Instance-Optimized Data Structures for Membership Queries

by

Kapil Eknath Vaidya

B.Tech., IIT Bombay (2018)
S.M., Massachusetts Institute of Technology (2021)

Submitted to the Department of Electrical Engineering
and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2023

© Massachusetts Institute of Technology 2023. All rights reserved.

Author . . . .
Department of Electrical Engineering
and Computer Science
September 15, 2022

Certified by . . . .
Tim Kraska
Associate Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by . . . .
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Instance-Optimized Data Structures for Membership Queries

by

Kapil Eknath Vaidya

Submitted to the Department of Electrical Engineering
and Computer Science
on September 15, 2022, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

We are near the end of Moore's law and hardware growth has hit a stagnation. Modern data processing systems need to continuously improve their performance to match the humongous growth of data. Data structures and algorithms such as sorting, indexes, filters, hash tables, query optimization, etc are the fundamental building blocks of these systems and dictate their performance. Traditional data structures and algorithms provide worst-case guarantees by making no assumptions about the data or workload. Thus, the resulting data processing system gives an adequate performance in the average case but may not be optimal for a particular use case. In this thesis, we will look at how to redesign membership query data structures so they can automatically adapt to an individual use case. These instance-optimized data structures act as drop in replacements for their counterparts in systems and improve their performance without any significant overhaul of the system or labor-intensive manual tuning.

Thesis Supervisor: Tim Kraska
Title: Associate Professor of Electrical Engineering and Computer Science

To my parents

Acknowledgments

First and foremost, I would like to express my sincere gratitude to my advisor, Tim Kraska, for his continuous support and guidance throughout my graduate studies. Tim provided me with great freedom to explore the research directions I was interested in which was crucial for my growth as an independent researcher. Working with him, I learned how to pick impactful problems, how to frame the research properly, and how to communicate it to others.

Next, I would like to thank Michael Mitzenmacher who has been a great collaborator and mentor. Michael offered unceasing help throughout the projects that we collaborated on. I hope to replicate his optimism and enthusiasm about research. I would also like to thank Sam Madden for his invaluable feedback and advice. Sam ensured that the group has good social health by encouraging us to organize various activities.

Also, I am grateful to the students and postdocs with whom I have collaborated. I would especially like to thank Ani Kristo who I co-authored my first paper with. Having him as a collaborator on my first project eased my transition into research. I would also like to thank Anshuman Dutt, Vivek Narasayya, and Surajit Chaudhuri with whom I spent a fruitful summer at Microsoft Research. I would also like to thank my other collaborators over the years: Andreas Kipf, Dominik Horn, Eric Knorr, Ibrahim Sabek, Jialin Ding, Ryan Marcus, Sanchit Misra, Stratos Idreos, Subarna Chatterjee, Umar Farooq Minhas, Ugur Cetintemel and Vikram Nathan.

I have been fortunate to be part of an amazing research group: Anil, Aniruddha, Anna, Brit, Emmanuel, Eugenie, Ferdi, Favyen, Geoffrey, Joana, Laurent, Lei, Markos, Matt, Oscar, Pascal, Raul, Sivaprasad, Tianyu, Wenbo, Xiangyao, Xinjing, Yi, Zeyuan, and Ziniu. I will miss the conversations at our daily lunches, fun hangouts, and weekend trips. Outside of work, I am blessed to have a fantastic set of friends: Aditi, Abhin, Ajay, Arjun, Manish, Parimarjan, Prateesh, Sarath, Siddhartha, and Sohil, who provided happy distractions during the stressful times of my research.

I will be forever indebted to my parents, Eknath Vaidya and Ganga Vaidya, for their unconditional love and selflessness. They have always put my needs before theirs. This thesis would not have been possible without them.

Contents

1 Introduction **17**

1.1 Bloom Filters: Approximate Point Membership Query Structure . . . 20

1.2 Range Filters: Approximate Range Membership Query Structure . . 21

1.3 Hash Tables: Exact Point Membership Query Structure 22

1.4 Discussion 22

2 PLBF: Partitioned Learned Bloom Filter **25**

2.1 Introduction 25

2.2 Background 28

2.2.1 Standard Bloom Filters and Related Variants 29

2.2.2 Learned Bloom Filter 29

2.3 Partitioned Learned Bloom Filter (PLBF) 31

2.3.1 Design 31

2.3.2 General Optimization Formulation 32

2.3.3 Solving the Optimization Problem 33

2.4 Evaluation 38

2.4.1 Overall Performance 39

2.4.2 Performance and the Number of Regions 40

2.4.3 Solving the Relaxed Problem using KKT conditions 41

2.4.4 Optimal False Positive Rate for given thresholds 41

2.4.5 Algorithm for finding thresholds 43

2.4.6 Additional Considerations 44

2.4.7 Sandwiching: A Special Case 44

2.4.8 Performance against number of regions k 45
2.4.9 Performance using various Bloom filter variants 46
2.4.10 Additional Experiments 47
2.4.11 Performance w.r.t standard Bloom filters 47
2.4.12 Performance and Model Quality 47
2.4.13 Discretization Effect on Dynamic Programming Runtime, PLBF Size 48
2.4.14 Construction Time for Various Baselines 49

3 SNARF: Sparse Numerical Array Based Range Filter 51
3.1 Introduction 51
3.2 SNARF: A learned filter 54
3.2.1 SNARF Description 55
3.2.2 Model Details 57
3.2.3 Managing the Bit Array 59
3.3 Analysis 62
3.4 Discussion 66
3.4.1 Key-Query Correlation in Workloads 66
3.4.2 Handling Updates 67
3.5 Related Work 68
3.6 Experimental Evaluation 74
3.6.1 Standalone Analysis 74
3.6.2 RocksDB Experiments 82

4 Learned Hash Tables 89
4.1 Introduction 89
4.2 Traditional Hash Functions 92
4.3 Learned Models as Hash Functions 94
4.3.1 Recursive Model Indexes (RMI) 94
4.3.2 Radix Spline Indexes (RadixSpline) 95
4.3.3 Piece-wise Geometric Model Indexes (PGM) 95

4.4 Perfect Hashing 95
4.4.1 Recursive Splitting (RecSplit) 97
4.4.2 MWHC 97
4.5 Hashing Schemes 98
4.5.1 Bucket Chaining (CHAIN) 98
4.5.2 Open-Addressing 99
4.6 Collisions Analysis for Hashing 101
4.7 Evaluation 105
4.7.1 Experimental Setup 105
4.7.2 Computation Throughput vs Collisions 108
4.7.3 Hash Table Performance 109
4.7.4 More Performance Analysis 115
4.7.5 Range Queries Performance 117
4.7.6 Hash-based Join Performance 119
4.8 Related Work 120

5 Discussion **123**
5.1 Summary 123
5.2 Future Work 124

List of Figures

1-1 Traditional data structures for exact point and range membership queries are hash tables and B-Trees whereas approximate variants are Bloom and range filters. Range data structures B-Trees and range filters have a tree structure whereas hash tables and Bloom filters make use of hash functions. Next, the instance-optimized variants of these data structures are shown which make use of an ML model to capture the data or workload distribution. 19

2-1 (A),(B),(C) represent the original LBF, LBF with sandwiching, and PLBF designs, respectively. Each region in (C) is defined by score boundaries t_i, t_{i+1} and a false positive rate f_i of the Bloom Filter used for that region. (D),(E) show the LBF and PLBF with score space distributions. (F) represents a PLBF design equivalent to the sandwiching approach used in Sec.2.4.7. 28

2-2 FPR vs Space for the (A) Synthetic (B) URLs (C) EMBER datasets for various baselines along with key and non-key score distributions. Space Saved as we increase number of regions for the (D) Synthetic (E) URLs (F) EMBER datasets for PLBF compared to the optimal Bloom filter 39

2-3 (A) represent LBF with sandwiching.(B) represents a PLBF design equivalent to the sandwiching approach. 45

2-4 FPR vs Space for the (A) Synthetic (B) URLs (C) EMBER datasets for various baselines along with key and non-key score distributions. 47

2-5 Space used by various baselines as we increase F1 score for Synthetic dataset 48
2-6 Construction time breakdown for various baselines for the URLs dataset 50

3-1 SNARF Idea: Given a set of keys S, SNARF builds a model $MCDF(x)$ to estimate the empirical cdf of the keys, which it then uses to set corresponding bits in a large bit array B for all $x \in S$. This sparse bit array which encodes key information is then compressed. The model and the compressed bit array are the main parts of SNARF data structure. 53
3-2 SNARF Numerical Model 58
3-3 Golomb coding 60
3-4 Elias Fano Encoding 61
3-5 FPR vs Space Used(in bits per key) by various filters. Each subfigure shows the space-FPR tradeoff for a (A) synthetic (B) real dataset and workload distribution and for a particular range query type (point, range query of size 256 and mixed query workload of size 0,16,64,256). 76
3-6 FPR vs Key-Query Correlation Degree of the queries on uniformly random/wiki keys. With increasing key-query correlation, SNARF and SuRF become worse and Rosetta turns out to be the better filter for very short and highly correlated range queries. 77
3-7 (A) FPR with increasing range query size for fixed space budget. (B) Filter Latency (in ns) against space used (bits per key) (C) Build Time(in millisecs) with increasing number of keys. (D) Filter Throughput as we vary the percentage of updates in the workload. 79
3-8 Filter Query Time (in ns) vs Space used by various filters across various datasets and workloads for mixed queries. 80
3-9 SNARF outperforms other baselines when fully integrated in RocksDB. 83
3-10 Workload Execution time in RocksDB for (A) real world datasets/workloads (B) correlated workloads and (C) varying percentage of empty queries (D) read-write workload (E) varying rnage query sizes 84

4-1 Proportion of collisions with increasing RMI size for uniform randomly and normally distributed keys. 104
4-2 Gap distribution of various datasets 106
4-3 Computation throughput and collisions tradeoffs for various hash functions and using different datasets. 108
4-4 Probe throughput for combinations of 4 hash functions and 3 hashing schemes: (A) bucket chaining, (B) linear probing, and (C) cuckoo hashing. Results are shown for 4 different datasets, and various load factors for each hashing scheme. 110
4-5 Insert throughput for combinations of 4 hash functions and 3 hashing schemes: (A) bucket chaining, (B) linear probing, and (C) cuckoo hashing. Results are shown for 2 different datasets, and various load factors for each scheme. 113
4-6 Performance counters per tuple for the probe experiment in Figure 4-4 using the *gap_10* (first row) and *fb* (second row) datasets at load factor 80%. 114
4-7 Effect of (1) gap distribution on *LMH* collisions (left), and (2) dataset size on building time (right). 115
4-8 Effect of increasing the bucket capacity on the probe throughput of a chained hash table at load factor of 50%. 116
4-9 Effect of both point queries percentage (first row), and range query size (second row) on the point/range queries throughput. 117
4-10 Runtime breakdown for the different implementations of non-partitioned hash join (NPJ) using various combinations of hashing functions and schemes. 119

List of Tables

2.1 DP runtime and space used by PLBF as we increase the discretization N in the URLs dataset 49
2.2 DP runtime and space used by PLBF as we increase the discretization N in the EMBER dataset 49

4.1 Default numbers of submodels in LMH functions. 107

Chapter 1

Introduction

Modern data processing systems need to continuously improve their performance to match the humongous growth of data. These systems cannot rely only on hardware for performance improvements as we are near the end of Moore's law and hardware growth has hit stagnation. One way to improve the performance of a system is to specialize it for a particular application. Specialization as a technique has been prevalent in the data systems community with systems like C-store [148], Streambase[151] built for analytical and streaming applications, respectively.

But one can find a diverse set of use cases even within a particular application. For example, certain analytical use cases might be concerned with identifying recent trends while others might want historical summaries of the data. These "instances" have the same higher-level goal (e.g., performing analytics on the data) but operate on different datasets or have different workload patterns.

In order to specialize for these instances, a recent trend in the research community is to build "instance-optimized" systems which are made up of components that can improve their performance by taking the data or workload distribution into account. An important class of components that are being "instance-optimized" are data structures and algorithms. Data structures and algorithms such as sorting, indexes, filters, hash tables, query optimization, etc are the fundamental building blocks of data processing systems and dictate their performance. Traditional data structures and algorithms provide worst-case guarantees by making no assumptions about the

data or workload. Thus, the resulting system gives an adequate performance in the average case but may not be optimal for a particular use case. By using data structures and algorithms that optimize for a particular instance, the system can achieve better performance for that particular instance. These instance-optimized data structures/algorithms act as drop-in replacements for their counterparts in systems and improve their performance without any significant overhaul of the system design. Such instance-optimized components automatically adapt to an instance without labor-intensive manual tuning and thus, save a huge amount of operational cost.

There have been various efforts in building instance-optimized components [88, 48, 45, 46]. A common theme in this redesign is the use of learned models that learn the data or workload distribution. The primary work that influenced this direction was Learned Indexes [86], which redesigned range indexes. Range Indexes store key-value pairs and support range queries which are queries that ask for values of keys in a certain range. The work on Learned Indexes [86] observed that B-Trees can be thought of as models. These models predict the position of the key-value pair within a sorted array, given the key value. Learned Indexes then propose using ML models to learn this mapping instead of a B-Tree. This helps them to be optimized for specific data distribution and empirically outperform B-Trees.

Range Index data structures fall under the broader umbrella of membership query data structures which are quite commonly used in data systems for data skipping, joins, aggregates, scans, data retrieval, etc. Membership queries check for the existence of keys with certain properties in the data. These membership queries might be for a single data point or a range of data points. Traditional data structures for point and range queries are hash tables and B-Trees, respectively. Some applications can tolerate approximate answers for better speed/memory usage, hence approximate variants of these data structures exist which are known as filters. Traditional data structures for approximate point and range queries are Bloom and range filters. These data structures are shown in Fig.1-1. Instance optimized variant of B-Trees was already proposed in the paper [86]. In this thesis, we design instance-optimized variants of the remaining three data structures by making use of ML models. Below we briefly

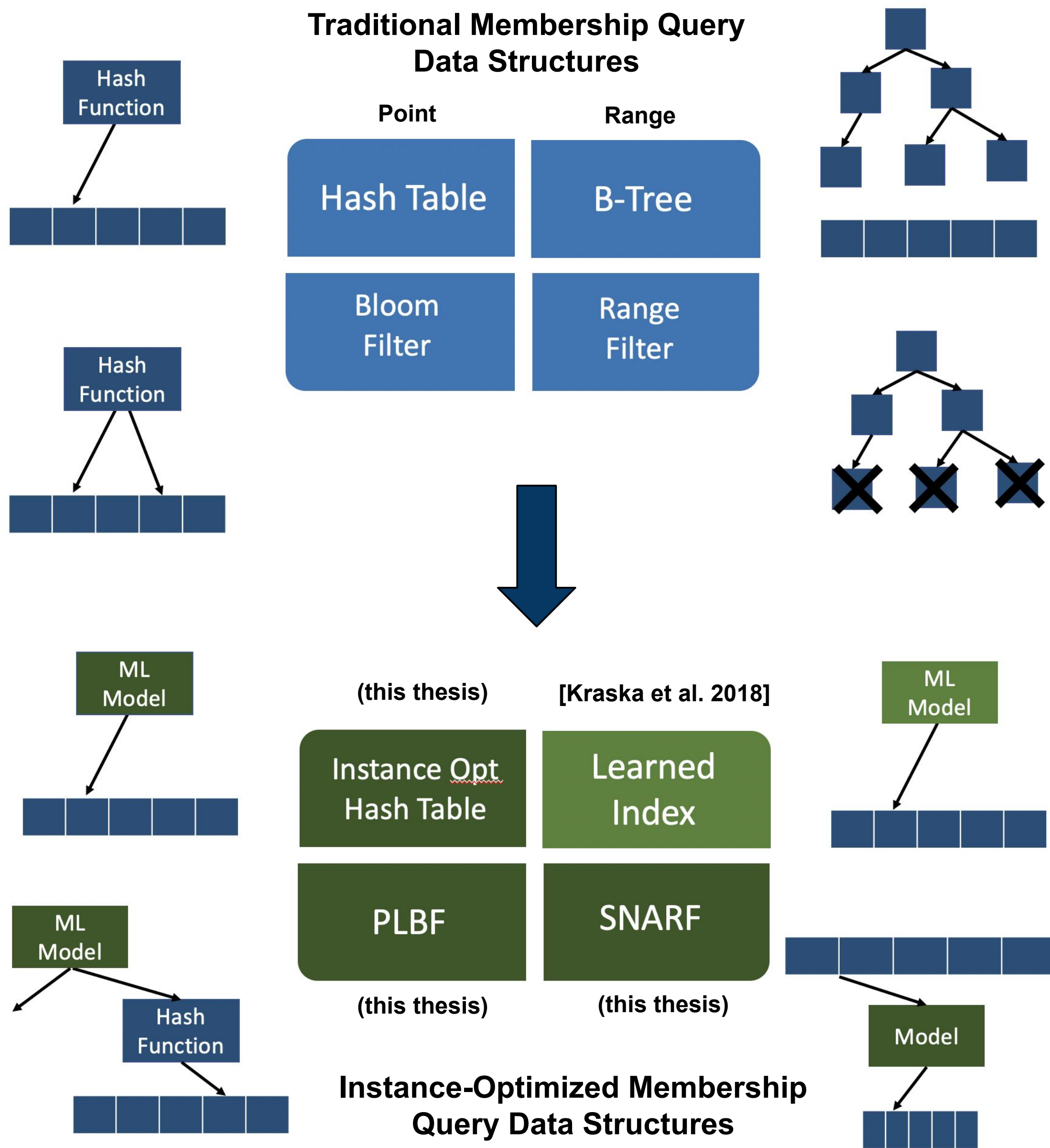


Figure 1-1: Traditional data structures for exact point and range membership queries are hash tables and B-Trees whereas approximate variants are Bloom and range filters. Range data structures B-Trees and range filters have a tree structure whereas hash tables and Bloom filters make use of hash functions. Next, the instance-optimized variants of these data structures are shown which make use of an ML model to capture the data or workload distribution.

discuss the ideas behind the design of these structures.

1.1 Bloom Filters: Approximate Point Membership Query Structure

Bloom filters are space-efficient data structures that are used to approximately answer point membership queries on a set S. The queries filters support are of the form: "Is x in the set S?". Bloom filters only have false positives and no false negatives. A false positive is a case where $x \notin S$ but the filter returns $x \in S$, and a false negative is a case where $x \in S$ but the filter says $x \notin S$. For applications where fetching data requires an expensive operation like disk I/O or S3 request, Bloom filters can help prune empty requests. These empty requests don't have data corresponding to them on the disk and lead to unnecessary expensive data fetches. No false negatives in Bloom filters mean that queries having data corresponding to them are always accepted and most empty queries are rejected, thus maintaining consistency of the results. The main trade-off in filters is between the space consumed by the filter and the false positive rate (FPR) provided by the filter.

The key idea of instance-optimized Bloom filters is that in many practical settings, given a query input, the likelihood that the input is in the set S can be deduced by some observable features which can be captured by a machine learning model. For example, a Bloom filter that represents a set of malicious URLs can benefit from a learned model that can distinguish malicious URLs from benign URLs. This model can be trained on URL features such as length of hostname, counts of special characters,etc. Previous learned Bloom filter approaches did not utilize the model efficiently due to their sub-optimal design and parameter tuning. Therefore, we propose Partitioned Learned Bloom Filter (**PLBF**) [156], an instance-optimized Bloom filter with a general design that automatically tunes its design parameters to optimal values. PLBF frames the problem of trading off space consumed by the filter and false positive rate achieved as an optimization problem in terms of the design parameters. The solution to this

optimization problem provides the optimal values for the parameters.

1.2 Range Filters: Approximate Range Membership Query Structure

Range filters are space-efficient data structures that are used to approximately answer range membership queries on a set S. The filters support queries of the form: ”Is there x in the set S between [A,B]?”. Similar to Bloom filters, they do not allow any false negatives and the main trade-off in filters is between the space consumed by the filter and the false positive rate (FPR) provided by the filter.

LSM-based key-value stores such as RocksDB, LevelDB in industry increasingly serve as the backbone of applications across the areas of social media, stream and log processing, file structures, and databases for geo-spatial coordinates, time-series, and graphs. LSM's store data in multiple immutable files on a disk. Retrieving a particular item or set of items in a particular range leads to multiple expensive I/O's to look up the items in these immutable files. In many settings, the item may not be present (or the set of items may have no item present) in the files, leading to unnecessary I/O's that degrade read performance.

Range filters residing in faster memory can help in this situation: if a query has no corresponding item, the filter most likely returns false and saves expensive I/O. We proposed Sparse Numerical Array-Based Range Filters (**SNARF**)[155], an instance-optimized range filter that supports both point and range queries for numerical data. SNARF uses a model of the data distribution to map the keys into a bit array which is stored in a compressed form. SNARF when integrated with RocksDB provides upto 10x better read performance compared to state-of-art filters.

1.3 Hash Tables: Exact Point Membership Query Structure

Hash tables are a fundamental data structure in database management, playing an important role in indexing and joins. The idea behind hash tables is to place keys in distinct slots using a mapping function. Retrieval of a key involves using the mapping function to identify the corresponding slot. Hash tables rely on traditional hash functions as the mapping function. Traditional hash functions aim to mimic a function that maps a key to a random value, which can result in collisions, where multiple keys are mapped to the same value. There are many well-known schemes like chaining, probing, and cuckoo hashing to handle collisions.

Another approach to build hash indexes is to use *perfect* hash functions instead of truly random hash functions. Perfect hash functions have no collisions; however, they must be specially constructed for a given data set, and have other costs in storage and computation time.

Along the lines of instance-optimized components we studied if by leveraging the data distribution we can build instance optimized hash functions and whether such a reduction translates to improved performance, particularly for hash table applications: indexing and joins. We showed that by leveraging the data distribution one can reduce collisions in some cases, which depend on how the data is distributed. These cases can be identified beforehand and instance optimized hash functions can be used for them.

1.4 Discussion

The work in this thesis demonstrates that instance-optimized components can give huge performance boosts, however more work remains to make them practical and robust for production-level deployment. We will discuss these points in detail in Section.5 and here we highlight some key points.

PLBF showed that by using utilizing the model efficiently one can get huge benefits. However, one of the major drawbacks is high query latency. In PLBF, the model gets

queried before the backup filters and thus, the query latency can be higher than that of a traditional Bloom filter. PLBF optimizes its design for a particular workload but the workload distribution might shift over time resulting in poor performance if the shifts are drastic.

SNARF improves the performance of range filters compared to previous approaches by using a static model of the data distribution. SNARF supports updates but might not be robust in scenarios where updates might change the distribution of the data. Making SNARF robust to these data shifts is still an open problem. Another interesting direction would be to make SNARF workload dependent as it currently only relies on the knowledge of data distribution.

Instance-optimized hash tables can provide performance improvements in certain scenarios. The models we primarily focused on were piece-wise linear models but complex models like NN, SVM, and decision trees need to be considered. The piece-wise linear models we focused on were limited to numerical keys, using models supporting strings to build hash tables is still unexplored.

A common drawback among all three works is handling data/workload shifts that arise from the nature of instance optimization. Once a component optimizes for a particular data/workload distribution, it provides good performance for that instance but might not perform well under shifts. Instance-optimized components need mechanisms to identify these shifts and lightweight techniques to adapt to the new instance.

Chapter 2

PLBF: Partitioned Learned Bloom Filter

2.1 Introduction

Bloom filters are space-efficient probabilistic data structures that are used to test whether an element is a member of a set [[16]]. A Bloom filter compresses a given set S into an array of bits. A Bloom filter may allow false positives, but will not give false negative matches, which makes them suitable for numerous memory-constrained applications in networks, databases, and other systems areas. Indeed, there are many thousands of papers describing applications of Bloom filters [[35], [41], [22]].

There exists a trade off between the false positive rate and the size of a Bloom filter (smaller false positive rate leads to larger Bloom filters). For a given false positive rate, there are known theoretical lower bounds on the space used [[123]] by the Bloom filter. However, these lower bounds assume the Bloom filter could store any possible set. If the data set or the membership queries have specific structure, it may be possible to beat the lower bounds in practice [[110], [23], [113]]. In particular, [[84]] and [[111]] propose using machine learning models to reduce the space further, by using a learned model to provide a suitable pre-filter for the membership queries. This allows one to beat the space lower bounds by leveraging the context specific information present in the learned model. [135] propose a neural Bloom Filter that learns to write to memory

using a distributed write scheme and achieves compression gains over the classical Bloom filter.

