CSC2525 Project Report

Guy Khazma

Abstract

Filter Data Structures are widely used in wide range of applications to answer approximate set-membership queries. These filters typically require pre-allocation with a specified capacity to ensure a certain false positive rate. In many instances, as data dynamically grows, the associated filters must expand accordingly. State-of-the-art designs achieve constant time for all access operations. However, these designs require doubling the array on each expansion resulting in wasteful memory allocation right after expansion.In this project, we will investigate a novel approach employing dynamic numeral systems for flexible filter expansion, allowing factors other than 2. This involves encoding key hashes with a base matching the hash table's slot count and utilizing the least significant digit for slot allocation. We conduct experimental performance evaluation of this technique and outline potential directions for future work.

1 Introduction

Filters. A filter is a compact probabilistic data structure used to represent keys in a set [14]. Filters are typically stored in memory and used to answer set-membership queries. A filter cannot return false negatives but may return false positives, the probability of which depends on the amount of memory assigned to it. Filters are widely employed across various applications [4, 8, 23, 24], particularly in KV stores to streamline data retrieval from storage [13, 15–17, 19].

Expandable Filters. In numerous applications relying on filters, data sizes typically increase over time, underscoring the need for filters that can expand accordingly. For example, modern log-structured key-value stores utilize in-memory tabular filters to efficiently track data locations [17]. Expandable filters are also vital in networking for tasks like managing blacklists and enabling multicast routing [32], and implementing virtual page tables [27].

Generic Expansion Methods. Numerous generic methods are available for expanding a filter, all of which facilitate expansion by a growth factor of $1 + \alpha$ for any $\alpha > 0$ [14]. The simplest approach is *full reconstruction*, which involves allocating a larger filter from scratch, re-reading the original data, and inserting it while discarding the original filter. For applications that do not frequently scan the entire dataset, this approach could incur prohibitively high costs. Alternatively, *chaining*, is a method in which a new filter is assigned once the previous one reaches capacity. However, this strategy increases the cost for queries and deletions because, in the worst-case, all filters must be searched. Furthermore, assuming a fixed number of bits per key, the false positive rate increases linearly with the number of filters.

Tabular Filters. As an alternative to Bloom filters [6], *tabular filters* have gained prominence over the past decade or so. These filters function by storing a fingerprint (hash digest) for each key within a compact hash table. Queries are processed by verifying if the fingerprints in a given hash slot match the fingerprint of the target key. Various tabular filters have been proposed with different hash collision resolution strategies (e.g., cuckoo [7], robin hood[11], or two-choice hashing[26]). Unlike traditional filters, many tabular filters support deletes and can associate a payload with each fingerprint enabling them to support dynamic applications like key-value stores [17] and duplicate detection in streams [21].

Fingerprint Sacrifice. Tabular filters introduce an additional mechanism for expansion called *fingerprint sacrifice*. Typically, expanding a tabular filter involves allocating a hash table with twice the capacity and migrating each fingerprint into this larger hash table. However, this process requires sacrificing one bit from each entry's fingerprint to contribute to its slot address, ensuring a balanced distribution of fingerprints across the expanded hash table. Consequently, as the data grows, the size of the fingerprints decreases, resulting in an increase in the false positive rate (FPR). Eventually, the fingerprints exhaust their bits becoming *void entries*, rendering the filter ineffective as it starts returning positive results for any query.

The State of the Art :Aleph Filter. Aleph Filter [2] is a recent tabular filter that can grow indefinitely while maintaining a stable performance, memory footprint, and false positive rate at the same time. Aleph Filter leverages *fingerprint sacrifice* and expands by migrating all fingerprints into a larger hash table and sacrificing one bit from each fingerprint in the process. The main features which enable it to scale are: (1) A hash slot format which enables to set longer fingerprints to newer entries, keeping the FPR stable, (2) Duplicating void entries on expansion to optimize queries by accessing only one hash table for any query at a cost of $O(1)$ time, (3) Using tombstones to indicate deletes pushing the garbage collection of duplicated entries to the next expansion.

