–1332. <https://doi.org/10.1145/2882903.2915964>
- [5] The RoaringBitmap authors. [n.d.]. Roaring Bitmap. <https://github.com/RoaringBitmap/RoaringBitmap>. [Online; accessed 27-May-2019].
- [6] David Benoit, Erik D. Demaine, J. Ian Munro, Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. 2005. Representing Trees of Higher Degree. *Algorithmica* 43, 4 (2005), 275–292. <https://doi.org/10.1007/s00453-004-1146-6>
- [7] Boost.org. [n.d.]. Boost C++ Libraries. <https://www.boost.org/>. [Online; accessed 04-Jun-2019].
- [8] Michael Cain and Kent Milligan. 2011. IBM DB2 for i indexing methods and strategies. IBM White Paper.
- [9] Samy Chambi, Daniel Lemire, Robert Godin, Kamel Boukhalfa, Charles R. Allen, and Fangjin Yang. 2016. Optimizing Druid with Roaring bitmaps. In *Proceedings of the 20th International Database Engineering & Applications Symposium, IDEAS 2016, Montreal, QC, Canada, July 11-13, 2016*. 77–86. <https://doi.org/10.1145/2938503.2938515>
- [10] Samy Chambi, Daniel Lemire, Owen Kaser, and Robert Godin. 2014. Better bitmap performance with Roaring bitmaps. *CoRR abs/1402.6407* (2014). arXiv:1402.6407 <http://arxiv.org/abs/1402.6407>
- [11] Chee Yong Chan and Yannis E. Ioannidis. 1998. Bitmap Index Design and Evaluation. In *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA*. 355–366. <https://doi.org/10.1145/276304.276336>
- [12] Chee Yong Chan and Yannis E. Ioannidis. 1999. An Efficient Bitmap Encoding Scheme for Selection Queries. In *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadelphia, Pennsylvania, USA*. 215–226. <https://doi.org/10.1145/304182.304201>
- [13] David R. Cheriton, Amin Firoozshahian, Alex Solomatnikov, John P. Stevenson, and Omid Azizi. 2012. HICAMP: architectural support for efficient concurrency-safe shared structured data access. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2012, London, UK, March 3-7, 2012*. 287–300. <https://doi.org/10.1145/2150976.2151007>
- [14] David R. Clark and J. Ian Munro. 1996. Efficient Suffix Trees on Secondary Storage (SODA '96), Vol. 96. Society for Industrial and Applied Mathematics, USA, 383–391.
- [15] Alessandro Colantonio and Roberto Di Pietro. 2010. Concise: Compressed 'n' Composable Integer Set. *Inf. Process. Lett.* 110, 16 (2010), 644–650. <https://doi.org/10.1016/j.ipl.2010.05.018>
- [16] Oracle Corporation. 2005. Bitmap Index vs. B-tree Index: Which and When? <https://www.oracle.com/technetwork/articles/sharma-indexes-093638.html>. [Online; accessed 14-Jun-2019].
- [17] Pooya Davoodi, Rajeev Raman, and Srinivasa Rao Satti. 2017. On Succinct Representations of Binary Trees. *Mathematics in Computer Science* 11, 2 (2017), 177–189. <https://doi.org/10.1007/s11786-017-0294-4>
- [18] François Delière and Torben Bach Pedersen. 2010. Position list word aligned hybrid: optimizing space and performance for compressed bitmaps. In *EDBT 2010, 13th International Conference on Extending Database Technology, Lausanne, Switzerland, March 22-26, 2010, Proceedings*. 228–239. <https://doi.org/10.1145/1739041.1739071>
- [19] Ziqiang Feng, Eric Lo, Ben Kao, and Wenjian Xu. 2015. 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*. 31–46. <https://doi.org/10.1145/2723372.2747642>
- [20] Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. 2014. From Theory to Practice: Plug and Play with Succinct Data Structures. In *13th International Symposium on Experimental Algorithms, (SEA 2014)*. 326–337.
- [21] Rodrigo González, Szymon Grabowski, Veli Mäkinen, and Gonzalo Navarro. 2005. Practical implementation of rank and select queries. In *Poster Proc. Volume of 4th Workshop on Efficient and Experimental Algorithms (WEA)*. 27–38.