The key idea of learned Bloom filters is that in many practical settings, given a query input, the likelihood that the input is in the set S can be deduced by some observable features which can be captured by a machine learning model. For example, a Bloom filter that represents a set of malicious URLs can benefit from a learned model that can distinguish malicious URLs from benign URLs. This model can be trained on URL features such as length of hostname, counts of special characters, etc. This approach is described in [[84]], which studies how standard index structures can be improved using machine learning models; we refer to their framework as the original learned Bloom filter, Given an input x and its features, the model outputs a score $s(x)$ which is supposed to correlate with the likelihood of the input being in the set. Thus, the elements of the set, or keys, should have a higher score value compared to non-keys. This model is used as a pre-filter, so when score $s(x)$ of an input x is above a pre-determined threshold t, it is directly classified as being in the set. For inputs where $s(x) < t$, a smaller backup Bloom filter built from only keys with a score below the threshold (which are known) is used. This maintains the property that there are no false negatives. The design essentially uses the model to immediately answer for inputs with high score whereas the rest of the inputs are handled by the backup Bloom filter as shown in Fig.2-1(A). The threshold value t is used to partition the space of scores into two regions, with inputs being processed differently depending on in which region its score falls. With a sufficiently accurate model, the size of the backup Bloom filter can be reduced significantly over the size of a standard Bloom filter while maintaining overall accuracy. [[84]] showed that, in some applications, even after taking the size of the model into account, the learned Bloom filter can be smaller than the standard Bloom filter for the same false positive rate.

The original learned Bloom filter compares the model score against a single threshold, but the framework has several drawbacks.

Choosing the right threshold: The choice of threshold value for the learned Bloom filter is critical, but the original design uses heuristics to determine the threshold

value.

Using more partitions: Comparing the score value only against a single threshold value wastes information provided by the learning model. For instance, two elements x_1, x_2 with $s(x_1) >> s(x_2) > t$, are treated the same way but the odds of x_1 being a key are much higher than for x_2. Intuitively, we should be able to do better by partitioning the score space into more than two regions.

Optimal Bloom filters for each region: Elements with scores above the threshold are directly accepted as keys. A more general design would provide backup Bloom filters in both regions and choose the Bloom filter false positive rate of each region so as to optimize the space/false positive trade-off as desired. The original setup can be interpreted as using a Bloom filter of size 0 and false positive rate of 1 above the threshold. This may not be the optimal choice; moreover, as we show, using different Bloom filters for each region(as shown in Fig.2-1(C)) allows further gains when we increase the number of partitions.

Follow-up work by [[111]] and [[34]] improve on the original design but only address a subset of these drawbacks. In particular, [[111]] proposes using Bloom filters for both regions and provides a method to find the optimal false positive rates for each Bloom filter. But [[111]] only considers two regions and does not consider how to find the optimal threshold value. [[34]] propose using multiple thresholds to divide the space of scores into multiple regions, with a different backup Bloom filter for each score region. The false positive rates for each of the backup Bloom filters and the threshold values are chosen using heuristics. Empirically, we found that these heuristics might perform worse than [[111]] in some scenarios.

A general design that resolves all the drawbacks would, given a target false positive rate and the learned model, partition the score space into multiple regions with separate backup Bloom filters for each region, and find the optimal threshold values and false positive rates, under the goal of minimizing the memory usage while achieving the desired false positive rate as shown in Fig.2-1(C). In this work, we show how to frame this problem as an optimization problem, and show that our resulting solution significantly outperforms the heuristics used in previous works. Additionally, we show

that our maximum space saving[1] is linearly proportional to the KL divergence of the key and non-key score distributions determined by the partitions. We present a dynamic programming algorithm to find the optimal parameters (up to the discretization used for the dynamic programming) and demonstrate performance improvements over a synthetic dataset and two real world datasets: URLs and EMBER. We also show that the performance of the learned Bloom filter improves with increasing number of partitions and that in practice a small number of regions ($\approx 4-6$) suffices to get a very good performance. We refer to our approach as a *partitioned learned Bloom filter* (PLBF). Experimental results from both simulated and real-world datasets show significant performance improvements. We show that to achieve a false positive rate of 0.001, [[111]] uses 8.8x, 3.3x and 1.2x the amount of space and [[34]] uses 6x, 2.5x and 1.1x the amount of space compared to PLBF for synthetic, URLs and EMBER respectively.

2.2 Background

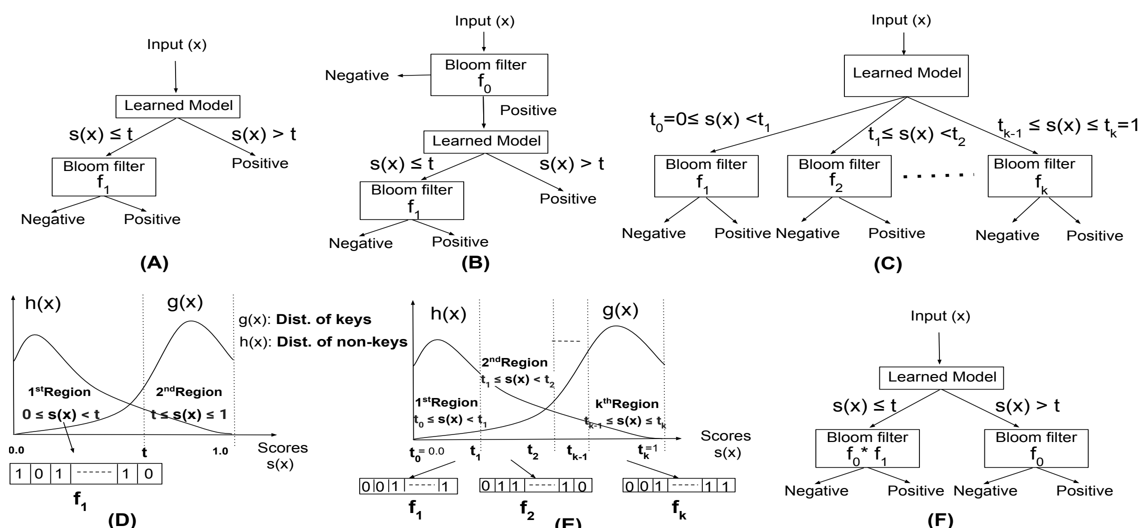


Figure 2-1: (A),(B),(C) represent the original LBF, LBF with sandwiching, and PLBF designs, respectively. Each region in (C) is defined by score boundaries t_i, t_{i+1} and a false positive rate f_i of the Bloom Filter used for that region. (D),(E) show the LBF and PLBF with score space distributions. (F) represents a PLBF design equivalent to the sandwiching approach used in Sec.2.4.7.

[1] space saved by using our approach instead of a Bloom filter

2.2.1 Standard Bloom Filters and Related Variants

A standard Bloom filter, as described in Bloom's original paper [[16]], is for a set $S = \{x_1, x_2, ..., x_n\}$ of n keys. It consists of an array of m bits and uses k independent hash functions $\{h_1, h_2, ...h_k\}$ with the range of each h_i being integer values between 0 and $m - 1$. We assume the hash functions are fully random. Initially all m bits are 0. For every key $x \in S$, array bits $h_i(x)$ are set to 1 for all $i \in \{1, 2, ...k\}$.

A membership query for y returns that $y \in S$ if $h_i(y) = 1$ for all $i \in \{1, 2, ...k\}$ and $y \notin S$ otherwise. This ensures that the Bloom filter has no false negatives but non-keys y might result in a false positive. This false positive rate depends on the space m used by the Bloom Filter. Asymptotically (for large m, n with m/n held constant), the false positive rate is given by

$$\left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k. \tag{2.1}$$

See [[22, 18]] for further details.

[[16]] proved a space lower bound of $|S| \times \log_2(\frac{1}{F})$ for a Bloom filter with false positive rate F. The standard construction uses space that is asymptotically $\log_2 e (\approx 1.44)$ times more than the lower bound. Other constructions exist, such as Cuckoo filters[[54]], Morton filters[[21]], XOR filters[[67]] and Vacuum filters[[159]]. These variants achieve slightly better space performance compared to standard Bloom filters but still are a constant factor larger than the lower bound. [[123]] presents a Bloom filter design that achieves this space lower bound, but it appears too complicated to use in practice.

2.2.2 Learned Bloom Filter

Learned Bloom filters make use of learned models to beat the theoretical space bounds. Given a learned model that can distinguish between keys and non-keys, learned Bloom filters use it as a pre-filter before using backup Bloom filters. The backup Bloom filters can be any variant including the standard, cuckoo, XOR filters, etc. If the size of the model is sufficiently small, learned models can be used to enhance the performance of

any Bloom filter variant.

We provide the framework for learned Bloom filters. We are given a set of keys $S = \{x_1, x_2, .., x_n\}$ from a universe U for which to build a Bloom filter. We are also given a sample of the non-keys Q which is representative of the set $U - S$. Features that can help in determining if an element is a member of S are determined. The learned model is then trained on features of set $S \cup Q$ for a binary classification task and produces a score $s(x) \in [0, 1]$. This score $s(x)$ can be viewed (intuitively, not formally) as the confidence of the model that the element x is in the set S. So, a key in S would ideally have a higher score value than the non-keys. An assumption in this framework is that the training sample distribution needs to match or be close to the test distribution of non-keys; the importance of this assumptions has been discussed at length in [[111]]. For many applications, past workloads or historical data can be used to get an appropriate non-key sample.

As discussed above, [[84]] set a threshold t and inputs satisfying $s(x) > t$ are classified as a key. A backup Bloom filter is built for just the keys in S satisfying $s(x) \leq t$. This design is represented in Fig.2-1(A). [[111]] proposes using another Bloom filter *before* the learned model along with a backup Bloom Filter. As the learned model is used between two Bloom filters as shown in Fig.2-1(B), this is referred to as the 'sandwiching' approach. They also provide the analysis of the optimal false positive rates for a given amount of memory for the two Bloom filters (given the false negative rate and false positive rate for the learned model, and the corresponding threshold). Interestingly, the sandwiching approach and analysis can be seen as a special case of our approach and analysis, as we describe later in Sec.2.4.7. [[34]] use multiple thresholds to partition the score space into multiple regions and use a backup Bloom filter for each score region. They propose heuristics for how to divide up the score range and choose false positive rate per region.

2.3 Partitioned Learned Bloom Filter (PLBF)

2.3.1 Design

As discussed before, the general design segments the score space into multiple regions using multiple thresholds, as shown in Fig.2-1(C), and uses separate backup Bloom filters for each region. We can choose different target false positive rates for each region[2]. The parameters associated with each region are its threshold boundaries and its false positive rate. Setting good values for these parameters is crucial for performance. Our aim is to analyze the performance of the learned Bloom filter with respect to these parameters, and find methods to determine optimal or near-optimal parameters.

The following notation will be important for our analysis. Let $G(t)$ be the fraction of keys with scores falling below t. We note that since the key set is finite, $G(t)$ goes through discrete jumps. But it is helpful (particularly in our pictures) to think of $G(t)$ as being a continuous function, corresponding to a cumulative probability distribution, with a corresponding "density" function $g(t)$. For non keys, we assume that queries involving non-keys come from some distribution $\mathcal{D}$, and we define $H(t)$ to be probability that a non-key query from $\mathcal{D}$ has a score less than or equal to t. Note that non key query distribution might be different from non key distribution. If non key queries are chosen uniformly at random, non key query distribution would be the same as non key distribution. We assume that $H(t)$ is known in the theoretical analysis below. In practice, we expect a good approximation of $H(t)$ will be used, determined by taking samples from $\mathcal{D}$ or a suitably good approximation, which may be based on, for example, historical data (discussed in detail in [[111]]). Here $H(t)$ can be viewed as a cumulative distribution function, and again in our pictures we think of it as having a density $h(t)$. Also, note that if queries for non-keys are simply chosen uniformly at random, then $H(t)$ is just the fraction of non-keys with scores below t. While our analysis holds generally, the example of $H(t)$ being the fraction

[2]The different false positive rates per region can be achieved in multiple ways. Either by choosing a separate Bloom filter per region or by having a common Bloom filter with varying number of hash functions per region.

of non-keys with scores below t may be easier to keep in mind. Visualization of the original learned Bloom filter in terms of these distributions is shown in Fig.2-1(D).

As we describe further below, for our partitioned learned Bloom filter, we use multiple thresholds and a separate backup Bloom filter for each region, as show in Fig.2-1(E). In what follows, we formulate the problem of choosing thresholds and backup Bloom filter false positive rates (or equivalently, sizes) as an optimization problem in section 2.3.2. In section 2.3.3, we find the optimal solution of a relaxed problem which helps us gain some insight into the general problem. We then propose an approximate solution for the general problem in section 2.3.3.

We find in our formulation that the resulting parameters correspond to quite natural quantities in terms of G and H. Specifically, the optimal false positive rate of a region is proportional to the ratio of the fraction of keys to the fraction of non-keys in that region. If we think of these region-based fractions for keys and non-keys as probability distributions, the maximum space saving obtained is proportional to the KL divergence between these distributions. Hence we can optimize the thresholds by choosing them to maximize this divergence. We show that we can find thresholds to maximize this divergence, approximately, through dynamic programming. We also show that, naturally, this KL divergence increases with more number of regions and so does the performance. In our experiments, we find a small number($\approx 4-6$) of partitions suffices to get good performance.

2.3.2 General Optimization Formulation

To formulate the overall problem as an optimization problem, we consider the variant which minimizes the space used by the Bloom filters in PLBF in order to achieve an overall a target false positive rate (F). We could have similarly framed it as minimizing the false positive rate given a fixed amount of space. Here we are assuming the learned model is given.

We assume normalized score values in $[0, 1]$ for convenience. We have region boundaries given by t_i values $0 = t_0 \leq t_1 \leq ....t_{k-1} \leq t_k = 1$, with score values between $[t_{i-1}, t_i]$ falling into the ith region. We assume the target number of regions k

is given. We denote the false positive rate for the Bloom filter in the ith region by f_i. We let G and H be defined as above. As state previously, Fig.2-1(E) corresponds to this setting, and the following optimization problem finds the optimal thresholds t_i and the false positive rates f_i:

$$\min_{t_i, f_i} \left(\sum_{i=1}^{k} |S| \times (G(t_i) - G(t_{i-1})) \times c \log_2 \left(\frac{1}{f_i} \right) \right) + \text{Size of Learned Model} \tag{2.2}$$

$$\text{constraints} \quad \sum_{i=1}^{k} \left(H\left(t_i\right) - H(t_{i-1}) \right) \times f_i \leq F \tag{2.3}$$

$$f_i \leq 1 \quad , i = 1...k \tag{2.4}$$

$$(t_i - t_{i-1}) \geq 0 \quad , i = 1...k \ ; \ t_0 = 0; t_k = 1 \tag{2.5}$$

The minimized term (Eq.2.2) represents the total size of the learned Bloom filter, the size of backup Bloom filters is obtained by summing the individual backup Bloom filter sizes. The constant c in the equation depends on which variant of the Bloom filter is used as the backup[3]; as it happens, its value will not affect the optimization.

The first constraint (Eq.2.3) ensures that the overall false positive rate stays below the target F. The overall false positive rate is obtained by summing the appropriately weighted rates of each region. The next constraint (Eq.2.4) encodes the constraint that false positive rate for each region is at most 1. The last set of constraints (Eq.2.5) ensure threshold values are increasing and cover the interval $[0, 1]$.

2.3.3 Solving the Optimization Problem

Solving a Relaxed Problem

If we remove the false positive rate constraints (Eq.2.4, giving $f_i \leq 1$), we obtain a relaxed problem shown in Eq.2.6. This relaxation is useful because it allows us to use

[3]The sizes of Bloom filter variants are proportional to $|S| \times \log_2(1/f)$, where S is the set it represents, and f is the false positive rate it achieves. See e.g. [[111]] for related discussion. The constant c depends on which type of Bloom filter is used as a backup. For example, $c = \log_2(e)$ for standard Bloom filter.

the Karush-Kuhn-Tucker (KKT) conditions to obtain optimal f_i values in terms of the t_i values, which we used to design algorithms for finding near-optimal solutions. Throughout this section, we assume the the relaxed problem yields a solution for the original problem; we return to this issue in subsection 2.3.3.

$$\min_{t_{i=1\ldots k-1}, f_{i=1\ldots k}} \quad \left(\sum_{i=1}^{k} |S| \times (G(t_i) - G(t_{i-1})) \times c \log_2\left(\frac{1}{f_i}\right)\right) + \text{Size of Learned Model}$$

$$\begin{aligned} \text{constraints} \qquad & \sum_{i=1}^{k} (H(t_i) - H(t_{i-1})) \times f_i \leq F; \\ & (t_i - t_{i-1}) \geq 0 \quad, i = 1\ldots k;\ t_0 = 0;\ t_k = 1 \end{aligned} \tag{2.6}$$

The optimal f_i values obtained by using the KKT conditions yield Eq.2.7 (as derived in Sec.2.4.3), giving the exact solution in terms of t_i's.

$$f_i = F\frac{G(t_i)-G(t_{i-1})}{H(t_i)-H(t_{i-1})} \tag{2.7}$$

The numerator $G(t_i) - G(t_{i-1})$ is the fraction of keys in the ith region and the denominator $H(t_i) - H(t_{i-1})$ is the probability of a non-key query being in the ith region. In intuitive terms, the false positive rate for a region is proportional to the ratio of the key density (fraction of keys) to non-key density (fraction of non-key queries). Since we have found the optimal f_i in terms of the t_i, we can replace the f_i in the original problem to obtain a problem only in terms of the t_i. In what follows, we use $g(\hat{\mathbf{t}})$ to represent the discrete distribution given by the k values of $G(t_i) - G(t_{i-1})$ for $i = 1, \ldots, k$, and similarly we use $h(\hat{\mathbf{t}})$ for the distribution corresponding to the $H(t_i) - H(t_{i-1})$ values. Eq.2.8 shows the rearrangement of the minimization term(excluding model size) after substitution.

$$\begin{aligned} \text{Min. Term} &= \sum_{i=1}^{k} |S| \times (G(t_i) - G(t_{i-1})) \times c \log_2\left(\frac{H(t_i) - H(t_{i-1})}{(G(t_i) - G(t_{i-1})) \times F}\right) \\ &= \sum_{i=1}^{k} |S| \times (G(t_i) - G(t_{i-1})) \times c \log_2\left(\frac{1}{F}\right) - c \times |S| \times D_{KL}\left(g(\hat{\mathbf{t}}), h(\hat{\mathbf{t}})\right) \end{aligned} \tag{2.8}$$

where D_{KL} is the standard KL divergence for the distributions given by $g(\hat{\mathbf{t}})$ and

$h(\hat{\mathbf{t}})$.

Eq.2.8 represents the space occupied by the backup Bloom filters; the total space includes this and the space occupied by the learned model.

$$c \times \left(|S| \times \log_2\left(\frac{1}{F}\right) - |S| \times D_{KL}\left(g(\hat{\mathbf{t}}), h(\hat{\mathbf{t}})\right)\right) + \text{Size Of Learned Model} \qquad (2.9)$$

The space occupied by the Bloom filter without the learned model is $c \times |S| \times \log_2(1/F)$. Thus, the space saved by PLBF in comparison to the normal Bloom filter is:

$$c \times \left(|S| \times D_{KL}\left(g(\hat{\mathbf{t}}), h(\hat{\mathbf{t}})\right)\right) - \text{Size Of Learned Model} \qquad (2.10)$$

The space saved by PLBF is therefore linearly proportional to the KL divergence of key and non-key distributions of the regions given by $g(\hat{\mathbf{t}})$ and $h(\hat{\mathbf{t}})$ of the regions.

This derivation suggests that the KL divergence might also be used as a loss function to improve the model quality. We have tested this empirically, but thus far have not seen significant improvements over the MSE loss we use in our experiments; this remains an interesting issue for future work.

Finding the Optimal Thresholds for Relaxed Problem

We have shown that, given a set of thresholds, we can find the optimal false positive rates for the relaxed problem. Here we turn to the question of finding optimal thresholds. We assume again that we are given k, the number of regions desired. (We consider the importance of choosing k further in our experimental section.) Given our results above, the optimal thresholds correspond to the points that maximize the KL divergence between $(g(\hat{\mathbf{t}}), h(\hat{\mathbf{t}}))$. The KL divergence of $(g(\hat{\mathbf{t}}), h(\hat{\mathbf{t}}))$ is the sum of the terms $g_i \log_2 \frac{g_i}{h_i}$, one term per region. (Here $g_i = G(t_i) - G(t_{i-1})$ and $h_i = H(t_i) - H(t_{i-1})$.) Note that each term depends only on the proportion of keys and non-keys in that region and is otherwise independent of the other regions. This property allows a recursive definition of KL divergence that is suitable for dynamic programming.

We divide the score space $[0, 1]$ into N consecutive small segments for a chosen value

of N; this provides us a discretization of the score space, with larger N more closely approximating the real interval. Given k, we can find a set of k approximately optimal thresholds using dynamic programming, where the solution is approximate due to our discretization of the score space. Let $DP_{KL}(n,j)$ denote the maximum divergence one can get when you divide the first n segments into j regions. Our approximately optimal divergence corresponds to $DP_{KL}(N,k)$. The idea behind the algorithm is that the we can recursively define $DP_{KL}(n,j)$ as represented in Eq.2.11. Here g', h' represent the fraction of keys and the fraction of non-key queries, respectively, in these N segments.

$$DP_{KL}\left(n,j\right) = \max\left(DP_{KL}(n-i,j-1) + \left(\sum_{r=i}^{n} g'(r) \times \log_2\left(\frac{\sum_{r=i}^{n} g'(r)}{\sum_{r=i}^{n} h'(r)}\right)\right)\right) \tag{2.11}$$

The time complexity of computing $DP_{KL}(N,k)$ is $\mathcal{O}(N^2k)$. One can increase the value of N to get more precision in the discretization when finding thresholds, at the cost of higher computation time.

The Relaxed Problem and the General Problem

We can find a near-optimal solution to the relaxed problem by first, obtaining the threshold values that maximize the divergence and then, getting the optimal f_i values using Eq.2.7. In many cases, the optimal relaxed solution will also be the optimal general solution, specifically if $F \times (G(t_{i-1}) - G(t_i))/(H(t_{i-1}) - H(t_i)) < 1$ for all i. Hence, if we are aiming for a sufficiently low false positive rate F, solving the relaxed problem suffices.

To solve the general problem, we need to deal with regions where $f_i \geq 1$, but we can use the relaxed problem as a subroutine. First, given a fixed set of t_i values for the general problem, we have an algorithm (Alg.1, as discussed in Sec.2.4.4) to find the optimal f_i's. Briefly summarized, we solve the relaxed problem, and for regions with $f_i > 1$, the algorithm sets $f_i = 1$, and then re-solves the relaxed problem with these additional constraints, and does this iteratively until no region with $f_i > 1$ remains. The problem is that we do not have the optimal set of t_i values to begin; as such, we

use the optimal t_i values for the relaxed solution as described in Section 2.3.3. This yields a solution to the general problem (psuedo-code in Alg.2), but we emphasize that it is not optimal in general, since we did not start with the optimal t_i. We expect still that it will perform very well in most cases.

In practice, we observe that keys are more concentrated on higher scores, and non-key queries are more concentrated on lower scores. Given this property, if a region with $f_i = 1$ (no backup Bloom filter used) exists in the optimal solution of the general problem, it will most probably be the rightmost region. In particular, if $(G(t_{i-1}) - G(t_i))/(H(t_{i-1}) - H(t_i))$ is increasing as t_{i-1}, t_i increase – that is, the ratio of the fraction of keys to the fraction of non-key queries over regions is increasing – then indeed without loss of generality the last (kth) region will be the only one with $f_k = 1$. (We say only one region because any two consecutive regions with $f_i = 1$ can be merged and an extra region can be made in the remaining space which is strictly better, as adding an extra region always helps as shown in Sec.2.4.8.) It is reasonable to believe that in practice this ratio will be increasing or nearly so.

Hence if we make the assumption that in the optimal solution all the regions except the last satisfy the $f_i < 1$ constraint, then if we identify the optimal last region's boundary, we can remove the $f_i \leq 1$ constraints for $i \neq k$ and apply the DP algorithm to find near optimal t_i's. To identify the optimal last region's boundary, we simply try all possible boundaries for the kth region (details discussed in Sec.2.4.5). As it involves assumptions on the behavior of G and H, we emphasize again that this will not guarantee finding the optimal solution. But when the conditions are met it will lead to a near-optimal solution (only near-optimal due to the discretization of the dynamic program).

2.4 Evaluation

We compare PLBF against the theoretically optimal Bloom filter [[16]][4], the sandwiching approach [[111]], and AdaBF [[34]]. Comparisons against standard Bloom filters[5] appear in Sec.2.4.11. We excluded the original learned Bloom filter [[84]] as the sandwiching approach was strictly better. We include the size of the learned model with the size of the learned Bloom filter. To ensure a fair comparison, we used the optimal Bloom filter as the backup bloom filter for all learned variants. We use 3 different datasets:

URLs: As in previous papers [[84], [34]], we used the URL data set, which contains 103520 (23%) malicious and 346646 (77%) are benign URLs. We used 17 features from these URL's such as host name length, use of shortening, counts of special characters,etc.