Problem: Wasteful Memory Allocation The standard expansion technique in approaches based on fingerprint sacrifice involves doubling the filter's capacity with each expansion (or more generally, forcing $1 + \alpha$ to be a factor of 2). This leads to 50% of the allocated memory for the filter being left unused right after expansion, presenting a challenge, especially with limited memory resources. Is there a way to expand the filter by a factor smaller than 2 while still maintaining comparable FPR to that of Aleph Filter?

Insight: Dynamic Numeral System Our key insight lies in leveraging a dynamic numeral system to enable the expansion of a filter by any growth factor $1 + \alpha$. This involves representing the hash of the key using a base that aligns with the number of slots in the expanded filter. Consequently, the slot associated with a key corresponds to the least significant digit of the hash in this representation. During expansion, we apply this approach by deriving the original hash of the key and encoding it using the new base. Subsequently, we reinsert it into the expanded hash table, placing it in the slot corresponding to the least significant digit in the new representation, while storing the remainder of the number as the fingerprint.

This Project. In this project, we investigate the integration of the dynamic numeral system with the Quotient Filter. We implement the integration and conduct experimental analysis. Finally, we identify potential future directions informed by our implementation and evaluation.

2 Background

In this section, we present the notation utilized throughout this paper and give background details on the Quotient Filter, which serves as the foundation for Aleph Filter. While the method outlined here is based on the Quotient Filter, further investigation is necessary as part of this project to adapt it for use with Aleph Filter. Therefore, we postpone the background description of Aleph Filter to the final report.

2.1 Notation

Let y be an integer in decimal representation. We use $[y]_b$ to denote the representation of y in base b. For example, for the decimal number 105 ($y = [105]_{10}$) its binary representation (base 2) is $[y]_2 = [01101001]_2$ and its hexadecimal representation is $[y]_{16} = [69]_{16}$.

2.2 Quotient Filter

A quotient filter [5, 20] is a hash table that stores a fingerprint for each inserted key. Each key is assigned a *mother hash* consisting of $F + S$ bits using some hash function $h(...)$. The least significant S bits of this mother hash denote the key's *canonical slot*, while the subsequent F bits represent its fingerprint. To resolve collisions, the Quotient Filter utilizes round-robin hashing [10]. Upon insertion, fingerprints are mapped to their canonical slots and displace any colliding fingerprints to the right. A sequence of contiguous fingerprints belonging to the same canonical slot is termed a *run* while a *cluster* refers to several adjacent runs where all but the first have been displaced to the right due to collisions. To mark the starts and end of clusters and runs, each slot contains metadata bits. The details of these metadata bits, as well as the specifics of access operations (query/insert/delete) in the quotient filter, are excluded from discussion, as they are not impacted by the technique proposed in this document.

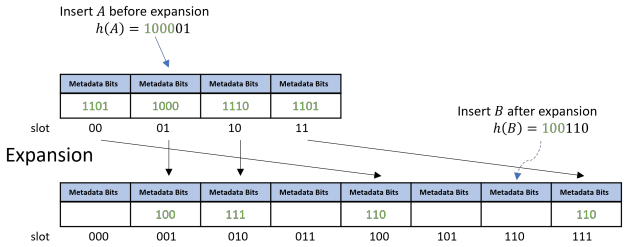

Figure 1. Fingerprint sacrifice method in quotient filter. Expansion doubles the hash table capacity and one bit is transferred from the fingerprint to the slot address. We use black and green in the figure to illustrate the slot addresses and fingerprints, respectively. Note that after the expansion 4 slots out of 8 slots (50%) are empty.

False Positive Rate. The false positive rate of quotient filter is determined by the fingerprint size F and the the fraction of occupied slots β and can be approximated as $\approx \beta \cdot 2^{-F}$. The rationale behind this is that, typically, on average, the target run contains β fingerprints, with each having a probability of 2^{-F} of matching the key being searched.

Expansion. Quotient Filter expands using the fingerprint sacrifice approach. It does so by first recovering the mother hash of every entry by concatenating its slot offset with its fingerprint. It then allocates a new filter with doubled capacity and re-insert all of the keys from left to right. For entry in slot s, it remains in the same slot in the new filter if its least significant fingerprint bit is zero (prior to expansion). Otherwise, it is placed at slot $s + n'/2$ where n' is the number of slots in the new filter. Figure 1 shows an example of expanding a filter from four to eight slots. For example, the entry in slot 0 is mapped to slot $[4]_{10} = [100]_2$ since the least significant bit in the fingerprint prior to expansion is set to 1. Following expansion, we utilize three bits instead of two to map an element to its slot, resulting in a fingerprint that is shorter by one bit.