- [22] Gheorghe Guzun, Guadalupe Canahuate, David Chiu, and Jason Sawin. 2014. A tunable compression framework for bitmap indices. In *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*. 484–495. <https://doi.org/10.1109/ICDE.2014.6816675>
- [23] Brian Hentschel, Michael S. Kester, and Stratos Dreuos. 2018. 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*. 857–872. <https://doi.org/10.1145/3183713.3196911>
- [24] Guy Jacobson. 1989. Space-efficient Static Trees and Graphs. In *30th Annual Symposium on Foundations of Computer Science, Research Triangle Park, North Carolina, USA, 30 October - 1 November 1989*. 549–554. <https://doi.org/10.1109/SFCS.1989.63533>
- [25] Theodore Johnson. 1999. Performance Measurements of Compressed Bitmap Indices. In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*. 278–289. <http://www.vldb.org/conf/1999/P29.pdf>
- [26] Albert Kim, Liqi Xu, Tarique Siddiqui, Silu Huang, Samuel Madden, and Aditya G. Parameswaran. 2016. Speedy Browsing and Sampling with NeedleTail. *CoRR abs/1611.04705* (2016). arXiv:1611.04705 <http://arxiv.org/abs/1611.04705>
- [27] Sangchul Kim, Junhee Lee, Srinivasa Rao Satti, and Bongki Moon. 2016. SBH: Super byte-aligned hybrid bitmap compression. *Inf. Syst.* 62 (2016), 155–168. <https://doi.org/10.1016/j.is.2016.07.004>
- [28] Nick Koudas. 2000. Space Efficient Bitmap Indexing. In *Proceedings of the 2000 ACM CIKM International Conference on Information and Knowledge Management, McLean, VA, USA, November 6-11, 2000*. 194–201. <https://doi.org/10.1145/354756.354819>
- [29] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. 2016. 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*. 311–326. <https://doi.org/10.1145/2882903.2882925>
- [30] C. C. Lee, D. T. Lee, and C. K. Wong. 1986. Generating Binary Trees of Bounded Height. *Acta Inf.* 23, 5 (1986), 529–544. <https://doi.org/10.1007/BF00288468>
- [31] Daniel Lemire. [n.d.]. Official Roaring Bitmap website. <https://roaringbitmap.org>. [Online; accessed 27-May-2019].

- [32] Daniel Lemire, Gregory Ssi Yan Kai, and Owen Kaser. 2016. Consistently faster and smaller compressed bitmaps with Roaring. *Softw. Pract. Exper.* 46, 11 (2016), 1547–1569. <https://doi.org/10.1002/spe.2402>
- [33] Daniel Lemire, Owen Kaser, and Kamel Aouiche. 2010. Sorting improves word-aligned bitmap indexes. *Data Knowl. Eng.* 69, 1 (2010), 3–28. <https://doi.org/10.1016/j.datak.2009.08.006>
- [34] Daniel Lemire, Owen Kaser, and Eduardo Gutarra. 2012. Reordering rows for better compression: Beyond the lexicographic order. *ACM Trans. Database Syst.* 37, 3 (2012), 20:1–20:29. <https://doi.org/10.1145/2338626.2338633>
- [35] Yanan Li, Craig Chasseur, and Jignesh M. Patel. 2015. A Padded Encoding Scheme to Accelerate Scans by Leveraging Skew. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*. 1509–1524. <https://doi.org/10.1145/2723372.2737787>
- [36] Yanan Li and Jignesh M. Patel. 2013. 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*. 289–300. <https://doi.org/10.1145/2463676.2465322>
- [37] Roger MacNicol and Blaine French. 2004. Sybase IQ Multiplex - Designed For Analytics. In *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, VLDB 2004, Toronto, Canada, August 31 - September 3 2004*. 1227–1230. <https://doi.org/10.1016/B978-012088469-8.50111-X>
- [38] Guido Moerkotte. 1998. Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing. In *VLDB '98, Proceedings of 24th International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA*. 476–487. <http://www.vldb.org/conf/1998/p476.pdf>