EMBER: Bloom filters are widely used to match file signatures with the virus signature database. Ember (Endgame Malware Benchmark for Research) [[4]] is an open source collection of 1.1M sha256 file hashes that were scanned by VirusTotal in 2017. Out of the 1.1 million files, 400K are malicious, 400K are benign, and we ignore the remaining 300K unlabeled files. The features of the files are already included in the dataset.

Synthetic: An appealing scenario for our method is when the key density increases and non-key density decreases monotonically with respect to the score value. We simulate this by generating the key and non-key score distribution using Zipfian distributions as in Fig.2-2(A). Since we directly work on the score distribution, the size of the learned model for this synthetic dataset is zero.

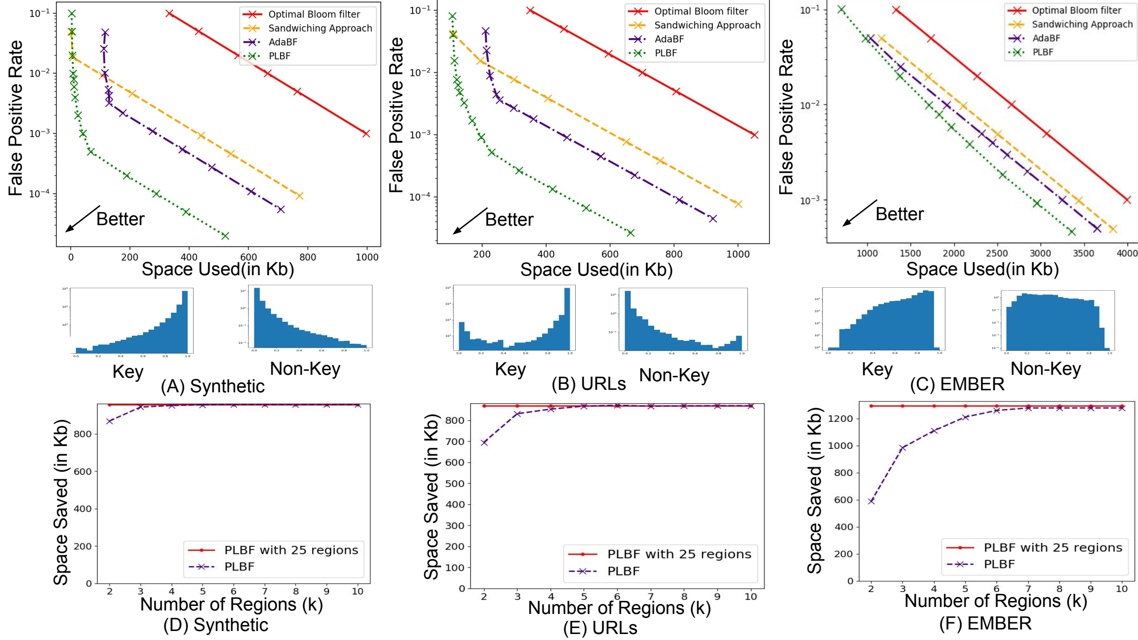


Figure 2-2: FPR vs Space for the (A) Synthetic (B) URLs (C) EMBER datasets for various baselines along with key and non-key score distributions. Space Saved as we increase number of regions for the (D) Synthetic (E) URLs (F) EMBER datasets for PLBF compared to the optimal Bloom filter

2.4.1 Overall Performance

Here, we compare the performance of PLBF against other baselines by fixing the target F and measuring the space used by each methods. We use PLBF Alg.3 with DP algorithm discretization(N) set to 1000. We train the model on the entire key set and 40% of the non-key set. The thresholds and backup Bloom filters are then tuned using this model with the aim of achieving the fixed *target* F. The rest of the non-keys are used to evaluate the *actual* false positive rate.

While any model can be used, we choose the random forest classifier from sklearn [[129]] for its good accuracy. The F1 scores of the learned models used for synthetic, URLs and EMBER were 0.99, 0.97, and 0.85, respectively. We consider the size of the model to be the pickle file size on the disk (a standard way of serializing objects in

[4]For the space of a theoretically optimal Bloom filter, we take the standard Bloom filter of same false positive rate and divide it's space used by $\log_2 e$, as obtaining near-optimality in practice is difficult. This uses the fact that the standard Bloom filter is asymptotically $\log_2 e$ times suboptimal than the optimal as discussed in Sec.2.2.1.

[5]PLBF performs better against standard Bloom filters, as discussed in Sec.2.4.9. Section 2.4.1 are conservative estimates of gains possible in practice using a PLBF.

Python). We use five regions ($k = 5$) for both PLBF and AdaBF as this is usually enough to achieve good performance as discussed in 2.4.2. Using higher k would only improve our performance.

The results of the experiment are shown in the Fig.2-2(A-C) along with the distribution of the scores of keys and non-keys for each dataset. As we can see from the figure, PLBF has a better Pareto curve than the other baselines for all the datasets. On the synthetic dataset and URLs dataset we observe a significantly better performance. In contrast, for the EMBER dataset our performance is only slightly better indicating that the model here is not as helpful. The difference between space used by PLBF and optimal Bloom filter first increases with decreasing false positive rate but converges to a constant value for all datasets, as given in Eq.2.10. For the same amount of space used(400Kb,500Kb,3000Kb space for synthetic,URLs,EMBER, respectively), PLBF achieves 22x, 26x, and 3x smaller false positive rates than the sandwiching approach, and 8.5x, 9x, and 1.9x smaller false positive rates than AdaBF for synthetic, URLs, and EMBER, respectively. To achieve a false positive rate of 0.001, the sandwiching approach uses 8.8x, 3.3x, and 1.2x the amount of space and AdaBF uses 6x, 2.5x, and 1.1x the amount of space compared to PLBF for synthetic, URLs, and EMBER datasets respectively.

2.4.2 Performance and the Number of Regions

The maximum space savings obtained by using PLBF is linearly proportional to the KL divergence of the distributions(Eq2.10) and this KL divergence strictly increases with the number of regions(Sec.2.4.8). Fig.2-2(D-F) show the space saved w.r.t the optimal Bloom filter as we increase the number of regions k for a target false positive rate of 0.001. The red line in the figure shows the savings when using 25 regions; using more regions provides no noticeable improvement on this data. Our results suggest using 4-6 regions should be sufficient to obtain reasonable performance. We have additional experiments in Sec.2.4.10 that shows PLBF performance against standard Bloom filters and PLBF performance w.r.t model quality.

2.4.3 Solving the Relaxed Problem using KKT conditions

As mentioned in the main text, if we relax the constraint of $f_i \leq 1$, using the stationary KKT conditions we can obtain the optimal f_i values. Here we show this work. The appropriate Lagrangian equation is given in Eq.2.12. In this case, the KKT coniditions tell us that the optimal solution is a stationary point of the Lagrangian. Therefore, we find where the derivative of the Lagrangian with respect to f_i is zero.

$$L\left(t_i, f_i, \lambda, \nu_i\right) = \sum_{i=1}^{k} \left(G(t_i) - G(t_{i-1})\right) \times c\log_2\left(\frac{1}{f_i}\right) + \lambda \times \left(\left(\sum_{i=1}^{k} \left(H(t_i) - H(t_{i-1})\right) \times f_i\right) - F\right) + \sum_{i=1}^{k} \nu_i \times (t_{i-1} - t_i) \tag{2.12}$$

$$\frac{\partial L(t_i, f_i, \lambda, \nu_i)}{\partial f_i} = 0 \tag{2.13}$$

$$\frac{\partial (G(t_i) - G(t_{i-1})) c\log_2\left(\frac{1}{f_i}\right)}{\partial f_i} = -\lambda \frac{\partial (H(t_i) - H(t_{i-1})) \times f_i}{\partial f_i} \tag{2.14}$$

$$f_i = \frac{c\ln(2) \times (G(t_i) - G(t_{i-1})) \times \lambda}{(H(t_i) - H(t_{i-1}))} \tag{2.15}$$

$$\lambda = \frac{F}{c\ln(2) \times \sum_{i=1}^{k} (G(t_i) - G(t_{i-1}))} = \frac{F}{c\ln 2} \tag{2.16}$$

$$f_i = \frac{(G(t_i) - G(t_{i-1})) \times FPR}{(H(t_i) - H(t_{i-1}))} \tag{2.17}$$

Eq.2.15 expresses fpr_i in terms of λ. Summing Eq.2.15 over all i and using the relationship between F and H we get Eq.2.16. Thus the optimal f_i values turn out to be as given in Eq.2.17.

2.4.4 Optimal False Positive Rate for given thresholds

We provide the pseudocode for the algorithm to find the optimal false positive rates if threshold values are provided. The corresponding optimization problem is given in

Algorithm 1 Finding optimal fpr's given thresholds

Input G' - the array containing key density of each region
Input H' - the array containing non-key density of each region
Input F - target overall false positive rate
Input k - number of regions
Output f - the array of false positive rate of each region

```
1:  procedure OptimalFPR(G', H', F, k)
2:      G_sum ← 0                                   ▷ sum of key density of regions with f_i = 1
3:      H_sum ← 0                                   ▷ sum of non-key density of regions with f_i = 1
4:      for i in 1, 2, ...k do
5:          f[i] ← (G'[i]·F) / H'[i]                ▷ Assign relaxed problem solution
6:      while some f[i] > 1 do
7:          for i in 1, 2, ...k do
8:              if (f[i] > 1) then f[i] ← 1         ▷ Cap the false positive rate of region to one
9:          G_sum ← 0
10:         H_sum ← 0
11:         for i in 1, 2, ...k do
12:             if (f[i] = 1) then G_sum ← G_sum + G'[i]; H_sum ← H_sum + H'[i]   ▷ Calculate key,non-key density in
    regions with no Bloom filter(f[i] = 1)
13:         for i in 1, 2, ...k do
14:             if (f[i] < 1) then f[i] = (G'[i]·(F − H_sum)) / (H'[i]·(1 − G_sum))   ▷ Modifying the f_i of the regions to ensure target false
    positive rate is FPR
15:     return fpr Array
```

Eq.2.18. As the boundaries for the regions are already defined, one only needs to find the optimal false positive rate for the backup Bloom filter of each region.

$$\begin{aligned} \min_{f_{i=1\ldots k}} \quad & \textstyle\sum_{i=1}^{k} \left(G(t_i) - G(t_{i-1})\right) \times c \log_2\left(\frac{1}{f_i}\right) \\ \text{constraints} \quad & \textstyle\sum_{i=1}^{k} \left(H(t_i) - H(t_{i-1})\right) \times f_i = F \\ & f_i \leq 1 \quad i = 1 \ldots k \end{aligned} \tag{2.18}$$

Alg.1 gives the pseudocode. We first assign false positive rates based on the relaxed problem but may find that $f_i \geq 1$ for some regions. For such regions, we can set $f_i = 1$, re-solve the relaxed problem with these additional constraints (that is, excluding these regions), and use the result as a solution for the general problem. Some regions might again have a false positive rate above one, so we can repeat the process. The algorithm stops when there is no new region with false positive rate greater than one. This algorithm finds the optimal false positive rates for the regions when the thresholds are fixed.

Algorithm 2 Using relaxed solution for the general problem

Input G_{dis} - the array containing discretized key density of each region
Input H_{dis} - the array containing discretized key density of each region
Input F - target overall false positive rate
Input k - number of regions
Output t - the array of threshold boundaries of each region
Output f - the array of false positive rate of each region
Algorithm ThresMaxDivDP - DP algorithm that returns the thresholds maximizing the divergence between key and non-key distribution.
Algorithm CalcDensity - returns the region density given thresholds of the regions
Algorithm OptimalFPR - returns the optimal false positive rate of the regions given thresholds
Algorithm SpaceUsed - returns space used by the back-up Bloom filters given threhsolds and false positive rate per region.

1: **procedure** SOLVE(G_{dis}, H_{dis}, F, k)
2: $t \leftarrow$ ThresMaxDivDP(G_{dis}, H_{dis}, k) ▷ Getting the optimal thresholds for the relaxed problem
3: $G', H' \leftarrow$ CalcDensity(G_{dis}, H_{dis}, t)
4: $f =$ OptimalFPR(G', H', F, k) ▷ Obtaining optimal false positive rates of the general problem for given thresholds
5:
6: **return** t , f Array

2.4.5 Algorithm for finding thresholds

We provide the pseudocode for the algorithm to find the solution for the relaxed problem; Alg.3 finds the thresholds and false positive rates. As we have described in the main text, this algorithm provides the optimal parameter values, if $(G(t_{i-1}) - G(t_i))/(H(t_{i-1}) - H(t_i))$ is monotonically increasing.

The idea is that only the false positive rate of the rightmost region can be one. The algorithm receives discretized key and non-key densities. The algorithm first iterates over all the possibilities of the rightmost region. For each iteration, it finds the thresholds that maximize the KL divergence for the rest of the array for which a dynamic programming algorithm exists. After calculating these thresholds, it finds the optimal false positive rate for each region using Alg.1. After calculating the thresholds and false positive rates, the algorithm calculates the total space used by the back-up Bloom filters in PLBF. It then remembers the index for which the space used was minimal. The t_i's and f_i's corresponding to this index are then used to build the backup Bloom filters. The worst case time complexity is then $\mathcal{O}(N^3k)$.

Algorithm 3 Solving the general problem

Input G_{dis} - the array containing discretized key density of each region
Input H_{dis} - the array containing discretized key density of each region
Input F - target overall false positive rate
Input k - number of regions
Output t - the array of threshold boundaries of each region
Output f - the array of false positive rate of each region
Algorithm ThresMaxDivDP - DP algorithm that returns the thresholds maximizing the divergence between key and non-key distribution.
Algorithm CalcDensity - returns the region density given thresholds of the regions
Algorithm OptimalFPR - returns the optimal false positive rate of the regions given thresholds

1: **procedure** SOLVE(G_{dis}, H_{dis}, F, k)
2: $MinSpaceUsed \leftarrow \infty$ ▷ Stores minimum space used uptil now
3: $index \leftarrow -1$ ▷ Stores index corresponding to minimum space used
4: $G_{last} \leftarrow 0$ ▷ Key density of the current last region
5: $H_{last} \leftarrow 0$ ▷ Non-key density of the current last region
6:
7: **for** i **in** $k-1, k, ...N-1$ **do** ▷ Iterate over possibilities of last region
8: $G_{last} \leftarrow \sum_{j=i}^{N} G_{dis}[j]$ ▷ Calculate the key density of last region
9: $H_{last} \leftarrow \sum_{j=i}^{N} H_{dis}[j]$
10: $t \leftarrow$ ThresMaxDivDp($G[1..(i-1)], H[1..(i-1)], k-1$) ▷ Find the optimal thresholds for the rest of the array
11: $t.append(i)$
12: $G', H' \leftarrow$ CalcDensity(G_{dis}, H_{dis}, t)
13: $f =$ OptimalFPR(G', H', F, k) ▷ Find optimal false positive rates for the current configuration
14: **if** ($MinSpaceUsed <$ SpaceUsed(G_{dis}, H_{dis}, t, f))
15: **then** $MinSpaceUsed \leftarrow$ SpaceUsed(G_{dis}, H_{dis}, t, f); $index \leftarrow i$ ▷ Remember the best performance uptil now
16:
17: $G_{last} \leftarrow \sum_{j=index}^{N} G_{dis}[j]$
18: $H_{last} \leftarrow \sum_{j=index}^{N} H_{dis}[j]$
19: $t \leftarrow$ ThresMaxDivDP($G[1..(index-1)], H[1..(index-1)], k-1$)
20: $t.append(index)$
21: $G', H' \leftarrow$ CalcDensity(G_{dis}, H_{dis}, t)
22: $f =$ OptimalFPR(G', H', F, k)
23:
24: **return** t , f Array

2.4.6 Additional Considerations

2.4.7 Sandwiching: A Special Case

We show here that the sandwiching approach can actually be interpreted as a special case of our method. In the sandwiching approach, the learned model is sandwiched between two Bloom filters as shown in Fig.2-3(A). The input first goes through a Bloom filter and the negatives are discarded. The positives are passed through the learned model where based on their score $s(x)$ they are either directly accepted when $s(x) > t$ or passed through another backup Bloom filter when $s(x) \leq t$. In our setting, we note that the pre-filter in the sandwiching approach can be merged with the backup filters to yield backup filters with a modified false positive rate. Fig.2-3(B) shows

what an equivalent design with modified false positive rates would look like. (Here equivalence means we obtain the same false positive rate with the same bit budget; we do not consider compute time.) Thus, we see that the sandwiching approach can be viewed as a special case of the PLBF with two regions.

However, this also tells us we can make the PLBF more efficient by using sandwiching. Specifically, if we find when constructing a PLBF with k regions that $f_i < 1$ for all i, we may assign $f_0 = \max_{1 \leq i \leq k} f_i$. We may then use an initial Bloom filter with false positive rate f_0, and change the target false positive rates for all other intervals to f_i / f_0, while keeping the same bit budget. This approach will be somewhat more efficient computationally, as we avoid computing the learned model for some fraction of non-key elements.

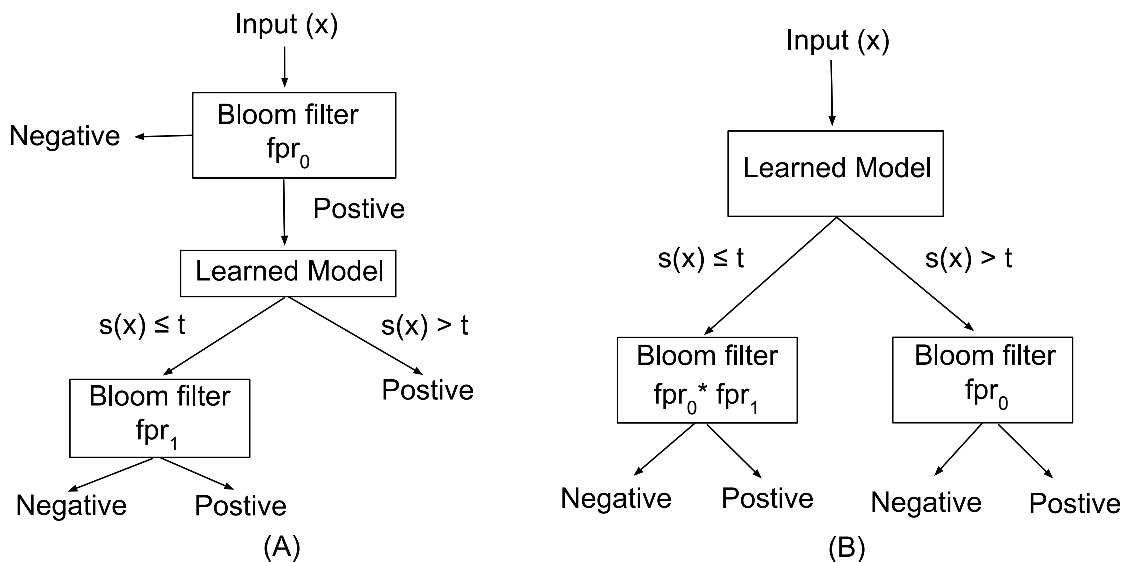


Figure 2-3: (A) represent LBF with sandwiching.(B) represents a PLBF design equivalent to the sandwiching approach.

2.4.8 Performance against number of regions k

Earlier, we saw the maximum space saved by using PLBF instead of a normal Bloom filter is linearly proportional to the $D_{KL}(g(\hat{\mathbf{t}}), h(\hat{\mathbf{t}}))$. If we split any region into two regions, the overall divergence would increase because sum of divergences of the two split regions is always more than the original divergence, as shown in Eq.2.19. Eq.2.19

is an application of Jensen's inequality.

$$\left((g_1 + g_2) \times \log \frac{(g_1+g_2)}{(h_1+h_2)}\right) \;\leq\; \left(g_1 \times \log \frac{g_1}{h_1}\right) + \left(g_2 \times \log \frac{g_2}{h_2}\right) \tag{2.19}$$

Increasing the number of regions therefore always improves the maximum performance. We would hope that in practice a small number of regions k would suffice. This seems to be the the case in our experience; we detail one such experiment in our evaluation(2.4.2).

2.4.9 Performance using various Bloom filter variants

We consider how the space saved of the PLBF varies with the type of backup Bloom filter being used. The PLBF can use any Bloom filter variant as the backup Bloom filter. When we compare our performance with a Bloom filter variant, we use that same Bloom filter variant as the backup Bloom filter for a fair comparison.

First, absolute space one can save by using a PLBF instead of a Bloom filter variant is given in Eq.2.10. This quantity increases with increasing c[6].

The relative space one saves by using PLBF instead of the given Bloom filter variant is shown in Eq.2.20. This quantity is the ratio of the space saved by PLBF (as shown in Eq.2.10) divided by the space used by the given Bloom filter variant ($c \times |S| \times \log_2(1/F)$) as shown in Eq.2.20.

$$\frac{\left(c \times |S| \times D_{KL}\left(\hat{g(\mathbf{t})}, \hat{h(\mathbf{t})}\right) - \text{Size Of Learned Model}\right)}{c \times |S| \times \log_2(1/F)} \tag{2.20}$$

Cancelling the common terms we obtain the following Eq.2.21.

$$\left(\frac{D_{KL}\left(\hat{g(\mathbf{t})}, \hat{h(\mathbf{t})}\right)}{\log_2(1/F)} - \frac{\text{Size Of Learned Model}}{c \times |S| \times \log_2(1/F)}\right) \tag{2.21}$$

The relative space saved, like the absolute space saved, also increases with increasing c. Thus, both the relative and absolute space saved for the PLBF is higher for a

[6]The sizes of standard Bloom filter variants are proportional to $|S| \times \log_2(1/f)$, where S is the set it represents, and f is the false positive rate it achieves. See e.g. [111] for related discussion. The constant c depends on which type of Bloom filter is used as a backup. For example, $c = \log_2(e)$ for standard Bloom filter, $c = 1.0$ for the optimal Bloom filter.

standard Bloom filter ($c = 1.44$) than an optimal Bloom filter ($c = 1.00$), and hence our experiments in Section 2.4.1 are conservative estimates of gains possible in practice using PLBF.

2.4.10 Additional Experiments

2.4.11 Performance w.r.t standard Bloom filters

Earlier, we evaluated our performance using optimal Bloom filters and here we present results using standard Bloom filters. As shown in Sec.2.4.9, PLBF performs better w.r.t standard Bloom filters than optimal Bloom filters. As one can see from Fig.2-4, PLBF performs better than the standard Bloom filter.

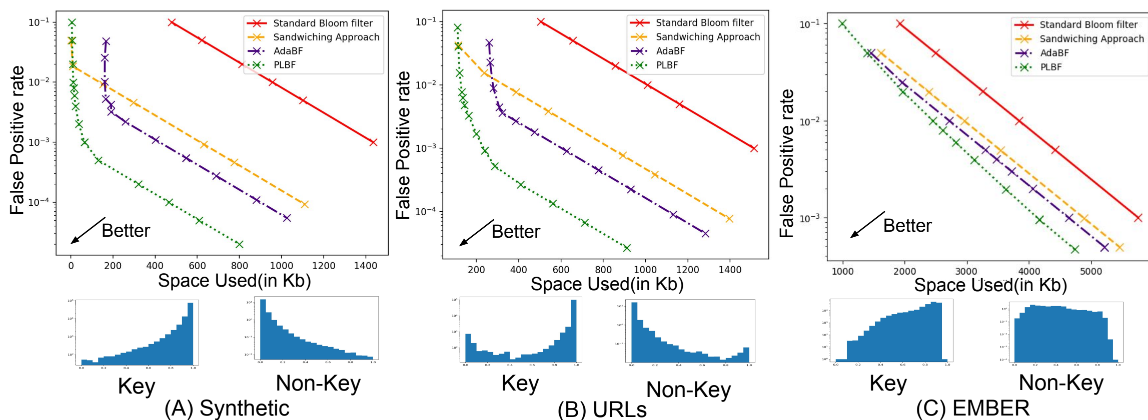


Figure 2-4: FPR vs Space for the (A) Synthetic (B) URLs (C) EMBER datasets for various baselines along with key and non-key score distributions.

2.4.12 Performance and Model Quality

Here we provide an experiment to see how the performance of various methods varies with the quality of the model. As discussed earlier, a good model will have high skew of the distributions g and h towards extreme values. We therefore vary the skew parameter of the Zipfian distribution to simulate the model quality. We measure the quality of the model using the standard F1 score. Fig.2-5(B) represents the space used by various methods to achieve a fixed false positive rate of 0.001 as we vary the

F1 score of the model. The figure shows that as the model quality in terms of the F1 score increases, the space required by all the methods decreases (except for the optimal Bloom filter, which does not use a model). The space used by all the methods goes to zero as the F1 score goes to 1, as for the synthetic dataset there is no space cost for the model. The data point corresponding to F1 score equal to 0.99 was used to plot Fig.2-2(A).