3 Dynamic Numeral System

The number base, or simply *base*, of a numeral system indicates the unique symbols and notations it employs to represent a value. For example, the number base 2 (binary) indicates that there are only two unique notations 0 and 1. Our observation is that representing a hash digest in a number base equal to the number of slots in the underlying hash table offers a straightforward mapping. Using this method, we can designate the least significant digit to correspond to the slot. With the number of unique symbols representing the digit matching the number of slots, we guarantee that any value is accurately mapped to a valid slot.

This generalization captures the doubling approach utilized by the quotient filter. Figure 2 (a) illustrates the same filter depicted in Figure 1 prior to expansion, employing base

4 instead of the standard binary encoding, as there are 4 slots available. In the figure, each cell's representation is given in base 4, with its binary equivalent shown in parentheses. The binary representation is used for storage, while the base 4 representation helps logically partition the mother hash into slot addresses and fingerprints. Consider $h(A) = [100001]_2$. In Figure 1, the three least significant bits are used to map it to slot 1, with the remaining bits stored as the fingerprint. Using the dynamic numeral system, $[100001]_2 = [201]_4$, indicating that the least significant digit, 1, maps it to slot 1, while the remainder, $[20]_4$, serves as the fingerprint. In this scenario, the binary representation of the fingerprint is equivalent in both bases (2 and 4). However, this equivalence does not hold true for bases that are not powers of two, as discussed below.

In Figure 2 (b), we demonstrate the application of the dynamic numeral system to expand the filter in factors smaller than 2. Here, we expand the filter by 50% to size 6. This process involves recovering the mother hash of each entry, encoding it in base 6, and mapping it to the relevant slot using the least significant digit, while storing the rest of the digits as the fingerprint. For instance, consider entry e_3 in slot 3 with a mother hash of $h(e_3) = [313]_4 = [110111]_2$. It is mapped to slot 1 in the expanded filter as $h(e_3) = [131]_6$. Its new fingerprint is $[13]_6 = [1001]_2$. For any subsequent insertion, the same base is utilized. For instance, consider $h(B) = [100110]_2 = [102]_6$. It is mapped to slot 2 with a fingerprint of $[10]_6$. Finally, figure 1 (c) depicts expanding to 8 slots, which is a power of two, we observe that its binary representation matches the expansion in figure 1.

In contrast to the doubling approach, we note that the fingerprint needs to retain the same number of bits after expansion for some bases. For example, in figure 2 which use base 6, the entry in slot 1 requires 4 bits to represent the fingerprint in binary, just as it did before the expansion. However, because we are not doubling the capacity, we are not fully utilizing the range of fingerprints that can be expressed using 4 bits. Using 4 bits enables to express 2^4 different fingerprints while in practice we will have up to 10 unique fingerprints since "some" of the representation is now incorporated to the slot number. This implies that the false positive rate in bases which are not factors of 2 can no longer be approximated as $\approx \beta \cdot 2^{-F}$ where F is the number of bits in the fingerprint.

Formally, let's assume the mother hash $h(\cdot)$ comprises of M bits, and the filter currently contains n slots. If we expand it by a factor of $1 + \alpha$, its new size becomes $n' = \lfloor (1 + \alpha)n \rfloor$. The expansion process start by allocating a new quotient filter with capacity of n'. It then iterates over the smaller quotient filter from left to right and derives for each fingerprint its mother hash by concatenating the slot address with the fingerprint using base n. Using the recovered hash we compute the slot address and fingerprint of the entry in the new filter represented with the new base n' as follows:

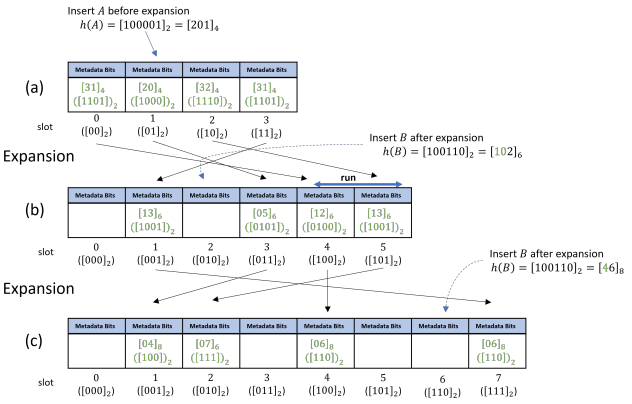

Figure 2. Fingerprint sacrifice method in using dynamic numeral system. Part (a) illustrate doubles the hash table capacity and one bit is transferred from the fingerprint to the slot address. We use black and green in the figure to illustrate the slot addresses and fingerprints, respectively.

$$\text{slot} = H(x) \ \% \ n' \tag{1}$$

$$\text{fingerprint} = \lfloor \frac{H(x)}{n'} \rfloor \tag{2}$$

Where % is the modulo operation. In other words, the slot corresponds to the least significant digit in base n', while the fingerprint encode the division of the hash by the base (in base 2). Given the fingerprint $fp(x)$ in slot s we can reconstruct the mother hash using

$$H(x) = n \cdot fp(x) + s \tag{3}$$

The number of bits required to represent the new fingerprint is $\lfloor \log_2 \frac{2^M-1}{n'} \rfloor + 1$. The maximum number of unique fingerprints in this new representation is $\lfloor \frac{2^M-1}{n'} \rfloor$. Since we use $\lfloor \log_2 \frac{2^M-1}{n'} \rfloor + 1$ bits to encode the fingerprint, $2^{\lfloor \log_2 \frac{2^M-1}{n'} \rfloor + 1} - \lfloor \frac{2^M-1}{n'} \rfloor$ encodings are not used thus affecting the FPR.

4 Implementation

We implemented the Dynamic Numeral System by integrating it with the current implementation of the Quotient Filter and Fingerprint Sacrifice method from Aleph Filter's implementation [2]. The original implementation assumed that the filter size was a power of two. Thus, our initial step involved refactoring the code to allow for extending a filter with arbitrary factors. In the process, we also refactored the library to separate the current tests into a separate package and used Maven to build the package [1].

In our implementation, we employ equations 1 and 2 to determine the slot and fingerprint, respectively, throughout insertion, querying, and expansion stages. Notably, during expansion, we utilize equation 3 to calculate the mother hash. Specifically in Java these are implemented as:

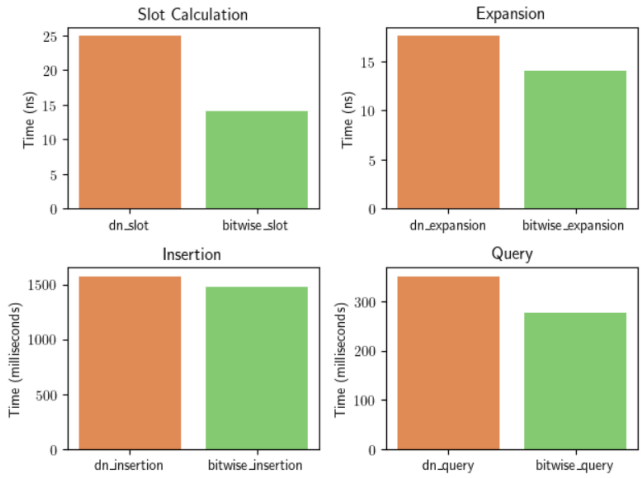

Figure 3. Comparison of run time for the Dynamic Numeral System vs bitwise operations. Top left and top right display avg run time for slot calculation and expansion respectively. The bottom left corner displays the total insertion time for inserting 2^{20} entries into a filter initially sized 8. The bottom right corner illustrates the runtime for issuing 2^{20} queries to the same filter.

```
// (1) slot
int slot = (int) Math.floorMod(hash, size);

// (2) fingerprint
long fingerprint_mask = (1L << fingerprintLength) - 1L;
long fingerprint = Math.floorDiv(hash, size) & fingerprint_mask;

// (3) motherhash
long mother_hash = size*fingerprint + bucket;
```

We use a mask to extract the relevant bits of the fingerprint.