- [39] J. Munro and V. Raman. 2001. Succinct Representation of Balanced Parentheses and Static Trees. *SIAM J. Comput.* 31, 3 (2001), 762–776. <https://doi.org/10.1137/S0097539799364092>
- [40] J. Ian Munro and Venkatesh Raman. 1997. Succinct Representation of Balanced Parentheses, Static Trees and Planar Graphs. In *38th Annual Symposium on Foundations of Computer Science, FOCS '97, Miami Beach, Florida, USA, October 19-22, 1997*. 118–126. <https://doi.org/10.1109/SFCS.1997.646100>
- [41] J. Ian Munro, Venkatesh Raman, and Adam J. Storm. 2001. Representing dynamic binary trees succinctly. In *Proceedings of the Twelfth Annual Symposium on Discrete Algorithms, January 7-9, 2001, Washington, DC, USA*. 529–536. <http://dl.acm.org/citation.cfm?id=365411.365526>
- [42] Parth Nagarkar, K. Selçuk Candan, and Aneesha Bhat. 2015. Compressed Spatial Hierarchical Bitmap (cSHB) Indexes for Efficiently Processing Spatial Range Query Workloads. *PVLDB* 8, 12 (2015), 1382–1393. <http://www.vldb.org/pvldb/vol8/p1382-nagarkar.pdf>
- [43] Gonzalo Navarro and Eliana Provedel. 2012. Fast, Small, Simple Rank/Select on Bitmaps. In *Experimental Algorithms - 11th International Symposium, SEA 2012, Bordeaux, France, June 7-9, 2012. Proceedings*. 295–306. https://doi.org/10.1007/978-3-642-30850-5_26
- [44] Patrick O'Neil and Goetz Graefe. 1995. Multi-table Joins Through Bitmapped Join Indices. *SIGMOD Rec.* 24, 3 (Sept. 1995), 8–11. <https://doi.org/10.1145/211990.212001>
- [45] Patrick E. O'Neil. 1987. Model 204 Architecture and Performance. In *High Performance Transaction Systems, 2nd International Workshop, Asilomar Conference Center, Pacific Grove, California, USA, September 28-30, 1987, Proceedings*. 40–59. https://doi.org/10.1007/3-540-51085-0_42
- [46] Patrick E. O'Neil and Dallon Quass. 1997. Improved Query Performance with Variant Indexes. In *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA*. 38–49. <https://doi.org/10.1145/253260.253268>
- [47] Ali Pinar, Tao Tao, and Hakan Ferhatosmanoglu. 2005. Compressing Bitmap Indices by Data Reorganization. In *Proceedings of the 21st International Conference on Data Engineering, ICDE 2005, 5-8 April 2005, Tokyo, Japan*. 310–321. <https://doi.org/10.1109/ICDE.2005.35>
- [48] Orestis Polychroniou and Kenneth A. Ross. 2015. 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*. 9:1–9:6. <https://doi.org/10.1145/2771937.2771943>
- [49] Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. 2007. Succinct indexable dictionaries with applications to encoding k -ary trees, prefix sums and multisets. *ACM Trans. Algorithms* 3, 4 (2007), 43. <https://doi.org/10.1145/1290672.1290680>
- [50] Denis Rinfret, Patrick E. O'Neil, and Elizabeth J. O'Neil. 2001. Bit-Sliced Index Arithmetic. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data, Santa Barbara, CA, USA, May 21-24, 2001*. 47–57. <https://doi.org/10.1145/375663.375669>
- [51] Lefteris Sidirourgos and Martin L. Kersten. 2013. 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*. 893–904. <https://doi.org/10.1145/2463676.2465306>
- [52] Lefteris Sidirourgos and Hannes Mühleisen. 2017. Scaling column imprints using advanced vectorization. In *Proceedings of the 13th International Workshop on Data Management on New Hardware, DaMoN 2017, Chicago, IL, USA, May 15, 2017*. 4:1–4:8. <https://doi.org/10.1145/3076113.3076120>
- [53] Rishi Rakesh Sinha and Marianne Winslett. 2007. Multi-resolution bitmap indexes for scientific data. *ACM Trans. Database Syst.* 32, 3 (2007), 16. <https://doi.org/10.1145/1272743.1272746>
- [54] The PostgreSQL Global Development Group. [n.d.]. Block Range Index (BRIN) in PostgreSQL. <https://www.postgresql.org/docs/11/brin.html>. [Online; accessed 01-Jul-2019].