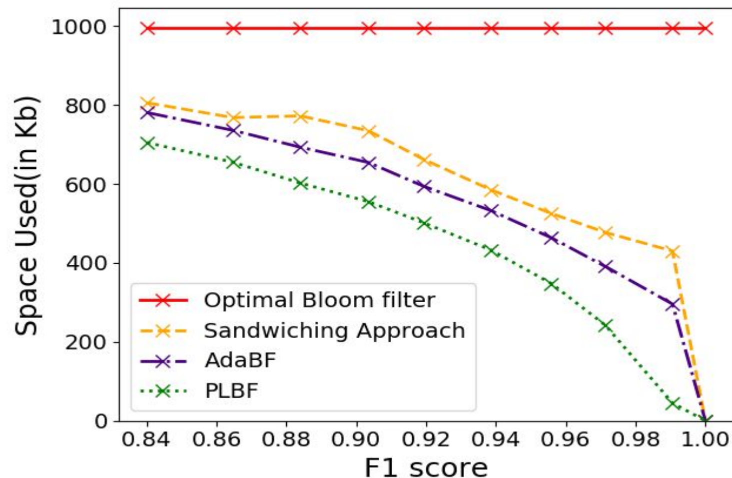


Figure 2-5: Space used by various baselines as we increase F1 score for Synthetic dataset

2.4.13 Discretization Effect on Dynamic Programming Runtime, PLBF Size

All the runtime experiments in this subsection and the next subsection are measured using an 2.8GHz quad-core Intel Core i7 CPU @ 2.80GHz with 16GB of memory. We use the *bloom-filter* python package [[17]] for our backup Bloom filters. The dynamic programming algorithms are implemented in Python.

Here we provide an experiment to see how the dynamic programming (DP) algorithm runtime (psuedo code in Alg.3) and PLBF size vary with level of discretization (N). In the tables below, we have the DP algorithm runtime and space taken by the PLBF to achieve an approximate empirical false positive rate of 0.001 for various N. As discussed in Sec. 2.3.3, with increasing value of N one gets closer to optimal

parameters, at the cost of higher computation time. This trend is demonstrated in the table below for the URLs and EMBER datasets. We note that if runtime is an issue, the increase in size from using smaller N is relatively small.

N	DP Runtime(in sec)	PLBF Size (in Kb)
50	1.17	187.6
100	2.15	184.37
500	10.97	183.63
1000	26.94	183.55
2000	56.79	182.85

Table 2.1: DP runtime and space used by PLBF as we increase the discretization N in the URLs dataset

N	DP Runtime(in sec)	PLBF Size (in Kb)
50	1.36	2952.33
100	2.52	2944.68
500	11.39	2933.09
1000	25.26	2928.76
2000	56.12	2926.79

Table 2.2: DP runtime and space used by PLBF as we increase the discretization N in the EMBER dataset

2.4.14 Construction Time for Various Baselines

Here we look at the construction time breakdown for the PLBF and various alternatives, with the goal of seeing the cost of in terms of construction time for using the more highly tuned PLBF. The construction time of all the learned Bloom filters includes the model training time and parameter estimation time, which are not required for the standard Bloom filter construction process. Since we use the same model for all learned baselines, the model construction time is the same for all of them. In Fig.2-6, we plot the construction time breakdown for various baselines in order to achieve an approximate empirical false positive rate of 0.001. Recall that the AdaBF and Sandwiching approaches use heuristics to estimate their parameters and unsurprisingly they therefore seems somewhat faster than PLBF. However, for $N = 100$ we see the parameter estimation time is smaller than the key insertion time and model training

time. The parameter estimation time for PLBF varies with the level of discretization we use for the DP algorithm. The PLBF with $N = 1000$ takes the longest to execute while standard Bloom filter is fastest baseline. As shown in Table2.1 above, using $N = 1000$ gives only a slight improvement in size. We therefore believe that if construction time is an issue, as for situations where one might want to re-learn and change the filter as data changes, one can choose parameters for PLBF construction that would still yield significant benefits over previous approaches.

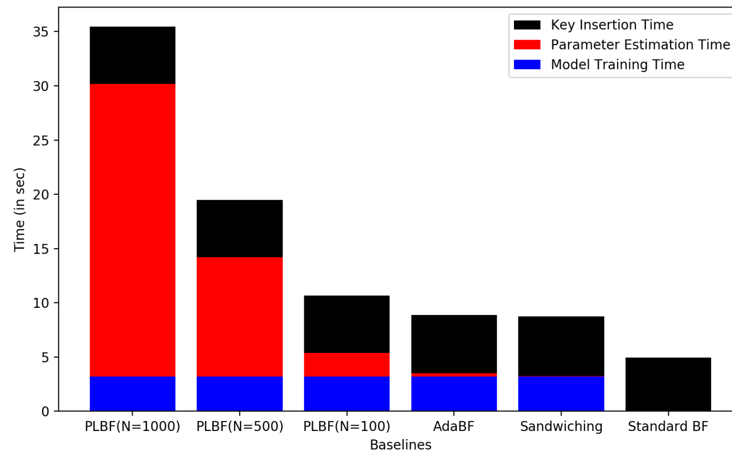


Figure 2-6: Construction time breakdown for various baselines for the URLs dataset

Chapter 3

SNARF: Sparse Numerical Array Based Range Filter

3.1 Introduction

Filters are space efficient, but approximate, data structures that are used to answer membership queries on a set S. Filters allow significant improvements in the performance for an array of applications, including big data systems [150] and networking [22]. For example RocksDB [53], a Log-Structure-Merge Tree (LSM) [121] based key-value store, stores data onto disks in blocks (called SST's). However, because of the LSM structure, RocksDB often needs to load several blocks from disk into main memory to determine which block contains the data for a given search key. To avoid loading disk blocks that do not contain the search key, RocksDB creates a filter per block for all keys stored in the block.

Point filters, such as Bloom Filters, support point queries of the form: "Is x in the set S?". Range membership filters answer more general queries of the form "Is there a key in the set S in between values p and q?" [2, 100, 161]. Here we are focused on approximate filters that guarantee that there are no *false negatives*. This is an important property many applications/systems require. There may, however, be false positives. For point queries, if the filter returns *true* for a search key, the key *might or might not be* contained in the block, but if it returns *false* it is guaranteed that the key

is not in the set/block; and this extends similarly to range queries. The probability of a false positive for a key not in the set is the false positive rate (FPR) of the filter; the FPR can be defined similarly for range queries.

In RocksDB, filters are usually orders of magnitude smaller than the blocks and are cached in main memory. Before loading a disk block into main memory, the filters are checked if the key might be contained in the block. A filter with low false positive rate helps to significantly reduce the number of unnecessary I/O requests to disk blocks to find the key. The benefit a filter can provide depends on the trade-off between its false positive rate and the size of the filter; the smaller and the more precise, the better it is. Interestingly, the latency of a filter to process a query normally matters less as they tend to protect against very expensive operations (e.g., disk or other cold storage access) that are often orders of magnitude slower (see also Experiment 3.6.2).

Range Filters: Range queries are often used in social web applications [30], distributed key-value storage replication [143], statistics aggregation for time series workloads [77], and SQL table accesses [91]. For example, from a table of customer orders, one might ask the following SQL query to retrieve all the orders between two particular dates: SELECT * FROM Orders WHERE Order_Date BETWEEN "07-14-2014" AND "07-21-2014" . Past work has shown that range filters can significantly improve the performance of systems for synthetic and real-world workloads. For example, [100, 161] showed that workloads on RocksDB can benefit from range filters, whereas [2] showed the advantages of range filters for Hekaton, which is part of the MS SQL Server.

Existing Range Filter Designs: Past efforts to provide range filtering resulted in the current state-of-the-art filters Succinct Range Filter (SuRF) [161] and Rosetta [100]. SuRF utilizes a compact trie-like data structure which can filter arbitrary range queries, whereas Rosetta utilizes a different approach by using a Bloom filter (a point query filter) [16] for range queries along with the help of a hierarchy of prefix Bloom filters. Unfortunately, which of the two filters is better depends highly on the workload. For the same filter size, Rosetta has a lower false positive rate for very short range sizes because of its clever combination of Bloom and prefix filters, whereas SuRF has

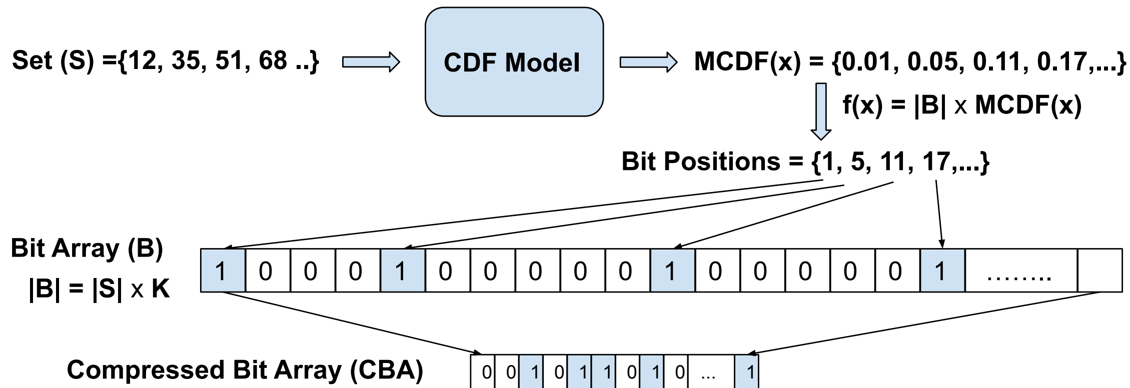


Figure 3-1: SNARF Idea: Given a set of keys S, SNARF builds a model $MCDF(x)$ to estimate the empirical cdf of the keys, which it then uses to set corresponding bits in a large bit array B for all $x \in S$. This sparse bit array which encodes key information is then compressed. The model and the compressed bit array are the main parts of SNARF data structure.

a lower FPR otherwise.

SNARF: In this paper, we introduce an entirely new approach to range filters, called Sparse Numerical Array-Based Range Filters (SNARF). SNARF is a learning enhanced range filter[1] that models the data distribution of the underlying key set S. SNARF then uses the model to encode partial information of the data in a sparse bit array. SNARF controls the false positive rate by changing the size of the bit array. The sparse bit array is then compressed to store it efficiently. SNARF answers range queries by using the model to extract the relevant information from the compressed bit array. Exploiting the data distribution and using effective compression schemes allow SNARF to encode the data set more effectively than previous schemes, leading to better space/false positive rate tradeoffs, while being competitive in terms of query latency.

SNARF Results: We evaluate SNARF on multiple synthetic and real-world numerical datasets against state-of-the-art range filters, such as SuRF and Rosetta, and also against point filters, such as Bloom filters [16] and Cuckoo filters [55]. We use a variety of query workloads, such as uniform, sampled from real-world, skewed

[1]We acknowledge that the term "learned" range filter might be a misnomer as we use simplistic modelling of the data using linear splines. However, the name is in line with previous works [56, 88, 85, 46].

(certain part of data is queried more often), and correlated (query endpoint is close to existing key) to test the effectiveness of the filters. For range queries on both real-world and synthetic datasets, SNARF is consistently able to provide up to a 50x better FPR than SuRF under the same space budget, and SNARF has up to 100x better FPR than Rosetta under same space budget. We do note, however, that performance depends on the dataset and query structure; for example, we have found that Rosetta is better than SNARF specifically in the case where the query workload has very short range queries and high correlation between queries and keys. Moreover, for point queries, SNARF can empirically provide FPRs that are better than Bloom filters and slightly better than Cuckoo filters under the same space budget across a diversity of query workloads.

Finally, we measured SNARF's impact on performance of an end-to-end system by integrating it with RocksDB. Here we found that SNARF can improve the workload execution time by up to 10x compared to SuRF and Rosetta for certain read only workloads.

In summary, we make the following **contributions**:

- We introduce SNARF, a novel range filter which combines models and compression schema (Section 3.2).
- We provide a heuristic theoretical analysis of SNARF that matches our empirical experiments well (Section 3.3).
- We discuss possible extensions of SNARF, including support for updates and support for approximate count queries (Section 3.4).
- We evaluate SNARF against state-of-the-art baselines and test the impact SNARF can have on a real system like RocksDB (Section 3.6).

3.2 SNARF: A learned filter

We first explain the idea behind SNARF (see Sec.3.2.1). Later, we describe the details of the model (see Sec.3.2.2) and the compressed bit arrays (see Sec.3.2.3).

3.2.1 SNARF Description

SNARF Construction:

Given a set of keys $S = \{x_1, x_2, \ldots, x_n\}$, we want to build a filter that answers range queries on this set. SNARF maps the keys into a bit array B, which has $|B| = K \times n$ bits for a suitably large K[2], via a monotonic function f. Initially, all bits are 0, but bit position $f(x_i)$ is set to 1 for all $x_i \in S$. The exact mapping function f is $f(x) = \lfloor MCDF(x) \times nK \rfloor$, where MCDF is a monotonic estimate of the empirical CDF (eCDF) of the keys in S. Storing an entire sparse bit array directly is not space-efficient, so SNARF stores a compressed version of the bit array. The compressed bit array (CBA) encodes the locations of the one bits in the array. Fig.3-1 illustrates the idea of SNARF.

Alg.4 has the pseudo-code for SNARF construction. Given a set of keys S and scale factor K for the bit array, the construction algorithm outputs a model of the eCDF of the keys in S and the compressed bit array. The first step is to train a model to estimate the eCDF of the keys. In the next step, this model is used to generate the set of bit positions in the bit array that are set to one. The bit array is then compressed into the CBA.

SNARF Range Query:

To answer a range query $[p, q]$, SNARF uses the model to get the bit positions $f(p)$ and $f(q)$ corresponding to the query endpoints. The data structure then returns true if a one bit is found in the range $[f(p), f(q)]$ of B and false otherwise. Alg.5 shows the pseudo-code for SNARF range query. Note that we want our CBA structure to efficiently support queries of the form: "Is there a one bit between bit positions a and b (inclusive)?".

Standard rank-select structures [163, 65, 128, 69] can provide compressed bit arrays with an efficient predecessor query which can be used to answer such queries. (One can

[2] We use K to control the FPR of the structure which we discuss in detail later on

check if the first one bit preceding b is before or after a.) Such a structure is naturally more efficient than decompressing the entire array and checking all bits between a and b. While rank-select structures could be used to speed up the computation of the a predecessor operation, we find they take more space than alternatives to do so. In our case, space is the primary resource we want to optimize for. This is because the latency of a filter to process a query normally matters less in RocksDB (see Sec.3.6.2). Also, with exponentially growing data, it is important to be able to filter more data with smaller filters. Thus, SNARF uses encoding schemes which provide near-optimal compression rather than fast query responses. We discuss simple techniques to optimize query response times in Sec.3.2.3.

Essential properties of Mapping Function f

Monotonicity: The monotonicity of the mapping function, so that $p < q \implies f(p) \leq f(q)$, is an essential property that ensures no false negatives in SNARF. Monotonicity ensures that for any range query $[p, q]$ with $p < q$, any key from S between the query endpoints will be mapped to a position between the bit positions of these endpoints. That is, if $x_i \in [p, q]$, then $f(p) \leq f(x_i) \leq f(q)$; there is a bit set in the range $[f(p), f(q)]$. Note, however, that it is possible that $x_i \notin [p, q]$, but either $f(x_i) = f(p)$ or $f(x_i) = f(q)$, leading to false positives.

Uniform Mapping: SNARF aims for a uniform mapping into the bit array B for performance reasons; that is, we desire the bits set in the array to be as equally spaced as possible. Mapping the keys approximately uniformly allows the range filter to be robust to skewed query workloads (workloads where certain part of the range is queried more often) as we discuss in detail in Sec.3.3. The empirical cumulative distribution function of a (discrete) set S has the property that it maps the keys uniformly over the range $[0, 1]$. Hence, SNARF makes use of a monotonic CDF model of the set S to achieve a monotonic and approximately uniform mapping of the keys. The details of the model we utilize are presented in Sec.3.2.2.

Algorithm 4 SNARF Construction:

Input S - set of keys
Input n - number of keys
Input K - Scaling factor for the bit array size
Output $MCDF$ - Monotonic CDF estimate of keys
Output CBA - Compressed bit array
Function $Train(S)$ - function that returns a model to estimate the cdf of keys in set S
Function $Encode(S)$ - function that encodes the numbers in the set S

```
1: procedure CONSTRUCTION(S, K)
2: //Building the monotonic CDF model for set of keys
3:     MCDF ← Train(S)
4:
5: //Get Bit positions that are set to one
6:     BitPositionList ← {}
7:     for key in S do
8:         BitPositionList.add(⌊ MCDF(key) × nK ⌋)
9:
10: //Compress the Bit Positions that are set to one
11:     CBA ← Encode(BitPositionList)
12:
13:     return MCDF, CBA
```

Algorithm 5 SNARF Range Query

Input n - number of keys
Input K - Scaling factor for the bit array size
Input $MCDF$ - Monotonic CDF estimate of keys
Input CBA - Compressed bit array
Input p, q - the range query endpoints
Output r - boolean answer of the range query
Function $CheckOneBit(a, b)$ - function that returns *true* if there is a 1 bit between bit locations $[a, b]$ else *false*.

```
1: procedure RANGEQUERY(MCDF, K, n, CBA, p, q)
2: //Get the bit location of the query endpoints
3:     LowerBitLoc ← ⌊MCDF(p) × Kn⌋
4:     UpperBitLoc ← ⌊MCDF(q) × Kn⌋
5:
6: //Check if 1 bit exists in the range.
7:     r ← CBA.CheckOneBit(LowerBitLoc, UpperBitLoc)
8:     return r
```

3.2.2 Model Details

As discussed before, the model needed for SNARF must be monotonic and provide an estimate of the empirical CDF. Further, we want the space overhead added by the model to be small. Here, we present models for fixed size numerical values such as doubles, floats, and 32/64/128 bit signed/unsigned integers. Recently, a hierarchy of linear models have been used for indexing numerical keys [56, 85, 46, 72]. This ensemble of linear models is both small in size and provides fast evaluation for

numerical values.[3] However, these models do not always guarantee monotonicity.

Inspired by them, we use linear spline models for CDF estimation. Given a set S, the idea is to use a small sample of keys from the input set and build linear models between consecutive keys in the sample to estimate the CDF as shown in Fig.3-2. This sample is stored in a sorted order, and we refer to it as the key array. The size of the sample determines how large the model is and the quality of the CDF estimation. Larger samples lead to better CDF approximation and larger models which increase the space used by SNARF.

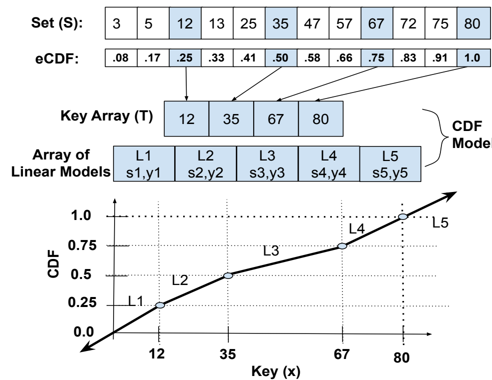


Figure 3-2: SNARF Numerical Model

Querying the Model: The number of keys stored by the model is one less than the number of linear models. The first step is to binary search in the sorted array of keys ($MCDF.keys$). The number of keys in the array that are less than the query point x gives the index to the linear model parameters that are supposed to be used. We then use the corresponding line's slope and intercept to obtain the final estimated CDF value for the value. If the computed CDF is outside the range $[0, 1]$, we correct the value to 0 or 1 as appropriate.

Training the Model: During training, we sort the input set S and compute the empirical CDF. We then choose keys at regular intervals (every $(N/R)^{th}$ key for a suitable R) and these keys form endpoints for linear spline models. Between every pair of consecutive sample points, we compute the slope and y-intercept of the line segment connecting the two points.

[3]We experimented with monotonic cubic splines [60] but found them to be slightly worse than a series of linear models

The number of line segments we use in our model can be tuned to improve the tradeoff between the CDF estimate and overall model size. The more lines the better one can potentially approximate the CDF, but the more space used as well. A good value for number of line segments will depend on the dataset. We empirically found that using $|S|/1000$ line segments generally gives good CDF estimates along with small model size. The space overhead of model when using $|S|/1000$ line segments is approximately 0.2 bits per key.

3.2.3 Managing the Bit Array

We describe compression schemes for bit arrays and simple techniques to make range queries faster on the compressed bit array.

Compressing the Bit Array

The main idea for space efficient encoding of a sparse bit arrays is to simply encode the positions of the one bits. We discuss two such specific techniques.

Golomb Coding: Golomb coding is a form of lossless delta compression which is the optimal lossless compression scheme for a sparse bit array with uniformly randomly spread one bits [63].[4]

In general delta compression schemes, the values to be encoded are sorted and then the differences, or deltas, between consecutive values are stored efficiently. In Golomb coding, for each delta value X to encode, X is divided by a fixed constant M to obtain a quotient $\lfloor X/M \rfloor$ and a remainder $X\%M$. The remainder is stored in a fixed length binary format using $\log_2(M)$ bits, whereas the quotient, which is expected to be small, is encoded in unary. The choice of the fixed constant is important in determining the size of the compressed array. For uniformly randomly generated values, the average delta value is the optimal constant. In our case, the average delta value will be the bit array size nK divided by number of one bits, which is approximately n. We therefore use K as the constant for our Golomb coding. Fig.3-3 describes an example of Golomb

[4]We expect nearly uniform randomly place one bits in our case.

encoding a sparse bit array.

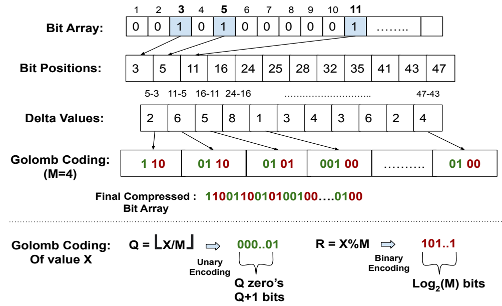


Figure 3-3: Golomb coding

In order to check if a bit is one in the range of bit positions $[a, b]$, one needs to decode the array from the start by adding the deltas one by one. This process continues until you either find a 1 after a and before b or you go past b. Decoding the array for each query can be slow; we discuss better approaches shortly.

Elias-Fano Encoding: Elias-Fano is a form of entropy encoding to represent a monotone non-decreasing sequence of N integers. The bit positions in our case form the non-decreasing sequence. In Elias-Fano encoding, the integers are first binary encoded using $\log_2(M)$ bits if $[0, M)$ is the universe range. This representation is split into two parts: an upper $\log_2(N)$ bits and the remaining lower $\log_2(M/N)$ bits. The lower bits are trivially stored by concatenating them and this uses $N \log_2(M/N)$ bits. The higher part is represented by a bit vector of $2N$ bits as follows. We first create a count of occurrences of upper bit values for all values between $[0, N-1]$. We then put this count vector in unary notation; that is, each count is represented in unary (a sequence of 1s) with 0 stop bit between values. This leads to $2N$ total bits, with one bit set to 1 for each of the N elements and one 0 bit for each possible values for the upper bits. Finally, the Elias-Fano representation is the concatenation of these two vectors. Fig.3-4 describes a Elias-Fano encoding for a set of integers. In our case, we will be encoding the bit positions so $M = nK$ and $N = n$. Thus, we will binary encode the $\log_2(K)$ lower bits and unary encode the upper $\log_2(n)$ bits for each bit position.

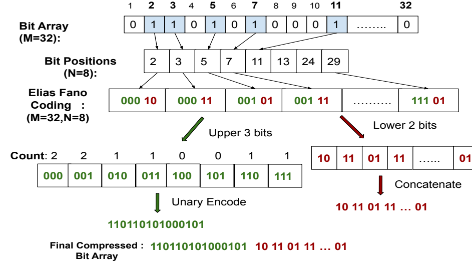


Figure 3-4: Elias Fano Encoding

While checking if a bit is one in the range $[a, b]$, one can decode the upper bit array from the start (similar to Golomb coding) but accessing the lower part is not always necessary. Any bit position with upper bit value less than $\lfloor \frac{a}{2^{M-N}} \rfloor$ will definitely be smaller than a. This is because the value of the lower part can be at max $2^{M-N} - 1$ and that is not enough for it to be greater than a. Thus, we only need to check the lower bits if the upper bits are relevant. This property greatly reduces the amount memory accessed during a range query compared to Golomb coding. On the other hand, Elias-Fano coding uses slightly more space ($\approx$ 0.4-0.5 bits per key) than Golomb coding. Thus, Elias-Fano coding has a faster query time compared to Golomb coding but with a slightly higher space overhead.