When the target size is a power of two, equations 1 and 2 can be expedited using bitwise operations for slot and fingerprint extraction, as implemented in the current code. Similarly, for recovering the mother hash when the original size is a power of two, concatenating the binary representations of the slot and fingerprint offers optimization. However, we opted to use the formulas directly to assess the overhead introduced by employing `floorDiv` and `floorMod` operations, which typically require more cycles (30-60 cycles) compared to the minimal cycles required by bitwise operations. While the derivation of the mother hash may exhibit slower performance, it's worth noting that modern processors often handle addition and multiplication operations as efficiently as bitwise operations [22].

5 Evaluation

In this section, we evaluate the implementation of the dynamic numeral system by comparing it to the existing method that utilizes bitwise operations. This comparison aims to determine the extent of overhead introduced by using the dynanmic numeral system.

5.1 Microbenchmarks

First, we conduct runtime benchmarking for the implementation of each equation outlined in Section 4. To ensure comparability with the existing code, our evaluation focuses on filter sizes that are powers of two, allowing for the utilization of bitwise operations to extract the slot and fingerprint, as well as to calculate the new slot and fingerprint during expansion. In our comparison, we analyze the runtime for slot calculation using `floorMod` in contrast to the bitwise approach, and for fingerprint computation using `floorDiv` compared to the bitwise approach. Additionally, we benchmark the computation required during expansion, which involves computing the mother hash followed by extracting the slot and fingerprint using the new size. This is then compared to the bitwise approach, which involves simply shifting a bit from the fingerprint to the slot.

To ensure consistent performance measurement, we first warm up the JVM by executing each function implementation one million times. Subsequently, we assess the average latency for each function by running it 2^20 times. This involves generating a random long integer, calculating the slot index using equation 1 and the existing bitwise method, computing the fingerprint of size 27 bits using Equation 2 and the existing method, and simulating expansion. For expansion, we use the generated slot and fingerprint to deduce the fingerprint and slot for the next expansion (2^21). This process is carried out once by recovering the mother hash using Equation 3 and once using the existing bitwise shifting method. We measure the average runtime for each function.

Figure 3 displays the outcomes of our analysis. Initially, we omit plotting the average calculation time for fingerprints since it remains nearly identical in both approaches, approximately 14ns. Moving forward, in the top-left section, we observe that slot calculation using the dynamic numeral approach lags by about 70% compared to the bitwise method. Nonetheless, this difference appears relatively inconsequential given the rapidity of the calculation, approximately 24ns versus 14ns. A similar, albeit smaller, discrepancy is observed in the expansion calculation, where the dynamic numeral approach trails by an average of 25%. It's noteworthy that the execution speed is remarkably swift, resulting in fluctuating average latency calculations. Moreover, the exact runtime is subject to variability based on the sequence of operations, which may undergo optimization over time by the JVM.

5.2 Macrobenchmarks

We now conduct a macrobenchmark to assess the overall performance of the new implementation. This benchmark involves initializing a filter with 8 slots and a fingerprint size of 27 bits. We then proceed to insert 2^{20} random entries into the filter, ensuring that expansions are triggered during this process. Following the insertion phase, we execute 2^{20} search queries to validate the presence of all inserted keys.

Throughout both stages, we measure the total runtime. This experiment is repeated 5 times, and we report the average runtime for each phase. The results depicted in Figure 3 confirm our expectations: while the slowdown in calculating the slot index does lead to a slight variance in runtime, this difference is not substantial.

6 Future Work

Our current implementation suggests that the dynamic numeral system is a feasible solution for scaling a filter in factors smaller than 2 (or powers of two), allowing for a more gradual expansion of the filter and mitigating wasteful memory allocation. We intend to investigate which expansion factors yield a balanced performance between memory usage and False Positive Rate. Furthermore, we aim to extend our implementation and employ a similar technique for InfiniFilter [14] and AlephFilter [2].

7 Related Work

Pre-allocation. One common solution for managing filter expansion is pre-allocating a large static filter. For instance, the Pliops data processor reserves a significant static filter of about 100GB upon system deployment to map data entries to their storage locations [18]. However, this approach demands substantial system memory from the outset, regardless of the dataset size, and imposes a maximum limit on the number of entries that can be accommodated.