- [55] Sebastiano Vigna. 2008. Broadword Implementation of Rank/Select Queries. In *Experimental Algorithms, 7th International Workshop, WEA 2008, Provincetown, MA, USA, May 30-June 1, 2008, Proceedings*. 154–168. https://doi.org/10.1007/978-3-540-68552-4_12
- [56] Bo Wang, Heiner Litz, and David R. Cheriton. 2014. HICAMP bitmap: space-efficient updatable bitmap index for in-memory databases. In *Tenth International Workshop on Data Management on New Hardware, DaMoN 2014, Snowbird, UT, USA, June 23, 2014*. 7:1–7:7. <https://doi.org/10.1145/2619228.2619235>
- [57] Jianguo Wang, Chunbin Lin, Yannis Papanikolaou, and Steven Swanson. 2017. An Experimental Study of Bitmap Compression vs. Inverted List Compression. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*. 993–1008. <https://doi.org/10.1145/3035918.3064007>
- [58] Sheng Wang, David Maier, and Beng Chin Ooi. 2014. Lightweight Indexing of Observational Data in Log-Structured Storage. *PVLDB* 7, 7 (2014), 529–540. <http://www.vldb.org/pvldb/vol7/p529-wang.pdf>
- [59] J. W. J. Williams. 1964. Algorithm 232: Heapsort. *Commun. ACM* 7, 6 (1964), 347–348.
- [60] John Wu and Kurt Stockinger. [n.d.]. FastBit: An Efficient Compressed Bitmap Index Technology. <https://sdm.lbl.gov/fastbit/>. [Online; accessed 27-May-2019].
- [61] Kesheng Wu, Sean Ahern, E Wes Bethel, Jacqueline Chen, Hank Childs, Estelle Cormier-Michel, Cameron Geddes, Junmin Gu, Hans Hagen, Bernd Hamann, et al. 2009. FastBit: interactively searching massive data. In *Journal of Physics: Conference Series*, Vol. 180. IOP Publishing, 012053.

- [62] Kesheng Wu, Ekow J. Otoo, and Arie Shoshani. 2004. On the performance of bitmap indices for high cardinality attributes. In *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, VLDB 2004, Toronto, Canada, August 31 - September 3 2004*. 24–35. <https://doi.org/10.1016/B978-012088469-8.50006-1>
- [63] Kesheng Wu, Ekow J. Otoo, and Arie Shoshani. 2006. Optimizing bitmap indices with efficient compression. *ACM Trans. Database Syst.* 31, 1 (2006), 1–38. <https://doi.org/10.1145/1132863.1132864>
- [64] Kesheng Wu, Arie Shoshani, and Kurt Stockinger. 2010. Analyses of multi-level and multi-component compressed bitmap indexes. *ACM Trans. Database Syst.* 35, 1 (2010), 2:1–2:52. <https://doi.org/10.1145/1670243.1670245>
- [65] Kun-Lung Wu and Philip S. Yu. 1998. Range-Based Bitmap Indexing for High Cardinality Attributes with Skew. In *COMPSAC '98 - 22nd International Computer Software and Applications Conference, August 19-21, 1998, Vienna, Austria*. 61–67. <https://doi.org/10.1109/CMPSAC.1998.716637>
- [66] Ming-Chuan Wu and Alejandro P. Buchmann. 1998. Encoded Bitmap Indexing for Data Warehouses. In *Proceedings of the Fourteenth International Conference on Data Engineering, Orlando, Florida, USA, February 23-27, 1998*. 220–230. <https://doi.org/10.1109/ICDE.1998.655780>
- [67] Jia Yu and Mohamed Sarwat. 2016. Two Birds, One Stone: A Fast, yet Lightweight, Indexing Scheme for Modern Database Systems. *PVLDB* 10, 4 (2016), 385–396. <http://www.vldb.org/pvldb/vol10/p385-yu.pdf>
- [68] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G. Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. 2018. SuRF: Practical Range Query Filtering with Fast Succinct Tries. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*. 323–336. <https://doi.org/10.1145/3183713.3196931>
- [69] Dong Zhou, David G. Andersen, and Michael Kaminsky. 2013. Space-Efficient, High-Performance Rank and Select Structures on Uncompressed Bit Sequences. In *Experimental Algorithms, 12th International Symposium, SEA 2013, Rome, Italy, June 5-7, 2013. Proceedings*. 151–163. https://doi.org/10.1007/978-3-642-38527-8_15