Making Compressed Bit Arrays Efficient:

As noted earlier, simply decoding from the beginning is an expensive approach; in the worst case, we might need to decode the entire bit array. To avoid this, we split the bit array into equal sized segments and then compress them separately. If nK is the bit array size and n is the number of keys, we divide the bit array into $K\beta$ sized segments generating n/β segments. Now to perform a range query $[p, q]$ for S we only need to decode the corresponding segments that overlap the range $[f(p), f(q)]$ in the CBA. On an average each segment has around β one bits. While answering the range query $[p, q]$, one only needs to consider segments from segment number $\lfloor f(p)\beta/n \rfloor$ to $\lfloor f(q)\beta/n \rfloor$. The first value greater then $f(p)$ either exists in segment number $\lfloor f(p)\beta/n \rfloor$ or in the next non-empty segment. Generally, decoding segment

number $\lfloor f(p)\beta/n \rfloor$ is sufficient as we find a number greater than p in it or the next segment.

Even though the uncompressed bit array size is the same, the compressed size of each segment differs. Hence, we need to store the starting point of each compressed segment. This creates a tradeoff between space used by SNARF and the range query response time provided by SNARF. Using more segments would lead to faster queries but larger metadata space overhead. Empirically, we found that $\beta \approx 50-100$ provides good range query response times and has negligible memory overhead (shown in Sec.3.6.1).

3.3 Analysis

In the following section, we provide an analysis regarding the tradeoff between the space used by SNARF and the corresponding false positive rate. We show that for point queries SNARF is competitive with Bloom filter variants. The results extend to queries over small ranges in the natural way. While this analysis is only for certain workloads, it provides understanding for why SNARF works well in many scenarios.

We start by showing that SNARF for uniformly distributed queries (point queries and small ranges) provides an FPR of approximately $1/K$ while using $2.4 + \log_2(K)$ bits per key.

Initial Assumptions: We assume all key values are in the range $[0, z]$ for some suitably large z with $z >> nK$.[5]

Notation: Our set S of n keys $S = x_1, x_2, .., x_n$. We use a model with t linear models and thus, we have one linear model per n/t points. Recall we use a bit array of size $n \times K$ and divide it into blocks of size $K\beta$ bits for faster queries; we assume also a per block metadata of c bits.

Analysis: Our goal is to show that for uniform workload SNARF provides a false positive rate of $1/K$ for point and small range queries, while using around $2.4+\log_2(K)$

[5] If $z < nK$, then each value in the domain would likely map to a different bit position. If each value has a different bit position then false positive rate would be zero.

bits per key. For uniform point queries, we have z total queries out of which $z-n$ are negatives. We proceed by showing that SNARF only gives false positive for $(z-n)/K$ point queries.

We divide the key range into t segments of size $\Delta z_1, \Delta z_2, ...\Delta z_t$, where $\sum_{i=1}^{t} \Delta z_i = z$ and each segment has a separate linear model. Let the corresponding segment endpoints be $z_0, z_1, \ldots, z_t$. For each segment the following holds:

- The number of keys from S in the segment $[z_{i-1}, z_i)$ is n/t as we build separate linear model for every n/t keys.

- Over each segment $[z_{i-1}, z_i)$, we have a total of $z_i - z_{i-1}$ distinct possible point queries out of which $((z_i - z_{i-1}) - n/t)$ are negative queries.

- The keys of S in the segment $[z_{i-1}, z_i)$ are evenly spread over the range $[(i-1)(nK/t), i(nK/t))$ in the bit array.

An implication of these statements is that for a non-key in the range $[z_{i-1}, z_i)$, the probability of false positive is at most the number of 1 bits in the range, which is at most n/t, divided by the corresponding size of the range in the bit array, which is nK/t. It follows that the number of keys that give false positives is

$$\sum_{i=1}^{t} ((z_i - z_{i-1}) - n/t)) \times \frac{n/t}{nK/t} = \frac{1}{K} \sum_{i=1}^{t} ((z_i - z_{i-1}) - n/t)) \, .$$

But since $\sum_{i=1}^{t} (z_i - z_{i-1}) = z$ the summation collapses, giving the total number of false positives is $(z-n)/K$. Since, we have $z-n$ negatives in the range the false positive rate turns out to be $1/K$ for the uniform distribution. This shows that for uniform workload using a bit array that is K times larger than the number of keys yields a false positive rate of approximately $1/K$ for point queries.

Extending to small ranges: Here, we perform a similar analysis for uniform range queries of size R. The main idea is to show that the total number of false positive range queries is at most the total number of false positive point queries. We show this for a region and then aggregate across the entire domain.

Consider a region $[p, q]$ of the domain such that all points in the region map to a one bit and values just outside the region map to zero bits. That is, the bit at location $f(p-1))$ is 0, and the bit at location $f(q+1))$ is 0, but for all $x \in [p, q]$, the bit at $f(x)$ is 0. Let l be the number of keys in this region. The number of false positive point queries is $(q+1-(p+l))$. The total number of range queries of size R intersecting with the region would be $(q+1+R-p)$. Out of the these, the number of true positive range queries is at least $(l+R)$ as we show later. Thus, the false positive range queries end up being at most $(q+1+R-p)-(l+R)=(q+1-(l+p))$ which is exactly equal to the number of false positive point queries in the region. Now, we can simply sum up the queries in each such region to get the total number. Thus, we can conclude that total number of false positive range queries is at most the total number of false positive point queries.

We argue that the number of true positive range queries is at least $l+R$ in a region. Let $k_1, k_2, ...k_l$, be the keys in the region $[p, q]$ in sorted order For the smallest key k_1 in the region, we have $R+1$ true positive ranges of size R as enumerated by set $\{(k_1-R, k_1), (k_1-R+1, k_1+1), ...(k_1-R+R, k_1+R)\}$. Now, if we consider k_2, then we can add a unique true positive range query (k_2, k_2+R) to the set. Similarly, every subsequent addition of a key increases the size of the set by at least one. Earlier, we showed that the total number of false positive point queries is z/K. The number of negative range queries is at least $z-nR$. Thus, the false positive rate for range queries is at most

$$\frac{(z/K)}{z-nR} \approx \frac{1}{K}$$

Here the approximation holds for small ranges R, so that $nR << z$, yielding a false positive rate close to $1/K$.

Extending to skewed workloads: We assume there is a distribution with cdf $w(x)$ that generates a point query, such that over suitably small intervals $[z_{i-1}, z_i]$, the probability of querying any point in the range is approximately uniform. Each

segment would independently have a false positive rate of approximately $1/K$, thus it follows that the false positive rate for point queries is:

$$FPR = \sum_{i=1}^{t} (w(z_i) - w(z_{i-1}))) \times \frac{n/t}{nK/t} = \frac{1}{K} \sum_{i=1}^{t} (w(z_i) - w(z_{i-1}))) .$$

The ratio in the summation is approximately $1/K$, giving an approximate false positive rate of $1/K$.

We indeed observe that the false positive rate is approximately $1/K$ for point queries as well as range queries over various query distributions for SNARF empirically for synthetically generated datasets and workloads, as we discuss in Sec. 3.6.1.

Model Size: The size of the model is dependent on the number of keys and linear models it stores. We assume the linear models utilize 2 double values and hence we use 128 bits per linear model. For uint64 integers, we need 64 bits to store each key in the key array. In our experiments, for example, we stored $n/1000$ models and thus, the space used by model is around $192n/1000$. This accounts to approximately 0.2 bits per key.

Compressed Bit array Size: Given that the bit array is Kn bits long, the compressed version of the bit array using Golomb and Elias Fano coding takes no more than $2n + n\log_2(K)$ bits in total[6]. This is because the unary code for both Golomb and Elias Fano coding takes no more than $2n$ bits and the binary representations take $\log_2 K$ bits per key . The space overhead due to dividing the compressed bit array into blocks of size βK bits is approximately nc/β, bits where c is the number of bits per block needed to store the metadata. In our experiments, c is around 20 bits and we fix β to be around 100. Thus, the space used by SNARF per key is around $(2 + \log_2(K) + c/\beta + 192n/1000) \approx (2.4 + \log_2(K))$ bits[7].

Recall that our heuristic analysis gives a false positive rate for point queries of $1/K$. This is close to the theoretical space lower bound of $\log_2(K)$ bits per key for Bloom filter variants [22]. Empirically we observed that SNARF gives a similar false

[6] Note this is the worst case space used. Golomb coding generally uses less space than $2n+n\log_2(K)$ bits.

[7] In practice, we do even better than $(2.0 + \log_2(K))$ bits per key

positive rate for point queries as cuckoo filters with the same space usage on synthetic datasets and workloads as shown in Sec.3.6.1.

3.4 Discussion

We discuss here various aspects of SNARF behavior, including performance on workloads with high key-query proximity, SNARF use for other queries, and handling updates.

3.4.1 Key-Query Correlation in Workloads

For the purpose of range filters, we say that a workload is correlated with the data, if the end point of a query is consistently close to some key. Assume a data set contains all multiples of 10 from 1 to 1000 (e.g., 10, 20, 30,...,1000). A correlated workload would be one which consistently ask for ranges close to these keys (e.g., 10.01-11.01, 28.99-29.99, etc.). When a query end point is consistently close to an actual key but the query does not include a key, it may yield a false positive in SNARF and SuRF. Meanwhile, Rosetta is relatively unaffected by correlated queries as it is uses Bloom filters which are robust towards correlation.

The fact that performance degrades for SuRF and SNARF for correlated queries in these ways is not surprising based on the lower bound result in [66]. The lower bound shows that a range filter that supports range queries up to size R and guarantees a false positive rate of FPR will take at least $\log_2(R) + \log_2(1/\text{FPR})$ bits per key. Hence, for a fixed memory budget, a range filter data structure cannot handle large ranges and a low false positive rate simultaneously without making further assumptions about the data set or workload.

Due to this FPR degradation in SNARF with correlation, Rosetta turns out to be the better filter for workloads with highly correlated and very short range queries. We demonstrate these behaviors empirically in Sec.3.6.1. In big data systems like RocksDB, data is stored in multiple blocks (called SST's in RocksDB) with each block having its own filter. Even if a query is correlated to a key in a certain block, SNARF

is still useful for the rest of the blocks (as shown in Sec.3.6.2).

3.4.2 Handling Updates

A variety of systems like LSM-based key-value stores use immutable files and thus do not need filters that support updates. On the other hand, OLTP systems which are not based on log structured storage schemes would benefit from an updatable range filter. SNARF is able to naturally support updates owing to its design. To support updates, we keep the mapping function static and only modify the bit array. Because we divide SNARF into small blocks for query efficiency, incremental updates only affect the corresponding block without affecting other blocks.

Update Procedure: To perform an insertion/deletion of a key, we use the mapping function to get the bit location of that key. The bit location is used to identify the corresponding block. We simply add/remove the bit location from the block depending on whether it was an insert/delete. In our basic implementation, we allocate a new block and copy all the bit locations from the old block to the new one after adding/removing the bit location corresponding to the update. Updates to a block can be made faster by using the standard technique of over-allocating memory for that block.

Particularly with deletes, removing a bit location might lead to inconsistency as multiple keys might be mapping to the same bit location. A simple workaround for this is to store duplicates of the bit location. If d keys map to the same bit location, we store precisely $d-1$ duplicates.[8] We show some experiments with our basic implementation of updatable SNARF in Sec.3.6.1.

Effect of Updates on SNARF's FPR: We modify the bit array but the mapping function($CDF(x) \times nK$) remains static during updates. Updates can lead n and $CDF(x)$ of the mapping function to diverge from the ideal values. We refer to updates that do not change the distribution of the data as in-distribution updates whereas the ones who do as out-of distribution updates.

[8]Duplication adds small space and query latency overhead for small values of K and the impact is not significant for larger K's.

In-distribution updates do not change the data distribution but may affect the number of keys. Let the final number of keys after updates be n'. After the updates, the ideal mapping function would have been $CDF(x) \times n'K$ to achieve an FPR of $1/K$, whereas we use $CDF(x) \times nK$. The FPR for SNARF thus ends up being $n'/(nK)$ instead of $1/K$. If the in-distribution updates are dominated by inserts, then the FPR becomes worse, and similarly with deletions it gets better.

Out-of-distribution updates may change the data distribution and the number of keys. For out-of distribution updates, predicting the FPR is more complex and it also depends on distribution of queries. We expect the combined effect of change in n and $CDF(x)$ to worsen the FPR of SNARF more than in-distribution updates.

The above discussion also applies to the case of append-only databases. In this setting, when a series of updates significantly reduces the FPR sufficiently, the model should be re-trained and a complete rebuild of the structure would be necessary.

3.5 Related Work

Filter Data Structures: There is a long history of using compact filters to represent sets that are deemed too expensive to store and query explicitly, for reasons including memory limitations, speed, hardware amenability, and others. Indeed, there are now many variants of the canonical Bloom filter [16] that use various hashing schemes to encode the key set (e.g., Cuckoo filters, Quotient filters, Xor filters, Ribbon filters [55, 13, 68, 42]). These filtering schemes are limited to testing a single key at a time. In some ways, our technique resembles compressed Bloom filters [109] and Golomb coded sets [132]. However, these structures do not handle range queries nor do they take advantage of the data distribution.

We note that theoretical results from [66] show that in the worst case, a data structure that can answer a range query of size up to R with a false positive rate of FPR needs to store $\Omega(\log_2(R) + \log_2(1/\text{FPR}))$ bits per key. Their lower bound suggests looking for structures that may not have worst case guarantees, which can obtain better performance in practical scenarios by focusing on the data and query

distributions.

Learning Enhanced Data Structures and Algorithms: We utilize the incorporation of learned models into traditional structures and algorithms. This technique has been applied for indexing [46, 62, 80, 72, 56, 33] and sorting [88, 90]. However, while both like SNARF leverage a model of the eCDF, those structures cannot be used as range filters unless they store all keys, which would not make them space efficient (e.g., one should consider how a B-Tree could be used as a space efficient Range or Bloom filter, which is equally hard/impossible). Learning-enhanced approaches also have been proposed for Bloom filters design [101, 112, 156] but again they are not designed for range queries. Moreover, existing ml-enhanced bloom filters are actually based on classification models, not empirical CDFs.

LSM based Key Value Stores: An important application of filter structures are key-value store data systems [75] based on log-structured merge trees (LSM) [121]. Numerous workloads served by key-value stores (social media, networking, security) include heavy portions of both point and range queries. LSM-based key-value systems store data in multiple immutable files on a disk. Retrieving a particular item or set of items in a particular range leads to multiple expensive I/O's to look up the items in these immutable files. In many settings, the item may not be present in the files, leading to unnecessary I/O's that degrade total query response time. Modern LSM-based key-value systems have extended the basic LSM structure with in-memory filters to address this problem: if a query has no corresponding item, the filter most likely returns false and saves expensive I/O.

Adaptive Range Filter: The Adaptive Range Filter (ARF) [2] uses a binary trie to encode integer key spaces. ARF only stores a number of prefixes of the key set and range queries are then processed by searching the trie for any prefixes of the given range. If a leaf node results in a false-positive, then it is extended until it would no longer do so and, if needed, an old branch is pruned to maintain memory constraints. ARF is not a space efficient data structure for many workloads and in some cases 1300× bigger than SuRF while having a worst FPR (see [161]). Hence, we do not consider ARF further here.

SuRF: The Succinct Range Filter (SuRF) [161] utilizes a compact trie-like data structure which can filter arbitrary range queries. The trie is culled at certain prefix lengths. The basic version of SuRF stores minimum-length prefixes such that all keys can be uniquely represented and identified. Other SuRF variants store additional information such as hash bits of the keys (SuRF-Hash) or extra bits of the key suffixes (SuRF-Real). A weakness of SuRF is that for point queries, SuRF can provide up to 100x worse FPR compared to Bloom filter variants such as Cuckoo filters under the same space budget.

Rosetta: Rosetta utilizes a different approach that performs better for point queries, correlated workloads, and very short ranges. Rosetta essentially uses a Bloom filter for range queries along with the help of a hierarchy of prefix Bloom filters that form an implicit segment tree. Empirically, this design helps Rosetta achieve little to no degradation for point queries compared to Bloom filters. On the other hand, the FPR for Rosetta, while good for small ranges, becomes worse with increasing range query size. For large range queries, Rosetta provides almost no filtering.

LSM Range Queries: ElasticBF [98] proposes a method to adapt Bloom filters in LSMs to query workload. The idea is to use larger filters for hot regions which can be used with SNARF or any other range filter as well. BloomRF [137] is another proposed filter which uses the idea of implicit segment tree with hierarchy of filters similar to Rosetta. It also suffers from FPR degradation with range size like Rosetta. Orthogonal to our approach, REMIX [162] focuses on making range queries faster by creating an alternative path on top of an LSM tree that maintains range indexing info.

Compression Schemes: SNARF needs to compress a sparse bitmap of size nK with n one bits. Assuming a uniform random spread of the one bits, the asymptotic information theoretic lower bound for lossless compression of such a bit array would be $\log_2(K)$ bits per key ($\log_2(K) - O((\log nK)/n)$ bits per key to be precise). Golomb Coding and Elias Fano Coding are near optimal coding schemes as they use at most 2 bits per key over this lower bound ($2n+n\log_2(K)$). Other compression techniques such as WAH[36], CONCISE[28], and Roaring[25] are less space efficient for our particular task, though they can be somewhat faster, so if speed was a concern they could be

substituted for our compression approach.

To support fast writes, a LSM-based key-value stores inserts data into an in-memory buffer. When the buffer grows beyond a predetermined threshold, it is flushed to disk and periodically merged with the old data. This process repeats recursively, effectively creating a multi-level tree structure where every subsequent level contains older data. Every level may contain one or more runs (merged data) depending on how often merges happen. The size ratio (i.e., how much bigger every level is compared to the previous one) defines how deep and wide the tree structure grows, and also affects the frequency of merging new data with old data. As the capacity of levels increases exponentially, the number of levels is logarithmic with respect to the number of times the buffer has been flushed. To support efficient point reads, LSM-trees use in-memory Bloom filters to determine key membership within each persistent run. Each disk page of every run is covered by fence pointers in-memory (with min-max information). Collectively Bloom filters and fence pointers help reduce the cost of point queries to at most one I/O per run by sacrificing some memory and CPU cost. Nit: Technically not 100% correct, at least not for RocksDB where L0 is kept unsorted (files may overlap). Wait... do you mean run or level here? Do we need this much info about LSMs and does it belong here? Maybe just a brief desctiption in the eval section?

Range queries are increasingly important to modern applications, as social web application conversations, distributed key-value storage replication, statistics aggregation for time series workloads, and even SQL table accesses as tablename prefixed key requests are all use cases that derive richer functionality from building atop of key-value range queries. What is a 'tablename prefixed key request'? While LSM based key-value stores support efficient writes and point queries, they suffer with range queries. This is because we cannot rule out reading any data blocks of the target key range across all levels of the tree. Range queries can be long or short based on selectivity. The I/O cost of a long range query emanates mainly from accessing the last level of the tree because this level is exponentially larger than the rest, whereas the I/O cost of short range queries is (almost) equally distributed across all levels. I don't understand this paragraph. Don't fence pointers already ensure that the correct files

are read? Don't range filters only help in cases where the qualifying files (according to fence pointers) are false positives?

A recent trend in research has also seen the incorporation of learned models into traditional structures and algorithms. This technique has also been applied to Bloom filters [101] and has been iterated upon, resulting in improved techniques such as sandwiching [112] and partitioning [156]. These filters can be more memory efficient in situations where the data set has characteristics that can be learned, but they suffer from the same issue with regards to range queries. [66] shows that to robustly answer a range query of size R with a false positive rate of f one needs to store $(log_2(R) + log_2(1/f))$ bits per key.

One means of addressing approximate range emptiness queries is to encode regions of the key space rather than individual members. For example, a prefix Bloom filter encodes the key space in fixed size regions by hashing the prefix of each region containing at least one member of the key set. This method works well for range queries of a known size, but ranges that are significantly smaller or larger than the encoded regions will not be filtered as reliably. As such, multiple projects have designed range filters that use multiple prefix filters or search trees to encode regions of different sizes.

The Adaptive Range Filter (ARF) [2] makes use of a binary trie to encode integer key spaces. The entire trie representing the key set is typically too large to fit in memory, so an ARF only stores a number of prefixes of the key set. Range queries are then processed by searching the trie for any prefixes of the given range. If a prefix is found, then the query is considered positive. An ARF regularly changes which prefixes it stores in response to queries. If a leaf node results in a false-positive, then it is extended until it would no longer do so and, if needed, an old branch is pruned to maintain memory constraints.

The Succinct Range Filter (SuRF) [161] is another trie based approach. While it does not change its shape in response to queries, it encodes prefixes of the key set as a fast succinct trie (FST). FSTs efficiently encode nodes representing eight prefix bits and support constant search time by using both dense and sparse node encodings.

This allows for more of the trie to fit within memory as well as fast queries independent of range size. Since the whole trie is still too large, SuRF's default pruning strategy is to store the minimal prefix that uniquely identifies each key within the key set. Additional memory can be spent to extend these prefixes or to store bits of the key hashes at each leaf node to improve point query performance.

Despite adapting to queries and using efficient encodings, these trie based methods still have trouble with more fine grained queries. Even though they can store prefixes of different lengths, the long prefixes necessary to filter fine grained queries are still quite expensive to encode in a trie. This means that only a limited portion of the key space can be encoded to support fine grained queries.

Alternatively, Rosetta [100] is another range filter aimed at addressing these more fine grained range queries. To do so, Rosetta makes use of multiple prefix Bloom filters with different prefix lengths. Unlike the tries, this allows memory to be allocated to longer prefix lengths at the same cost as any other prefix length. These prefix filters implicitly encode levels of the same binary trie used by ARFs. Whether a node is present in the trie is determined by querying the corresponding prefix in the Bloom filter with the respective prefix length. To perform well on finer grained queries, Rosetta allocates most of its memory to the bottom levels of the trie, i.e. the Bloom filters with the longest prefix lengths; however, this again results in worse performance for larger ranges.

There exist structures which can provide faster range queries than the encoding schemes we discussed above. Typical Rank Select structures provide efficient predecessor queries. In order to check if there is a one bit in the bit range($[a, b]$) , one can use the predecessor operation $pred(b + 1)$ to get the position of the first bit at or before position b. If $pred(b + 1) < a$ holds, then there cannot be a set bit in the range($[a, b]$) since the first one bit before $b + 1$ is more than $b - a$ bits away. The predecessor operation is therefore sufficient to answer range queries over a bit vector. A lot of work has focused on improving the efficiency of Rank-Select structures [126, 64**?** , 119].

I just skimmed through it. We probably have to shorten this section. I am also wondering, if we should move it up to Section 2 as background and related work.

3.6 Experimental Evaluation

We now demonstrate that SNARF can bring more than one order of magnitude improvement when compared to state-of-the-art filters. We evaluate SNARF both as a standalone filter as well as part of RocksDB. [9]

3.6.1 Standalone Analysis

Our experiments comparing SNARF against other baselines aim to support the following key claims:

- SNARF offers a better FPR-space tradeoff curve than other baselines on various synthetic and real world datasets/workloads.
- The FPR provided by SNARF is robust to increasing query range sizes as well as skew in query workload (certain part of data queried more often).
- SNARF performance drops with correlation (as discussed in Sec. 3.4.1) resulting in Rosetta being better for very short and highly correlated range queries.
- SNARF has a reasonable construction time and its query response time can be tuned as needed. SNARF with Elias Fano encoding has a faster query response time than with Golomb Coding at a slightly higher space cost.
- SNARF supports updates at reasonable throughput.

We now provide experiment details.

Baselines:We evaluate SNARF against three other baselines:

SuRF: We use the SuRF implementation from [27] with real suffixes as they provided the best performance. [10]

Rosetta: We use the original Rosetta implementation [100].

[9] For our experimental design, we follow the evaluation setup as done in SuRF and Rosetta as much as possible.

[10] Note, SuRF has a limited range of operation as the implementation starts with minimum of 10 bits per key (0 bits as the suffix length).

Cuckoo Filter: For our point queries, we compare against the Cuckoo Filter implementation from [26] in the semi-sorted setting as it achieved the best FPR-space tradeoff.

Datasets: For our experiments, we build a filter on 100 million keys chosen from the following datasets.[11] We use two synthetic datasets and three real world datasets from [104]:

Uniform Random: Keys are generated uniformly at random in the range $[0, 2^{50}]$.