Chaining based expansion methods. A commonly used method to expand filters involves allocating new empty filters into which more insertions can be made as the current filter approaches capacity [3, 12, 25, 30]. However, despite their theoretical flexibility in expanding a filter by any growth factor, these methods frequently entail increased access costs, as all filters may need to be considered in worst-case scenarios. In contrast, fingerprint sacrifice-based expansion, demonstrated by Aleph Filter and InfiniFilter, reliably achieves constant-time access operations at any scale.

Optimally re-sizable block based arrays. Optimally resizing an array was studied in theory [9, 28]. These works assume a model where any block of memory can be accessed in constant time given the block pointer and an index into the block. The goal is to expand the array by adding new indexes on the right. Brodnik et al. [9] present a method that maintains a resizable array of size N with only $N + O(\sqrt{N})$ space, while still achieving $O(1)$ time per operation. Recent work [28] extended these results by distinguishing between the space needed to store the re-sizable array and accessing its item and the temporary space needed while growing/shrinking. We aim to explore the potential application of these insights in dynamically allocated filters as future work.

Optimally re-sizable contiguous arrays. Java's ArrayList employs a resizing strategy by allocating a larger memory block, typically 1.5 times the current size, and copying existing data into the new block while appending new elements. This resizing factor aids in memory self-compaction, reducing fragmentation [31]. Research indicates that the golden ratio, 1.618, is the optimal growth rate for dynamically allocated contiguous arrays [29]. Our method, which enables resizing at any factor, allows us to explore these principles in the context of filters, potentially improving resize efficiency in filter operations.

8 Conclusion

Traditional approaches to filter expansion necessitate doubling the filter's size, leading to inefficient memory allocation. In this project, we introduced and implemented a novel method that allows the filter to expand by any factor using a dynamic numeral system. Our experiments demonstrate that the overhead of this method is negligible, offering a promising avenue for exploring optimal expansion factors.

References

[1] 2023. Maven. https://maven.apache.org/.

[2] 2024. Aleph Filter: To Infinity in Constant Time . https://github.com/nivdayan/AlephFilter.

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

[4] Ronald Barber, Guy Lohman, Ippokratis Pandis, Vijayshankar Raman, Richard Sidle, Gopi Attaluri, Naresh Chainani, Sam Lightstone, and David Sharpe. 2014. Memory-efficient hash joins. *Proceedings of the VLDB Endowment* 8, 4 (2014), 353–364.

[5] Michael A Bender, Martin Farach-Colton, Rob Johnson, Bradley C Kuszmaul, Dzejla Medjedovic, Pablo Montes, Pradeep Shetty, Richard P Spillane, and Erez Zadok. 2011. Don't thrash: How to cache your hash on flash. In *3rd Workshop on Hot Topics in Storage and File Systems (HotStorage 11)*.

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

[7] Alex D Breslow and Nuwan S Jayasena. 2018. Morton filters: faster, space-efficient cuckoo filters via biasing, compression, and decoupled logical sparsity. *Proceedings of the VLDB Endowment* 11, 9 (2018), 1041–1055.

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

[9] Andrej Brodnik, Svante Carlsson, Erik D Demaine, J Ian Ian Munro, and Robert Sedgewick. 1999. Resizable arrays in optimal time and space. In *Algorithms and Data Structures: 6th International Workshop, WADS'99 Vancouver, Canada, August 11–14, 1999 Proceedings 6*. Springer, 37–48.

[10] Pedro Celis, Per-Åke Larson, and J. Ian Munro. 1985. Robin Hood Hashing (Preliminary Report). In *26th Annual Symposium on Foundations of Computer Science, Portland, Oregon, USA, 21-23 October 1985*. IEEE Computer Society, 281–288. https://doi.org/10.1109/SFCS.1985.48

[11] Pedro Celis, Per-Ake Larson, and J Ian Munro. 1985. Robin hood hashing. In *26th annual symposium on foundations of computer science (sfcs 1985)*. IEEE, 281–288.

[12] Hanhua Chen, Liangyi Liao, Hai Jin, and Jie Wu. 2017. The dynamic cuckoo filter. In *2017 IEEE 25th International Conference on Network Protocols (ICNP)*. IEEE, 1–10.

[13] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2017. Monkey: Optimal navigable key-value store. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 79–94.

[14] Niv Dayan, Ioana Bercea, Pedro Reviriego, and Rasmus Pagh. 2023. InfiniFilter: Expanding Filters to Infinity and Beyond. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–27.

[15] Niv Dayan and Stratos Idreos. 2018. Dostoevsky: Better space-time trade-offs for LSM-tree based key-value stores via adaptive removal of superfluous merging. In *Proceedings of the 2018 International Conference on Management of Data*. 505–520.

[16] Niv Dayan and Stratos Idreos. 2019. The log-structured merge-bush & the wacky continuum. In *Proceedings of the 2019 International Conference on Management of Data*. 449–466.

[17] Niv Dayan and Moshe Twitto. 2021. Chucky: A succinct cuckoo filter for lsm-tree. In *Proceedings of the 2021 International Conference on Management of Data*. 365–378.

[18] Niv Dayan, Moshe Twitto, Yuval Rochman, Uri Beitler, Itai Ben Zion, Edward Bortnikov, Shmuel Dashevsky, Ofer Frishman, Evgeni Ginzburg, Igal Maly, et al. 2021. The end of Moore's law and the rise of the data processor. *Proceedings of the VLDB Endowment* 14, 12 (2021), 2932–2944.

[19] Niv Dayan, Tamar Weiss, Shmuel Dashevsky, Michael Pan, Edward Bortnikov, and Moshe Twitto. 2022. Spooky: granulating LSM-tree compactions correctly. *Proceedings of the VLDB Endowment* 15, 11 (2022), 3071–3084.

[20] Peter C Dillinger and Panagiotis Manolios. 2009. Fast, all-purpose state storage. In *Model Checking Software: 16th International SPIN Workshop, Grenoble, France, June 26-28, 2009. Proceedings 16*. Springer, 12–31.

[21] Sourav Dutta, Ankur Narang, and Suman K Bera. 2013. Streaming quotient filter: A near optimal approximate duplicate detection approach for data streams. *Proceedings of the VLDB Endowment* 6, 8 (2013), 589–600.

[22] Agner Fog. 2022. Instruction tables.

[23] Shahabeddin Geravand and Mahmood Ahmadi. 2013. Bloom filter applications in network security: A state-of-the-art survey. *Computer Networks* 57, 18 (2013), 4047–4064.

[24] Bob Goodwin, Michael Hopcroft, Dan Luu, Alex Clemmer, Mihaela Curmei, Sameh Elnikety, and Yuxiong He. 2017. BitFunnel: Revisiting signatures for search. In *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 605–614.

[25] Deke Guo, Jie Wu, Honghui Chen, Ye Yuan, and Xueshan Luo. 2009. The dynamic bloom filters. *IEEE Transactions on Knowledge and Data Engineering* 22, 1 (2009), 120–133.

[26] Hunter McCoy, Steven Hofmeyr, Katherine Yelick, and Prashant Pandey. 2023. High-performance filters for gpus. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 160–173.

[27] Dimitrios Skarlatos, Apostolos Kokolis, Tianyin Xu, and Josep Torrellas. 2020. Elastic cuckoo page tables: Rethinking virtual memory translation for parallelism. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 1093–1108.

[28] Robert E Tarjan and Uri Zwick. 2023. Optimal resizable arrays. In *Symposium on Simplicity in Algorithms (SOSA)*. SIAM, 285–304.

[29] Chris Taylor. 2011. Optimal memory reallocation and the golden ratio. https://archive.li/Z2R8w#selection-83.0-86.0

[30] Robert Williger and Tobias Maier. 2019. *Concurrent dynamic quotient filters: Packing fingerprints into atomics*. Ph. D. Dissertation. Karlsruher Institut für Technologie (KIT).

[31] Alan Wolfe. 2016. Who cares about dynamic array growth strategies? https://blog.demofox.org/2016/05/18/who-cares-about-dynamic-array-growth-strategies/

[32] Yuhan Wu, Jintao He, Shen Yan, Jianyu Wu, Tong Yang, Olivier Ruas, Gong Zhang, and Bin Cui. 2021. Elastic bloom filter: deletable and expandable filter using elastic fingerprints. *IEEE Trans. Comput.* 71, 4 (2021), 984–991.