Normal: Keys are generated from normal distribution ($N(\mu = 100, \sigma = 20)$) and are linearly scaled to range $[0, 2^{50}]$.

wiki: Keys represent the time an edit was made on Wikipedia.

osm: cell IDs from Open Street Map representing a location.

fb: unique Facebook user IDs [157].

Workload: We use 100 million queries for our experiments. The queries are of the type [*left*,*left*+*range_ size*]. If *range_ size*=0, then the query is a point query. We first generate the left endpoint(*left*) of the range query from a certain distribution and then the right endpoint of the query is calculated by adding the left endpoint and the *range_ size*. The range query workloads use a range size of 256 while the mixed-query workloads use range sizes of 0, 16, 64 and 256 in equal proportion. We generate the left endpoint(*left*) of the queries in following manner:

Uniform Random: left endpoint chosen uniformly at random in the range $[0, 2^{50}]$.

Exponential: We use an exponential distribution($p(x) = \lambda e^{-\lambda x}; \lambda = 10$) which results in certain part of the data being queried more often. We then scale them to range $[0, 2^{50}]$.

Correlated: This distribution generates queries which are close to the keys. A key is chosen uniformly at random from the dataset and then left endpoint is chosen

[11]We evaluate on integer keys but would also work for floats. Floats are numerical keys, the current CDF model for SNARF works for them. We expect minimal change in the performance of SNARF for floating point values.

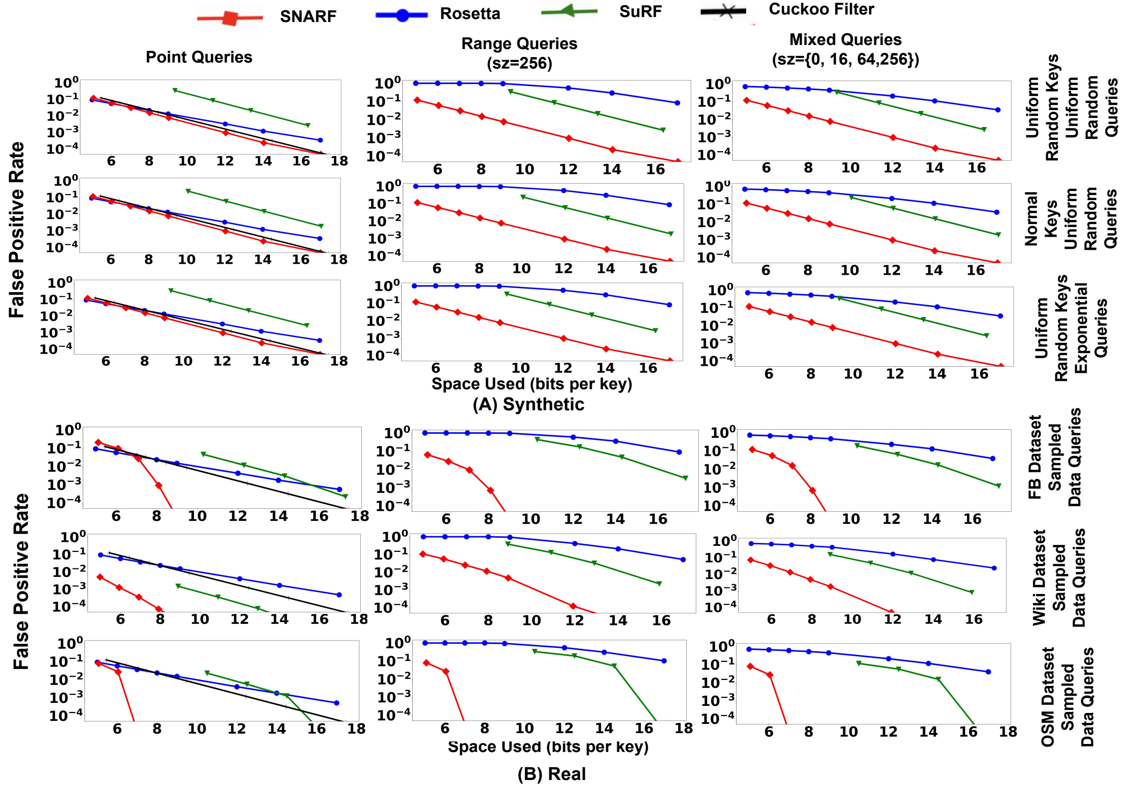


Figure 3-5: FPR vs Space Used(in bits per key) by various filters. Each subfigure shows the space-FPR tradeoff for a (A) synthetic (B) real dataset and workload distribution and for a particular range query type (point, range query of size 256 and mixed query workload of size 0,16,64,256).

uniformly at random from [key, key+$2^{30*(1-corr_degree)}$]. Higher $corr_degree$ implies increased proximity between keys and queries, so that $corr_degree = 1$ generates extremely correlated queries (left end point being $key + 1$) whereas $corr_degree = 0$ generates queries independent of the key value.

Sampled Data: This is used to generate range queries for real world datasets (as previously done in SuRF). We first divide the dataset into two equally sized parts by choosing keys uniformly at random. A filter is built on one half of the dataset and the other half is used as the left endpoints for queries in the respective workload.

SNARF parameters: The CDF model uses $(N/1000)$ linear models unless stated otherwise. By default, we choose $\beta = 100$ and thus divide the bit array into $(N/100)$ equally sized segments. We use Golomb coding for SNARF unless specified.

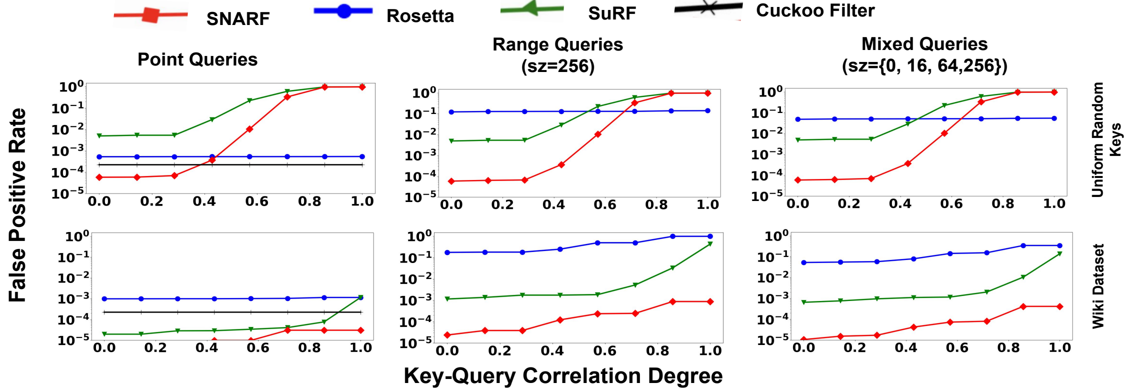


Figure 3-6: FPR vs Key-Query Correlation Degree of the queries on uniformly random/wiki keys. With increasing key-query correlation, SNARF and SuRF become worse and Rosetta turns out to be the better filter for very short and highly correlated range queries.

FPR vs Space Tradeoff for Synthetic Dataset/Workloads:

In Fig.3-5(A), each subfigure corresponds to a particular key and query distribution along with a particular query workload. Each subfigure shows the space used by the baselines in bits per key and the FPR achieved by them on the corresponding query workload. For point queries, SNARF achieves performance similar to Cuckoo filters for all cases. For range queries, SNARF consistently has a better Pareto curve than all other baselines. When using 16 bits per key, SNARF and SuRF provide false positive rates of 6.2×10^{-5} and 1.1×10^{-3}, respectively. Rosetta is competitive for point queries but its performance degrades as query range size increases.

Even with exponentially distributed data, SNARF maintains its performance as the CDF model can capture this skew in data distribution. As discussed in Sec.3.3, mapping the keys evenly across the bit array results in a robust false positive rate and consistent performance across different skewed query distributions.

FPR vs Space Used Tradeoff for Real Dataset/Workloads:

In Fig.3-5(B), each subfigure corresponds to a particular dataset along with a particular query workload. Each dataset is divided into two equal parts. One part forms the set of keys and the other half forms the left endpoint of the query. The right endpoint is decided by the range query size.

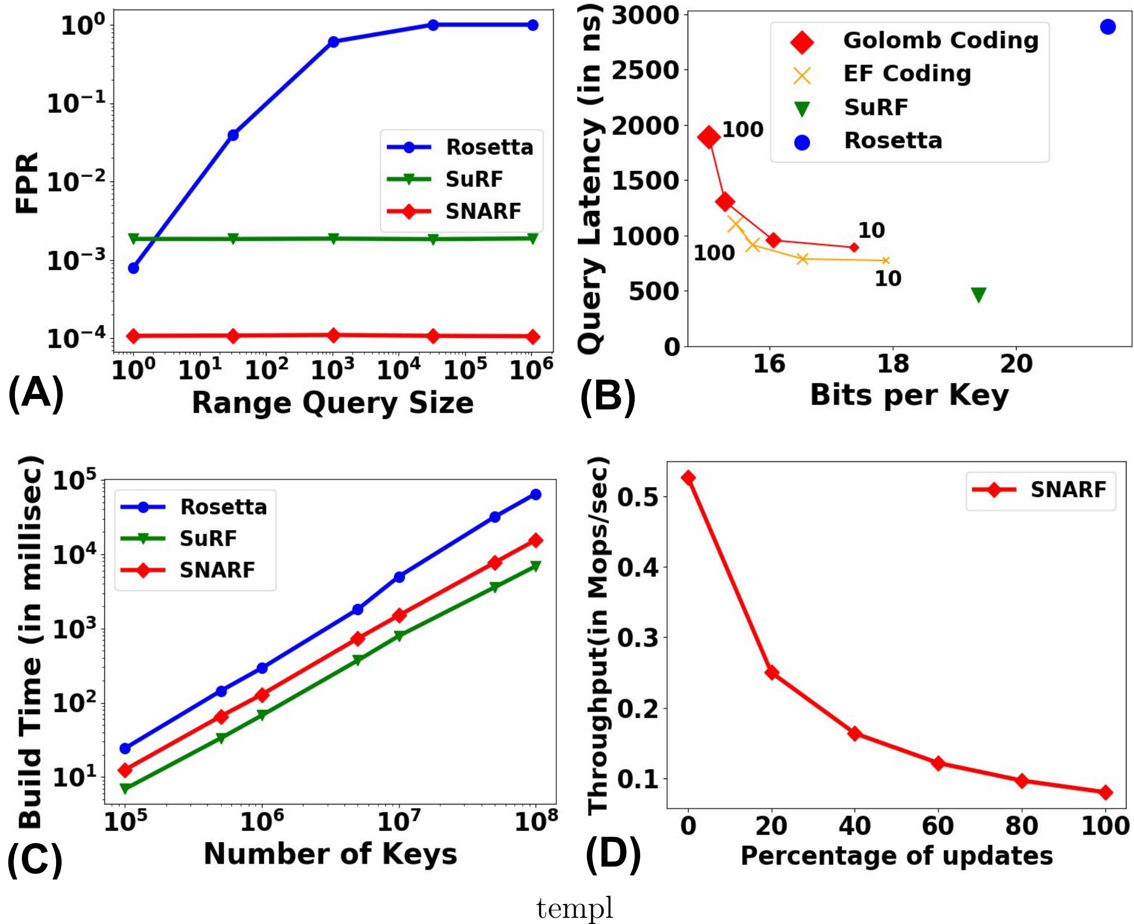
SNARF has a better Pareto curve than other baselines for all cases. SNARF is able to perform particularly well on real-world datasets due to certain patterns present in them. A common pattern we observed in our real-world datasets is that they have large empty contiguous ranges; for example, S={10,78,95,10045,10052,10089,30011,.....}, where the sorted keys suddenly jump by large amounts. While we do not have a clear global reason for such behavior, it is natural for settings such as when the set is a collection of numerical IDs; different ID subranges may be assigned by different entities. Both SNARF and SuRF effectively model large empty ranges in a way that is both succinct and avoids false positives.

In some cases, while keys may be from a large domain, they may be concentrated in a small range. For example, the keys may lie in the domain $[0, 2^{32})$ but all appear in the small range $[2^{10}, 2^{12}]$. The modelling step of SNARF automatically takes advantage of this type of pattern to benefit performance[12]. For most cases, SNARF is able to achieve a low FPR (below 10^{-4}) using less than 10 bits per key. For the *osm* dataset, SuRF also achieves a FPR below 10^{-4} but still uses more memory ($\approx 15 - 16$ bits per key). Similar to our previous experiments on synthetic data, Rosetta is competitive for point queries but its performance degrades as query range size increases.

Correlated Workload:

As discussed in Sec.3.4.1, the correlated workloads are when the query endpoint is close to a key, which is more likely to lead to a false positive in SNARF and SuRF. In Fig.3-6, we show the FPR vs key-query correlation degree tradeoff for various baselines for a fixed memory budget of 15 bits per key. Higher the key-query correlation degree closer the queries are to the keys. As expected, FPR of both SNARF and SuRF degrades with increasing correlation. Both SNARF and SuRF provide virtually no filtering for uniform dataset when workload is highly correlated. On the other hand, Rosetta is unaffected by this correlation and performs the best for very short range queries and highly correlated workloads.

[12]This is similar to the case when $z < nK$ and all values are mapped to distinct bit positions leading to no false positives.



templ

Figure 3-7: (A) FPR with increasing range query size for fixed space budget. (B) Filter Latency (in ns) against space used (bits per key) (C) Build Time(in millisecs) with increasing number of keys. (D) Filter Throughput as we vary the percentage of updates in the workload.

FPR vs Range Size:

In Fig.3-7(A), we vary the range query size from 1 to 10^6 and report the FPR of various range filters under a memory budget of 15 bits per key. We use uniformly randomly distributed keys and workloads for this experiment. As discussed in Sec.3.3, the FPR of SNARF stays constant with the range query size. SuRF also maintains its FPR with increasing range query size but has a 17x worse FPR than SNARF. Rosetta becomes worse with increasing range size and provides almost no filtering for range sizes greater than 1000.

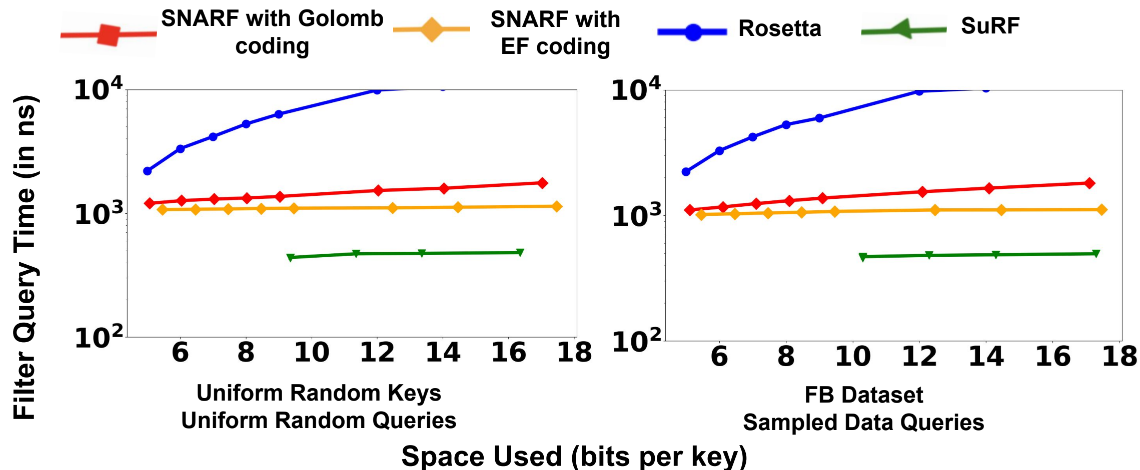


Figure 3-8: Filter Query Time (in ns) vs Space used by various filters across various datasets and workloads for mixed queries.
[short]long

Filter Query Latency vs Space Used:

In Fig.3-8, we show the query latency of various filters with increasing filter sizes for uniform random and FB datasets for mixed range queries. We skip other datasets/workloads as we observed similar trends for them. For this experiment, we fix the size of the dataset to 100 million keys and vary the filter parameters that control its size. In both of the subfigures, both variants of SNARF are slower than SuRF but faster than Rosetta. SNARF with Elias Fano encoding is consistently faster than SNARF with Golomb coding. This is because Golomb coding (a form of delta coding) requires decoding of the first key and the following delta values to retrieve a key, which is not the case for Elias Fano coding. For SNARF, filter query time increases slightly with increasing filter size. This is because as the filter size increases, the model size remains constant but the encoded bit array size increases, so SNARF then has to parse more data to decode the bit array. The filter query latency increases drastically for Rosetta as its filter size increases, as larger internal Bloom filters mean Rosetta has to perform a greater number of random accesses.

Effect of Bit Array Division on Space and Query Latency:

As discussed in Sec.3.2.3, for SNARF we can improve the query latency by reducing the segment size. Recall we use small segments of size βK in the bit array, and using smaller β can improve latency at the cost of extra space overhead. The overhead arises because when we have a larger number of segments in the bit array there is more associated metadata. Fig.3-7(B) shows the query latency and the space used by the various baselines to achieve a FPR of 2^{-13} on uniform random keys and uniform randomly generated mixed sized queries. We show multiple configurations for SNARF with β values 10, 20, 50, and 100 (increasing marker size representing larger β values). The results show that with decreasing β we get better query latency. Elias Fano coding is faster than Golomb coding for the same number of segments. By varying β, Golomb coding and EF coding with SNARF are able to achieve a query latency of 890 ns and 746 ns, respectively. SuRF is the fastest baseline with latency of 480ns, but uses around 19.4 bits per key.

Build Time:

In Fig.3-7(C), we vary the number of the keys from 10^5 to 10^8 and report the build times of various range filters. We use a uniformly random distribution for the keys. The build times of all the filters grow linearly with the number of keys. The build time for learned range filters is around 5x faster than Rosetta and around 2x slower than SuRF. Depending on the application, filter construction might play a more or less important role. For example, for LSM trees, filter construction only plays a minor role as part of the merge phase as shown in Sec.3.6.2.

Updates:

In Fig.3-7(D), we vary the percentage of updates(50% insertions and 50% deletions) in the query workload (the rest of the workload is range queries) and report the throughput. SuRF and Rosetta do not support both inserts and deletes, so we only analyze SNARF here. We use the SNARF variant with duplication in order to support

deletes. We use a uniform random distribution for the keys. The workload contains 1 million operations overall and is also uniformly randomly distributed. Since, the updates do not change the distribution of the data, the FPR stays constant. On average an update takes around 12k ns whereas a range query takes around 1898 ns. The throughput of the filter decreases with increase in proportion of updates as updates are slower than range queries.

3.6.2 RocksDB Experiments

Our experiments on RocksDB integrated with SNARF aim to support the following key claims:

- RocksDB with SNARF offers better read performance than other baselines on various synthetic and real world datasets and workloads.
- SNARF's as well as other filters impact reduces as the proportion of empty range queries in the workload decreases. This leads to SNARF's performance improvement over other filters to reduce as well.
- In RocksDB with SNARF, read performance drops with correlation (as discussed in Sec. 3.4.1) resulting in Rosetta being better for very short (range size less than 16) and highly correlated range queries.
- SNARF adds little overhead to RocksDB
- SNARF improves end-to-end performance of RocksDB for a typical read-write workload.

Integration with RocksDB: We use a RocksDB integration and workload generation setup identical to that of Rosetta [100]. We utilized an API of filter functionalities such as populating, querying, serializing, and deserializing the filter to integrate SNARF. RocksDB stores its data in multiple immutable tables called SST (Sorted String Tables). A SNARF instance is created for each SST file. We store the filter on disk as a character array and the process of converting the filter to char array is called

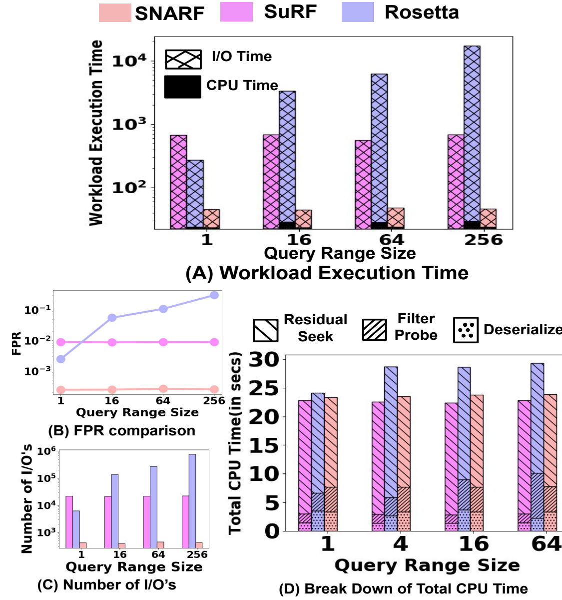


Figure 3-9: SNARF outperforms other baselines when fully integrated in RocksDB.

serialization. In order to use the filter we need to read it to memory from the disk and deserialize it[13]. We enable the block cache and allow the caching of filters.[14]

Implementation Overview of a Range Query: For a range query $[p, q]$, RocksDB probes filter instances of all levels for existence of keys within this range. If all filter instances return negative, an empty result is returned. If one or more filters return positive, RocksDB seeks the lower end (p) incurring an I/O. When RocksDB get a valid pointer, it reads data until q is reached and incurs as much I/O's needed to reach q.

Setup and Workloads: We use 14 bits per key for all the filter baselines(as previously done in SuRF)[15]. We first populate RocksDB with 50 million 64-bit keys from a distribution and 512 byte values. Each experiment has a description of the workload. After population, we run the workload on this populated RocksDB instance. Total execution time of this workload is usually the metric of interest.

[13]To reduce the deserialization overhead we maintain a dictionary that has the deserialized bits for each filter instance and its corresponding SST similar to [100]

[14]*cache_index_and_filter_blocks*=true. We also ensure that the fence pointers and filter blocks have a higher priority than data blocks when block cache is used *cache_index_and_filter_blocks_with_high_priority*=true, *pin_l0_filter_and_index_blocks_in_cache*=true.

[15]14 bits per key allows reasonable performance with fpr below 10% for all filters

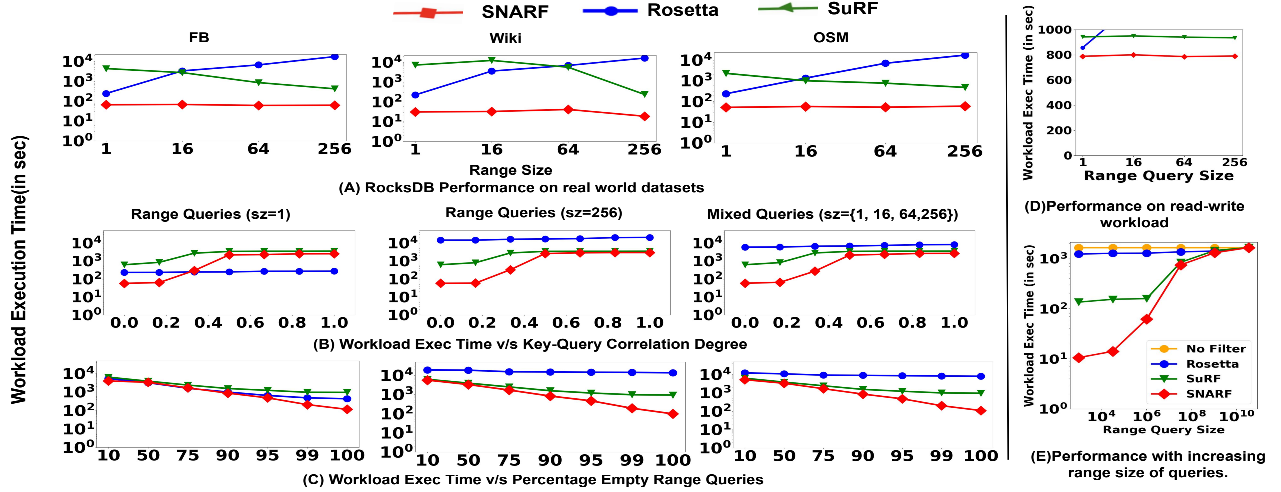


Figure 3-10: Workload Execution time in RocksDB for (A) real world datasets/workloads (B) correlated workloads and (C) varying percentage of empty queries (D) read-write workload (E) varying rnage query sizes

We use uniform random distribution for keys/workload generation by default and we have 100k queries in a workload as default. We used the same distributions of dataset and workload mentioned in Sec.3.6.1. The workloads are primarily read only to highlight the impact of filters, but we also have a few experiments with a mixture of reads and writes. Each workload is run with read queries of various range sizes (1, 16, 64, 256).

SNARF improves RocksDB Performance:

For this experiment, we generate YCSB key-value workloads that are variations of Workload E, a majority range scan workload modeling a web application use case [31]. The quality of the filter is best judged with empty range queries as filters enhance performance by identifying empty queries for which an unnecessary seek can be avoided. Thus, we compose our workload with $100,000$ empty range queries.

Fig.3-9(A) shows the workload execution time of various baselines. The workload execution time consists of two parts, time spent by the CPU and time spent on I/O. We observe that I/O time dominates the CPU time. SNARF's workload execution time is consistently one order of magnitude less than the other baselines. SNARF has a better FPR than SuRF and Rosetta leading to fewer I/O's and hence lower

workload execution time. As shown in Fig.3-9(B), SuRF has a FPR 40x worse than that of SNARF across all range sizes. Rosetta's FPR becomes worse with increasing range size. Worse FPR leads to more block I/O's as shown in Fig.3-9(C). In summary, Rosetta and SuRF have significantly high I/O time due to their worse FPR.

SNARF adds little CPU overhead In the previous experiment, we further break down the total CPU time for various baselines in Fig.3-9(D). The CPU time is further divided into deserialization time, filter probing time, and residual seek time. The residual seek time is the time taken for routine jobs performed by RocksDB iterators – looking for checksum mismatch and I/O errors; going forward and backward over the data, filters and fence pointers; and creating and managing database snapshots for each query. The filter probe time is time taken to probe the filters and deserialization time is the time taken for filter deserialization. The filter probe time accounts for at most 20% of the total CPU time for even the slowest filter (Rosetta). Residual seek time accounts for the dominant portion of the CPU time. Thus, a CPU intensive filter does not affect the performance of RocksDB much.

SNARF improves RocksDB Performance on real world datasets For this experiment, we populate RocksDB with 50 million keys from real world datasets and use sampled-data workload consisting of 100k empty range queries. Fig.3-10(A) shows the workload execution latency of this workload in RocksDB. SNARF exhibits a lower workload latency than other baselines for all three datasets. Same as previous experiment, this is due to the better FPR that SNARF delivers compared to other filters.

As range size increases, SNARF improves RocksDB Performance for correlated workloads As discussed in Sec.3.4.1, SNARF and SuRF become worse with increased correlation between queries and keys. For this experiment, we use a correlated workload consisting of 100k empty range queries. In Fig.3-10(B), we vary the key-query correlation degree of the queries and measure the workload execution latency. The execution time of SNARF and SuRF increases with correlation but not beyond a certain level. This is because even if a query is highly correlated to a key in a particular SST file, SNARF and SuRF are still useful for the rest of the SST files.

Rosetta is the better filter for range size equal to one and a highly correlated workload otherwise SNARF is the better filter.

SNARF performance for mixed workload (empty and non-empty range queries):

Here, we measure the workload execution latency on a mixed read-only workload of empty and non-empty range queries by varying the percentage of empty range queries from 10 to 100. As shown in Fig.3-10(C), the workload execution time of all filters decreases with an increase in the proportion of empty range queries. This is because filters are more effective on empty queries than non empty queries. Notice, even with majority non-empty workload filters are still useful. This is because non-empty range queries will return a true positive for one SST but other SST's might still return a false positive leading to additional unnecessary scans. The decrease in execution time is faster for SNARF than other baselines because SNARF has better FPR than others and thus, is more effective in reducing unnecessary I/O's.

SNARF performance for read-write mixed workload:

In order to simulate real working of RocksDB, we used a majority write workload (only 1 percent reads) with 10 million operations similar to YCSB workload A(majority updates). Read and writes are performed in an interleaved manner. We first start with a RocksDB instance that already has 50 million uniform randomly distributed keys . Reads and writes are generated using the uniform random distribution. Each write operation is a point write which inserts a unique key into the RocksDB instance with a corresponding randomly generated value. All the read queries are empty range queries and we evaluate 4 different workloads for read queries with 4 different range sizes: 1, 16, 64 and 256.

Writing keys to RocksDB leads to compaction and creation of new SST files. Creation of new SST files involves constructing the filter and thus filter construction time gets accounted for in the overall execution time. While performing reads, the query response time of the filter gets accounted for in the execution time. Thus, this

experiment evaluates the end-to-end filter performance as it accounts for reduced I/Os due to filtering, filter query response time and filter construction time. In Fig.3-10(D), we show the workload execution time of the workload for various range sizes. Owing to its lower FPR, SNARF has a lower workload execution latency than SuRF and Rosetta. SNARF's slightly slower filter query time and construction time compared to SuRF is offset by gains produced in lower I/Os.

SNARF impact with increasing range query size:

In Figure 3-10(E), we show the workload execution time as we increase query range size for uniformly randomly generated keys and workloads. The impact of filters decreases with increasing range size. For range sizes around $\approx 10^3 - 10^4$ most queries are empty; accordingly, filters have a large impact and here SNARF outperforms other filters by an order of magnitude. For range sizes around $\approx 10^7 - 10^8$ most queries are non-empty, touching a few SSTs, and filter have less impact. For range sizes around $\approx 10^9 - 10^{10}$ most queries touch most SSTs and filters have negligible impact.

Chapter 4

Learned Hash Tables

4.1 Introduction

Hashing and hashing-based algorithms and data structures find countless applications throughout computer science, such as in machine learning, computer graphics, bioinformatics, and compilers (e.g., [83, 102, 15, 115]). Hashing is also a fundamental operation in database management (e.g., [10, 136, 76]), including playing a key role in the implementation of numerous core database data structures and algorithms (e.g., indexes [76, 78], filters [74], joins [10], partitioning [130], and aggregation [49]). Due to its numerous applications, considerable research efforts have focused on introducing efficient hashing functions (e.g., [136, 124, 96, 102]).

Traditionally, hash functions aim to mimic a function that maps a key to a random value in a specified output range. For typical cases where the size of the output range is linear in the number of keys, this random assignment results in colliding keys. A collision occurs when multiple keys get mapped to the same output value. A typical hash index approach allocates a number of fixed size slots (the number of slots generally being a constant times the expected number of keys) and maps incoming keys into these slots using a hash functions. The ideal case for indexes would have no keys collide, so each key goes to its own separate slot. This would make key lookups and updates faster, as one would simply check the corresponding slot for the key. With truly random hash functions, collisions are unavoidable, and one can readily

calculate the expected number of collisions given number of slots and keys [114].

Naturally, there are many well-known schemes like chaining, probing, and cuckoo hashing to handle collisions. As the name suggests, chaining handles collisions by creating a chain of colliding keys. Probing checks neighboring slots to find an empty slot to place the key. Cuckoo hashing handles collisions by using multiple hash functions to provide alternate slots for colliding keys. For each of these schemes, more collisions reduces their performance.

Another approach to build hash indexes is to use *perfect* hash functions instead of truly random hash functions. Perfect hash functions have no collisions; however, they must be specially constructed for a given data set, and have other costs in storage and computation time.

In recent years, several works have utilized the idea of using machine learning to improve the performance of many database components (e.g., [107, 79, 103]) and basic data structures (e.g., [87, 43, 97, 47]). By using machine learning to explicitly capture trends in the underlying data, these methods can aim for instance-optimal performance. For example, in a recent benchmarking study [105], it has been shown that learned index structures (e.g., RMI [87], RadixSpline [81]), which employ CDF-based learned models, can outperform traditional indexes on practical workloads.

As one direction in this line of research, it was suggested in [87] that such learned models can be used to obtain an efficient hash function with fewer collisions. They also provided some empirical evidence that a hash index with learned model as the hash function can have better performance than using a truly random hash function. What is unclear, however, is when such learned models are effective in replacing existing hash functions in applications. At one end, *traditional* hash functions [52, 154] are fast to compute, but suffer from collisions [149] that can reduce query performance. On the other hand, *perfect* hash functions [102] avoid collisions, but are difficult to construct [93], and are not scalable [38], in the sense that the size of the function representation grows with the size of the input data. As an alternative, learned models can potentially provide a better tradeoff between computation and collisions. If the model learns a good approximation of the empirical CDF of the input keys, we may

achieve few collisions; and if the data allows a compact learned model, we may achieve a model size independent of or very slowly growing with the input data size.

Surprisingly, though, we are not aware of a thorough experimental study examining the performance of learned models against both traditional and perfect hashing in query processing operations like indexing and joins. We aim to remedy that here. We make the following contributions:

- We provide an analysis of the factors affecting collisions for learned models, helping us to identify situations where they can have fewer collisions than traditional hash functions.
- We perform an extensive benchmarking study for traditional, perfect, and learned model hash functions. We benchmark them through three different applications: hash table probing/inserting, range querying, and joins. We test using multiple synthetic and real-world datasets.
- Through the empirical study and analysis we find useful insights on when to use learned models instead of traditional and perfect hashing in various database applications.
- We provide a unified open-source implementation for the baselines used in our experiments.

As a summary, we gained the following key insights based on our collisions analysis and experimental benchmarking:

- The performance of learned models depends on the input keys, and specifically on the distribution of gaps between consecutive keys in the sorted list of inputs. Generally, evenly spaced gaps are favourable for learned models.
- The computation throughput of learned models decreases with model size and is on par with traditional hash functions for small model sizes.
- Collision reduction due to learned models translates to improved hash table insert and probe throughput. The gain varies across different hashing schemes.

- Using order-preserving learned models as hash functions provides additional advantage of supporting range queries. Such learned hash tables provide better probe throughput for majority point query workloads than other baselines.
- We show that this type of hash table can reduce the running time of non-partitioned hash join [92, 153] with at least 28% over other baselines.
- In many other cases, however, such as with data from typical distributions (like the normal distribution), we do not see gains using learned models.

Based on our insights, we recommend special handling of datasets with evenly spaced gaps by using simple learned models or other suitable techniques, particularly for hash based indexing and joins.

4.2 Traditional Hash Functions

A uniform hash function $h(x) : X \mapsto U$ attempts to map arbitrary inputs to independent and identically distributed (i.i.d.) uniform random outputs. Obtaining true randomness is not feasible in practice [83]. However, state-of-the-art hash functions appear to come reasonably close to imitating true randomness in many practical settings [154, 122]. The extent to which a hash function avoids collisions, i.e., instances where two distinct inputs map to the same output, is often referred to as the its' quality. There is a seemingly endless supply of different proposed hash functions to choose from [154]. Here, we briefly give a background on some of the well-known functions that we study in the paper.

Multiplicative Hashing (MultiplyPrime). This method is prominently described by Donald Knuth [83] as a family of hash functions with great properties for practical applications. He explicitly advertises their non uniform random properties, i.e., sensitivity to the data distribution, as a strength [83]. Let A be a constant, relatively prime 2^w with w being the machine word size. Then, the following function produces outputs in $[0, M)$.

$$h(x) = \left\lfloor M \cdot \left(\left(\frac{A}{2^w} x \right) \mod 1 \right) \right\rfloor$$

The trick to make this efficient is to avoid fractional computations by shifting the entire calculation by w, i.e., to multiply with $\frac{A}{2^w} \ll w = A$ instead of the complex decimal computation: $h(x) = \left\lfloor \frac{M}{2^w} \cdot (Ax \mod 2^w) \right\rfloor$. Neatly, this will allow us to get rid of the modulo since most physical machines with a word size w will naturally compute everything $\mod 2^w$. According to Knuth, M should be some power of the machine's radix [83] to ensure that we are including the more significant bits in the final result.

Fibonacci Hashing (FibonacciPrime). It is an instance of multiplicative hashing, choosing $\frac{A}{w} = \Phi^{-1} = \left(\sqrt{5} - 1\right)/2$ based on the golden ratio. It promises to inherit Φ^{-1}'s neat scattering characteristics, i.e., that each added consecutive element falls in the largest remaining interval, dividing it by the golden ratio [83, 145, 7, 164, 146]. As in multiplicative hashing, we implement Fibonacci hashing using the integer multiplication trick. However, this time we choose $C = \Phi \cdot 2^w$ with w as the machine word size. Some implementations also round C to the next closest prime.

Murmur Hashing (Murmur). Murmur is a family of simple and fast hash functions developed by Austin Appleby [6, 5], and has been studied extensively in previous works (e.g., [3, 136]). Its name is derived from the original idea for its implementation, i.e., repeatedly applying **mu**ltiply and **r**otate instructions to imitate true randomness. Ultimately, however, it ended up being implemented as a sequence of multiply, shift, and xor operations. In particular, its 64-bits finalizer merely consists of three xors, three shifts, and two multiplications [5, 136].

XXHash. It is a widely used open-source uniform hash function with support for many programming languages [29]. It targets RAM speed limits for hashing large enough blobs of data, all while promising decent performance on small inputs.

AquaHash. It is a uniform hash function that utilizes Advanced Encryption Standard (AES) intrinsics [70], i.e., AES encryption primitives implemented in hardware on many modern CPUs [139]. In a previous study, AquaHash has shown promising results compared to XXHash and Murmur for small keys [140].

4.3 Learned Models as Hash Functions

Learned index structures [87] approximate the cumulative distribution function (CDF) of the data to predict the position of a lookup key in a sorted array. When the data has a learnable pattern, i.e., has low entropy, learned indexes can be much smaller than the input data itself. While initial proposals considered using neural networks to approximate the CDF, state-of-the-art learned indexes use a collection of simple *linear models*, which we refer to as submodels; these are fast to both learn and evaluate. Some indexes aim to minimize the root-mean-squared-error (i.e., L2 loss) [87] and others bound the maximum prediction error. Assuming a perfect modeling of the CDF, a learned index would constitute a perfect order-preserving hash function, i.e., a collision-free mapping from keys to positions. For the rest of this paper, we refer to Learned Model based Hash functions as *LMH*. Since real-world data contains many irregularities that make it hard to approximate, a learned index inevitably needs to trade off precision for space. With larger models, inference time increases because of limited cache sizes [105]. We describe the three main learned indexes we evaluate for hashing.

4.3.1 Recursive Model Indexes (RMI)

The recursive model index (RMI) is a multi-stage model combining simpler models [87]. When the data fits into memory, an RMI rarely has more than two stages. It is built in a "top-down" fashion. The stage-one model computes a rough approximation of the CDF, which is scaled between 0 and the branching factor B. This value is used to select a second-stage model, which approximates the local distribution of the data and is used to produce the final approximation. In other words, the stage-one model partitions the data into B buckets and each second-stage model approximates the data that falls into its corresponding bucket. A recent study [105] showed that RMI, amongst other indexes, achieves the best tradeoff between inference time and space.

4.3.2 Radix Spline Indexes (RadixSpline)

RadixSpline [81] is another learned index variant, that is built "bottom up", and consists of a linear spline [120] to approximate the CDF and a radix lookup table that indexes resulting spline points. Compared to RMI, RadixSpline can be built in a single pass with constant cost per element. RadixSpline's spline-building algorithm [120] bounds the maximum prediction error. Besides the maximum error, RadixSpline is parameterized with a certain number of radix bits that define the size of the radix table. Lookups first consult the radix table, which indexes r-bit prefixes of spline points and is used to narrow the search range over the spline points. Then binary search is used on the narrowed range to identify the two spline points surrounding the lookup key. Finally, linear interpolation between the two spline points is used to obtain a prediction. The necessity to search over the spline points make it somewhat slower than RMI which does not require any search in inner nodes.

4.3.3 Piece-wise Geometric Model Indexes (PGM)

Similar to RadixSpline, the Piece-wise Geometric Model Index (PGM) [57] provides an error-bounded approximation of the CDF. It consists of multiple levels where each level represents an error-bounded piece-wise linear regression (PLR). In contrast to a spline where consecutive spline points are connected, a PLR additionally stores an intercept value with each point. Like RadixSpline, PGM is built "bottom up" but instead of using a radix layer it recursively applies its PLR algorithm until a certain error threshold has been met. PGM can also be built in a single pass with constant amortized cost per element. Due to its multi-level structure, PGM can have slightly higher inference cost than RadixSpline [105] but is more robust when outliers are present.

4.4 Perfect Hashing

Where traditional hash functions aim to produce (near)-i.i.d. uniform random outputs, *perfect hash functions* provide an injective function that maps a set of elements into a

range. That is, for a given input set, the function will produce no collisions [59, 51, 12, 102]. Here, we focus on two types of perfect hash functions: *minimal perfect* (MPHF), and *order preserving minimal perfect* (OMPHF). We first explain the corresponding definitions, and then describe the state-of-the-art MPHF and OMPHF algorithms we study.

Perfect. A hash function $h(x) : X \mapsto [0, N]$ is *perfect* for the domain X if it is injective. Equivalently, it produces zero collisions in the output domain ($\forall x_1, x_2 \in X : h(x_1) = h(x_2) \implies x_1 = x_2$).

Minimal. A hash function $h(x) : X \mapsto [0, N]$ is *minimal perfect* if it is perfect and a bijection; that is, each element of the output range has a single corresponding domain element (Perfect, and additionally $\forall y \in [0, N] : \exists x \in X \mid h(x) = y$). The information theoretical lower bound for storing a minimal perfect hash function is $\lg e \approx 1.44$ bits per key [59, 51, 12, 73, 20] since key-related information is not retained after construction. For this reason, querying with non-keys (unknown keys) generally yields arbitrary results.

Order Preserving. Order preserving perfect hash functions order their outputs according to the original relative order $\preceq$ of input elements ($\forall x_1, x_2 \in X : x_1 \preceq x_2 \implies h(x_1) \leq h(x_2)$). Being able to store any arbitrary data order induces an $\Omega(n \log n)$ space cost [12].

Comparison to Traditional Hashing. In general, building a MPHF, $h(x) : X \mapsto [0, N]$, requires knowing the entire input set X a priori. In many implementations, the set X is not stored or reconstructible after the MPHF is built. Querying with a non-key $x' \notin X$ generally yields some arbitrary output value; most often, $h(x') \in [0, N]$, but this is not guaranteed. MPHF are generally not easily updated in place; often a full rebuild is performed if a new element is inserted, or other expensive (non-constant) time work. Compared to traditional hashing, where only constant work is necessary for initialization, MPHF and OMPHF generally require running a one time $\mathcal{O}(n)$ build algorithm before they can be used, which is sometimes even further relaxed to *expected* $\mathcal{O}(n)$ [51].

4.4.1 Recursive Splitting (RecSplit)

RecSplit [51] is a MPHF which has been shown to deliver state-of-the-art results in regards to space usage, lookup, and build time. Specifically, it comes close to achieving the theoretically optimal 1.44 bits per key in practice, while only requiring expected linear and constant times for construction and lookups, respectively [51].

RecSplit works by recursively partitioning inputs into ever smaller buckets until brute force search for a MPHF, i.e., a *bijection*, is viable. The threshold for this search, called *leaf size* l, as well as the average *bucket size* b for partitioning are parameters of the construction algorithm. RecSplit utilizes an indexed family of uniform random hash functions (examples in Section 4.2). This enables efficiently encoding the tree of brute-force determined indexes using an optimal Golomb-Rice instantaneous code [51, 141].

4.4.2 MWHC

MWHC [102], named after its four inventors Majewski, Wormald, Havas and Czech, was originally proposed as a family of OMPHFs with expected $\mathcal{O}(n)$ construction and $\mathcal{O}(1)$ access time. It has been extended to provide a practical MPHF with constant access and requiring 3 bits per key storage [19, 12]. We refer to our simplified implementation of the latter approach as *BitMWHC*. Abstractly, MWHC utilizes a hypergraph to efficiently find a solution for a randomly generated system of linear equations that is used to store the desired order preserving has function $f(x) : X \mapsto U$ given by

$$f(x) = v(h_1(x)) \diamond \ldots \diamond v(h_k(x)).$$

Each h_i denotes a distinct uniform random hash function, $v(x)$ maps each hash function output to a value in U and $\diamond$ reduces $U \times U$ to U. In practice, $v(x)$ may, for example, be implemented as a simple array of values, h_i as a family of reasonably high quality hash functions such as Murmur with seed values, and $\diamond$ as *xor* or as *addition* with an additional modulo computation at the end.

The construction algorithm first builds a k-hypergraph with each of the $\lambda|X|$

vertices corresponding to one entry of $v(x)$ and one edge $[h_1(x), ..., h_k(x)]$ for each input, where k and λ are user-defined parameters. All h_i are randomly chosen from a suitable family of hash functions as described above. A valid assignment for $v(x)$ exists, i.e., $f(x)$ is solvable iff this hypergraph is acyclic. A simple peeling scheme is used to both determine acyclicity and the order in which we can safely assign values to each $v(x)$ to yield the desired values for $f(x)$ for each $x \in X$. We simply restart if the acyclicity test fails, hence the *expected* $\mathcal{O}(n)$ construction time [102]. For $k = 3$, we require $\lambda \geq 1.23$ to efficiently find a suitable acyclic hypergraph [102, 117].

4.5 Hashing Schemes

When collisions occur in a hash table, they are resolved using *hashing schemes*. In this section, we give a brief background on the hashing schemes we study in this paper. We focus on (1) chained hashing (Section 4.5.1) and (2) two open-addressing schemes: linear probing and cuckoo hashing (Section 4.5.2). In each scheme, we discuss how the hash table is implemented and how collisions are handled.

4.5.1 Bucket Chaining (CHAIN)

Bucket chaining is a classic collision resolving scheme [10, 136, 140]. In this scheme, the hash table is implemented as an array of pre-allocated buckets, where each bucket stores multiple tuples, with collided keys, at a specific slot in the table. To insert a tuple, the key of this tuple is first hashed to a slot in the hash table, and then the whole tuple is first tried to be placed in the corresponding bucket at this slot. If the current bucket is already filled up, a new one is created, pre-allocated and chained to it. To query for a tuple, the query key is first hashed to a slot in the table (similar to what happens in inserts), then the chain of buckets at this slot is traversed until either the matching tuple is found or the end of the chain is reached (i.e., the matching tuple is not found). In general, bucketization improves the data locality, and reduces the number of cache misses. That being said, choosing the bucket size should be carefully tuned to avoid wasting large spaces.

4.5.2 Open-Addressing

In open-addressing, all tuples are inserted in the hash table slots themselves, without extra chains to handle collisions. In case of a tuple with a colliding key, the hash table slots are probed (i.e., searched), until a slot is found to place the tuple [32, 136]. Typically, a *probing scheme* decides the set of hash table slots to check, referred to as a probing sequence, till a place is found to insert the tuple. Query operations follow the same probe sequence. There are two main categories of probing schemes: (1) schemes that probe for the first available (i.e., empty) slot, and (2) schemes that evict the existing tuple at the probe location (i.e., when a collision occurs) and replace it with the new tuple. In this paper, we study an example of each of these two categories (linear probing and cuckoo hashing).

Linear Probing (LP)

This is the most basic probing scheme for collision handling in open-addressing. In this scheme, when inserting (or querying) a tuple, the key of this tuple is first hashed to obtain a hash table slot (i.e., initial probe location). Then, the hash table is sequentially traversed starting from this slot. In case of insertion, the traversal stops if an available slot is found. In case of querying, the traversal stops if we find either the matching tuple or an empty slot (i.e., matching tuple is not found). Linear probing has two main advantages: (1) its simple design, and (2) cache efficiency due to the sequential scan. In contrast, its performance degrades when large contiguous blocks of hash table slots are occupied, referred to as primary clusters. In this case, the number of nearby empty slots around each probe location is significantly reduced, and the scheme tends to have long probe sequences. Such performance issue can be avoided by either (1) increasing the hash table size such that the percentage of its occupied slots (a.k.a load factor) is always kept less than 60% [136] or (2) carefully tuning its update operations [14]. We note that there are two other popular variants of linear probing: (1) quadratic [83, 32], and (2) robinhood [24], which are efficient for write-heavy and high unsuccessful lookup workloads, respectively. However, according

to a recent study [136], linear probing outperforms both of them using the appropriate load factor. Therefore, we focus on linear probing here.

Cuckoo Hashing (CUCKOO)

Cuckoo hashing [124] provides another useful alternative hash table design. A simple variation of cuckoo hashing uses two subtables, where each subtable has an independent hash function. To insert a tuple, the key of this tuple is hashed with the first (or primary) hash function to obtain a slot in the primary table. If this primary slot is available, then the tuple is inserted and the probe sequence ends. Otherwise, the tuple tries to be inserted in the second (or secondary) subtable using the second hash function. If the secondary slot is occupied as well, then a kicking strategy is applied to evict the existing tuple in either the primary or the secondary slot, and replace it with the current input tuple. After that, the evicted tuple is reinserted again, following the same steps. The eviction chain continues until either all evicted tuples are successfully inserted or a maximum chain length is reached. This last case is a failure; one solution is for all tuples in both hash tables to be rehashed with two new hash functions.

With balanced kicking [124], the primary or the secondary slot is randomly selected for eviction. In biased kicking [39, 78], the tuple in the secondary slot is preferred for eviction, which has been shown to improve performance for positive lookups. We experimentally found that biased kicking performs better, so we use it throughout all our experiments involving cuckoo hashing.

To probe for a tuple, we need only to check the primary and secondary slots, which yields at most two cache misses regardless of the load factor. However, a major drawback of the simple variation of cuckoo hashing is the failure case, where the maximum length of the eviction chain is reached, happens at low loads. Higher loads can be handled by generalizing to use more hash tables (e.g., 4 instead of 2) [58, 136] or allowing multiple tuples per slot [40, 3, 140]. In this paper, we employ the bucketized variant, where each hash table slot allows more than one tuple, which again limits to two cache misses when a bucket fits in a cache line.

4.6 Collisions Analysis for Hashing

Here, we identify and analyze the factors affecting collisions for both *LMH* and traditional hash functions. This analysis helps us to identify situations where *LMH* can have fewer collisions than traditional hash functions. We specifically focus on *LMH* functions with piece-wise linear submodels for this analysis.

Notation. For ease of analysis, we start by focusing on the task of mapping N keys to N locations. This analysis readily generalizes, and the high-level conclusions are independent of this assumption, with the main difference being the number of locations increases, the number of collisions decreases. Assume that we apply a hash function f on the keys, where f could be a traditional hash (Section 4.2) or a *LMH* function (Section 4.3). Let $x_0, x_1, \ldots, x_{N-1}$ be the sorted array of the N input keys, and let $y_0, y_1, \ldots, y_{N-1}$ be the sorted array of the hashing outputs $f(x_0)$, $f(x_1), \ldots, f(x_{N-1})$ (note that $y_i = f(x_j)$ for some j, but y_i is not necessarily $f(x_i)$). For *LMH* functions, the y_i's may be on the real-valued range $[0, N)$, and we would then map each key to the location corresponding to the value of y_i rounded down to an integer. For convenience, we let $y_{-1} = 0$. The sorted output values generate a set of gaps $g_0, g_1, g_2, ...$ such that $y_i = \left(\sum_{t=0}^{i} g_t\right)$. We assume that g_i's are i.i.d, with probability density function $f_G(z)$ and CDF $F_G(z)$; this is a reasonable approximation for analysis. For example, for uniformly randomly distributed outputs $f(x_i)$, the gaps between y_i are approximately exponentially distributed [114].

Characterizing Collisions. A collision occurs when two keys are mapped to the same location. The key insight regarding collisions is that they depend on the gaps between consecutive sorted hashing output values $(y_i - y_{i-1})$. If the gap between two consecutive values is greater than one (i.e., $y_i - y_{i-1} \geq 1$), then the corresponding keys would definitely be placed in separate locations. On the other hand, if the gap is smaller than one (i.e., $y_i - y_{i-1} \leq 1$), the corresponding keys may be mapped to the same location; it depends where y_i and y_{i-1} relative to the integer boundary.

Ideally, we would want all the gaps to be more than one, to have zero collisions.

However, the gap values are constrained by the condition that the sum of all the gaps should be less than the number of locations which is N here.[1] Thus, the gap distribution would have to be the trivial distribution that is always 1 to avoid collisions.

Let c be the number of colliding keys (i.e., keys that are not alone in a location). Assuming that f is not a lattice distribution[2], we can describe the expected number of colliding keys $\mathbf{E}[c]$ with the following lemma. In the below, recall $\{x\} = x - \lfloor x \rfloor$. As N grows large, $\mathbf{E}[c]$ converges to

$$N\left(1 - \int_{u=0}^{1}\left(\int_{t=1-u}^{\infty}(1 - F_G(1 - \{t+u\}))\cdot f_G(t)dt\right)du\right).$$

We remark that the proof reveals that this formula is also a good approximation for large N.

Let Z_i be the indicator random variable that is 1 if y_i is alone in its own location. We first consider the position of y_{i-1}. For sufficiently large i, $\{y_{i-1}\}$, the fractional part of y_{i-1}, is known to converge to the uniform distribution on $[0,1]$ (see, e.g., Thm 5.8.4. of [82]). We therefore treat $\{y_{i-1}\}$ as being distributed uniformly on $[0,1]$. Accordingly, the probability y_i is in a different location from y_{i-1} is given by

$$\int_{u=0}^{1}\left(\int_{t=1-u}^{\infty} f_G(t)dt\right)du.$$

We also need, however, that y_{i+1} is also in a different location from y_i. This depends on the value of $\{y_i\}$. Taking this into consideration yields the following probability for Z_i:

$$Pr(Z_i = 1) = \int_{u=0}^{1}\left(\int_{t=1-u}^{\infty}(1 - F(1 - \{t+u\}))\cdot f_G(t)dt\right)du.$$

As N grows large, the approximation of uniformly distributed $\{y_{i-1}\}$ is arbitrarily accurate (as accurate as desired) for almost all i, giving the convergence.

Collisions for Traditional Hash Functions. In case of a truly random hash

[1] Sum of gaps is: $\sum_{t=1}^{N-1}(y_t - y_{t-1}) = y_{N-1} - y_0 \leq N$.

[2] Lattice Distribution: A discrete probability distribution concentrated on a set of points of the form a+nh, where h>0, a is a real number and n=0,±1,±2,.

function, the output values will be uniformly distributed in the range $[0, N]$ irrespective of the input distribution. Therefore, the gap distribution of the output values is very well approximated by the exponential distribution with mean 1. Murmur, XXHash and most other traditional hash function displayed this behaviour in our evaluation.

Collisions for *LMH* Functions with Piece-wise Linear Submodels. To gain intuition, let us start by using a *single* linear model to approximate the CDF of the input data $x_0, x_1, \ldots$, and this will give us our hash function f. Let the linear model be $m * (x - x_0)$ where m is $(N - 1)/(x_{N-1} - x_0)$. Note that the slope would be approximately the mean of the gap distribution of the input keys. The resulting hash function would be $h(x) = (m * (x - x_0))$ which maps the input keys in the range $[0, N)$. After applying this hash function to obtain the output values $y_0, y_1, \ldots$, we notice that the gaps between the output values are simply the scaled version of the gaps between the input keys: $y_{i+1} - y_i = (x_{i+1} - x_i) * m$. At a high level, if the input is evenly spaced, then our outputs will similarly be evenly spaced, resulting in fewer collisions. If the input gaps are high in variance, we would expect more collisions. In *LMH* functions, this scaling would happen at the submodels scale.

Accordingly, if the data is generated similarly to our theoretical model, with a gap distribution $g^{'}$ $(x_0, x_1 = x_0 + g^{'}_0, x_2 = x_1 + g^{'}_1, ...)$, the gap distribution of the input keys determines the gap distribution of the output keys and thus the amount of collisions. In certain cases, like auto-generated keys (1,2,3,4,5,...) perhaps with some deletions or noise, the input gaps are mostly constant. In this scenario, a piece-wise linear model can lead to fewer collisions than a traditional hash function. However, if the input keys are generated by sampling from a distribution instead of sequentially, multiplying the CDF value of the key by the array size will behave as an order-preserving hash function. A *LMH* function that approximates this underlying distribution would behave essentially the same as a truly random hash function in terms of collisions.

Increasing the number of submodels can improve the accuracy of when using a piece-wise linear model to approximate a CDF. This helps in the case of indexing an item, but from our argument above, we see that this does not necessarily reduce the number of collisions. We show this via an example. We mapped 100 million

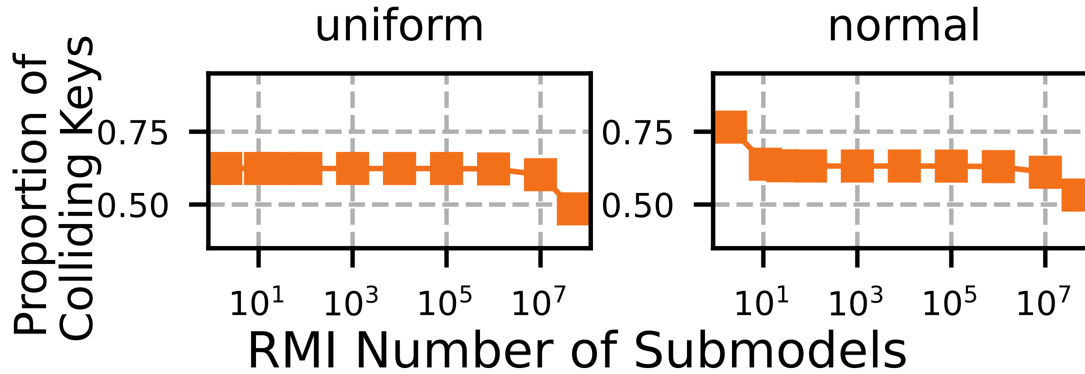


Figure 4-1: Proportion of collisions with increasing RMI size for uniform randomly and normally distributed keys.

uniform randomly and normally distributed keys to 100 million slots using RMI with varying number of submodels. In Figure 4-1, we plot the proportion of collisions as we increase the number of submodels in RMI. We observe that for uniform randomly distributed keys increasing the number of linear submodels does not affect collision metric until we reach 50 million submodels. RMIs with 100 submodels and 100000 submodels are both able to approximate the CDF of the distribution well and the output is approximately uniformly randomly distributed in both cases. The larger RMI provides better accuracy than the smaller one but essentially the same number of collisions. The RMI with 50 million submodels essentially memorizes the empirical CDF of the dataset and thereby results in lower collisions. For the normal distribution, an initial increase in the number of submodels reduces collisions as an RMI with only 1-2 submodels fails to approximate the CDF of normal distribution well.

Conclusions. In summary, our discussion and analysis in this section supports the following points:

- Collisions are dependent on the gaps g_i between consecutive sorted hashing output values.
- For *LMH* functions with piece-wise linear submodels, the number of collisions depends on the key distribution, specifically the gaps between consecutive sorted keys $(x_i - x_{i-1})$.
- Having more linear submodels in the *LMH* function improves the model accuracy

but may not reduce collisions.

4.7 Evaluation

In this section, we present an empirical study for the performance of *LMH* functions and compare them against both traditional and perfect hashing. Our main objective is to answer the following question: *what are the main workload characteristics, scenarios, and operations where employing LMH functions would improve performance?* We first study the collisions and computation time tradeoffs (Section 4.7.2). Then, we evaluate the performance of the various types of hash functions in supporting the main hash table operations, lookup and insertion, for different types of hash tables (Section 4.7.3). We also provide some more detailed experiments regarding issues such as how collisions affect performance in practice, and the impact of construction time for *LMH* (Section 4.7.4). Finally, we move to some higher-level operations that use hash tables, and show cases where *LMH* can improve the performance of range queries (Section 4.7.5) and non-partitioned hash join (Section 4.7.6).

4.7.1 Experimental Setup

Datasets. We use both real and synthetic key datasets in our experiments. All keys are 64-bit integers. For real keys, we use two datasets from the SOSD benchmark [105]. These datasets are (1) *fb*, which has 200 million randomly sampled Facebook user IDs, and (2) *wiki*, which has 200 million timestamps of edits from Wikipedia. In any experiment, we use either the whole dataset or a sample from it (details are mentioned in each experiment separately).

For synthetic keys, we use three different key generation processes: (1) *gap_10*, in which sequential keys are first generated at regular intervals of 10 and then 10% of the keys are uniformly randomly deleted (this represents the case of auto-generated IDs after removal of certain users), (2) *uniform*, in which keys are generated uniformly at random in the range $[0, 2^{50}]$, and (3) *normal*, in which keys are generated from a

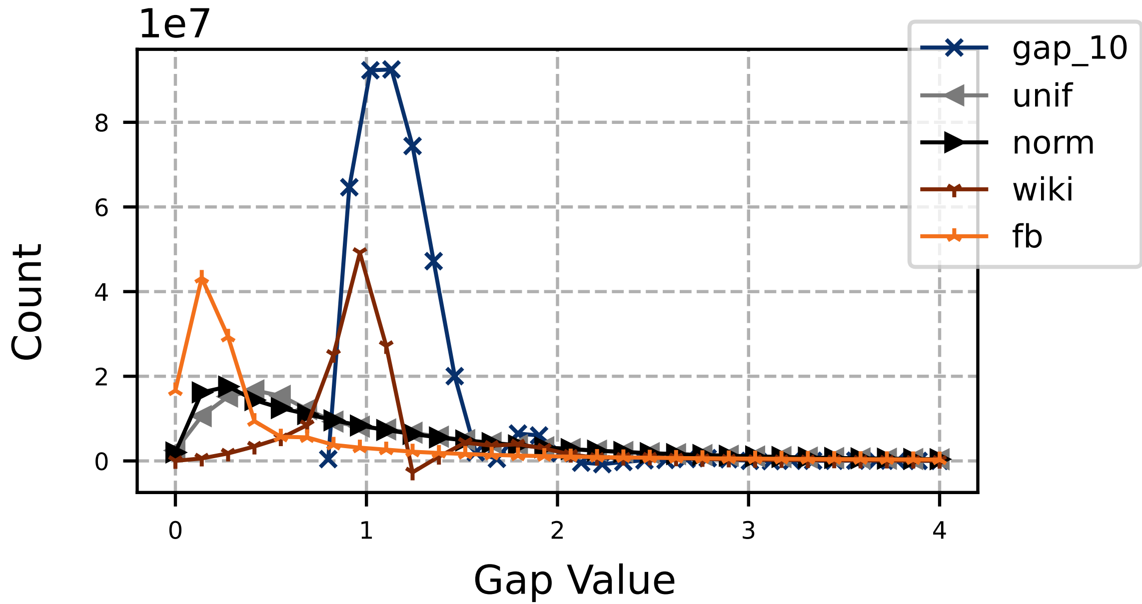


Figure 4-2: Gap distribution of various datasets

normal distribution ($N(\mu = 100, \sigma = 20)$) and then are linearly scaled to the range $[0, 2^{50}]$.

As discussed in Section 4.6, the gaps between sorted hash outputs determine collisions. In order to understand the distribution of these gaps in our datasets, we use an RMI, with 1 million submodels, to map 100 million keys from each dataset to 100 million slots and then plot the gaps between consecutive sorted output values. In Figure 4-2, x-axis shows the gap value and y-axis shows the count of this gap. We observe that *gap_10* and *wiki* datasets have gaps concentrated around 1. *uniform* and *normal* datasets have very similar gap distributions concentrated around 0.25-0.35, while *fb* dataset has significant number of gaps concentrated around 0.1.

In all hash table, range query, and join experiments, we generate 8-byte payloads chosen randomly from the range $[0, 2^{64}]$. All tuples (or keys) are randomly shuffled before running any experiment.

Hardware. All experiments are conducted in the main memory on a machine with 256 GB of RAM and an Intel(R) Xeon(R) Gold 6230 CPU @ 2.10GHz with Skylake micro-architecture (SKX) and L3 cache of 55MiB. The operating system is Arch Linux with a page size of 4KB (default page size). The implementation of all hashing

	wiki	*fb*	*gap_10*	*uniform*	*normal*
RMI	10^3	10^7	10	10^2	10^2
RadixSpline	10^3	10^8	10	10^2	10^2

Table 4.1: Default numbers of submodels in *LMH* functions.

functions and schemes is our own and in C++. The binaries are compiled with clang++ (12.0.1) using optimization -O3. We have activated prefetching.

Default Settings. Unless otherwise mentioned, we set the number of submodels in RMI and RadixSpline as stated in Table 4.1. Each value represents the least number of submodels needed to give the least amount of collisions in a specific dataset. For PGM models, we vary the error bound values from 1 to 10000. The number of tuples (or keys) in each synthetic dataset is set to 100 million. We use a default bucket size of 1 in the bucket chaining scheme. To support cuckoo hashing with a load factor up to 90%, we use a bucket size of 4 as described in [3]. As mentioned in Section 4.5.2, we use the biased kicking strategy as it performs better than the balanced one. We set 50000 as a maximum number of kicks. This value led to a suitably small number of insert failures.

Metrics. Throughput is the default metric in most of the experiments. When studying the hash function itself, we use the *computation throughput*, which is the number of hash function operations executed per second. In the hash table and range query experiments, we use the number of completed queries (e.g., probe/insert queries on hash tables) per second (i.e., *queries throughput*). For the join experiments, we use the runtime instead of the throughput to perform a breakdown for the join phases.

Measurement and Profiling. For all experiments, we report the average of three independent runs, where we use a different random seed for generating and shuffling synthetic and real data, respectively, in each run. We use the PerfEvent library [99] to profile the low-level hardware counters in Section 4.7.3. These counters include L1 and LLC cache misses, branch misses and cycles.

Beyond Scope. Our study focuses only on the single-threaded setup to fairly compare the performance of *LMH* functions with traditional and perfect hashing, without parallelism optimizations. That being said, we believe that multi-threaded

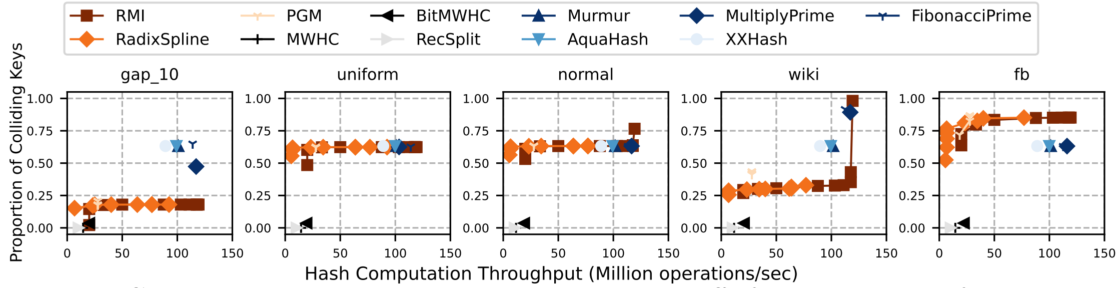


Figure 4-3: Computation throughput and collisions tradeoffs for various hash functions and using different datasets.

implementations of these hashing schemes should be evaluated in a standalone study, which we currently plan as an extension for this work.

4.7.2 Computation Throughput vs Collisions

In this experiment, we are interested in studying the tradeoff between the hash function quality and its efficiency. We use the eleven hash functions and five datasets previously discussed. In each dataset, we map a randomly-selected 100 million keys into 100 million hash table slots, and measure both the hash function computation throughput, and the proportion of colliding keys.

Figure 4-3 shows the results of this experiment. Note that each traditional and perfect hash function is represented as a single point in the scatter plot. However, in *LMH* functions, we vary (1) the number of submodels in RMI and RadixSpline from 1 to 50 million and (2) the error bound of PGM as in the default settings, yielding multiple points on the plot. As expected, traditional hash functions have a significant number of collisions, and perfect hash functions are slow. All traditional functions have similar throughput (90-100 million operations/sec) and colliding keys proportion (0.63-0.65) across all datasets. This proportion of colliding keys nearly matches that for truly random hash function which is approximately ($1 - 1/e \approx 0.632$). All perfect hash functions have no collisions (by definition), but low throughput (10-20 million operations/sec) due to the high computation overhead coming from either an expensive traversal over the splitting tree in RecSplit [51] or multiple random accesses to the array storing the hypergraph-related values in MWHC [102].

The performance of *LMH* functions, however, depends on the gap distribution of the input datasets as discussed in Section 4.6. The RMI and RadixSpline hash functions, at their best configurations, can achieve low collisions (0.2 and 0.3) and high throughput (80 to 120 million operations/sec) in two datasets, *gap_10* and *wiki*. For these datasets, the gaps are more or less evenly spaced, and hence *LMH* functions yield a very low number of collisions. In addition, the number of submodels needed for these datasets is small, which makes the *LMH* computation overhead efficient. For *fb*, the variance in the gap distribution is very high, yielding a large number of collisions. Reducing these collisions requires using a large number of submodels (the best proportion of colliding keys is 0.5), yielding low throughput.

In the case of *uniform* and *normal* datasets, we observe that *LMH* and traditional functions have similar collision behavior, regardless of the used number of submodels. This matches our understanding that the CDF-based hashing of *LMH* for these datasets will lead to a distribution of items in buckets that is nearly the same as traditional hashing (as described in Section 4.6). In general, as discussed in Section 4.6, increasing the number of submodels in *LMH* functions does not necessarily decrease the collisions. For example, in *wiki*, the proportion of colliding keys using RMI significantly drops from 0.9 to 0.3 after an initial increase in the number of submodels from 1 to 1000, and then becomes stable regardless the number of submodels used.

For the rest of our experiments, we choose the best hash functions, in each hashing category, in terms of both computation time and collisions: Murmur and MultiplyPrime for traditional hashing, RMI for *LMH* and MWHC for perfect hashing.

4.7.3 Hash Table Performance

Here, we are interested in studying the performance of two main hash table operations; probe and insert.

Probe Throughput. In this experiment, we first insert 100 million tuples in a hash table with varying number of slots (i.e., buckets). Then, we probe the hash table with all the inserted tuples (i.e., query workload), after randomly shuffling them, and measure the throughput. We generate different load factors by varying the number of

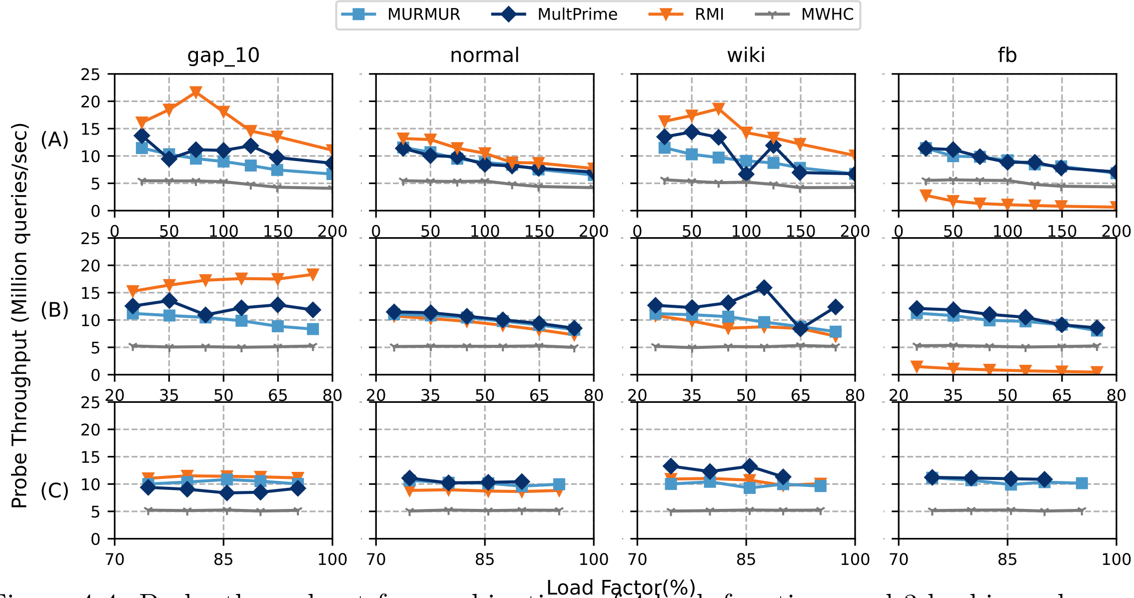


Figure 4-4: Probe throughput for combinations of 4 hash functions and 3 hashing schemes: (A) bucket chaining, (B) linear probing, and (C) cuckoo hashing. Results are shown for 4 different datasets, and various load factors for each hashing scheme.

slots. Figure 4-4 shows the results for this experiment while using four input datasets (*uniform* and *normal* nearly have the same results). For each hashing scheme, we use a different range of load factors that are suitable for the scheme. For example, we use load factors $\geq 100\%$ in bucket chaining as it can support inserting tuples more than the total slots in a hash table. Also, we only use high load factors ($\geq 75\%$) with cuckoo hashing because, in smaller load factors, cuckoo hashing is always dominated by other schemes [136].

For bucket chaining, we observe a clear ranking among the different hash functions. RMI has the best throughput in all datasets, except *fb*, with average 1.4x better throughput than the second best function, MultiplyPrime. In addition, MWHC has the worst throughput in all datasets, except *fb* in which RMI becomes the worst option with average throughput of 1.5 million queries/sec only. This is because (1) RMI has the fewest collisions in *gap_10*, *normal*, and *wiki*, and (2) MWHC has the highest hash computation overhead in all datasets. Fewer collisions results in shorter chains that need to be traversed during the probe queries, and hence fewer cache misses. For *fb*, although MWHC still has the highest computation overhead, the high amount of collisions for RMI leads it to perform the worst.

We also look at the throughput across different load factors. Increasing the load factor increases collisions because there are fewer slots, which degrades the throughput. For example, Murmur has throughputs of 12.5 and 6 million queries/sec at load factors of 25% and 200%, respectively. However, we observe two exceptions to this throughput trend when using: (1) RMI at load factors between 25% and 100% in *gap_10* and *wiki*, where the throughput actually increases, and (2) MWHC in all load factors, where the throughput is fixed around 6 million queries/sec, regardless of the dataset. The reason for the first exception is that collisions are already close to zero in these two datasets, so increasing the load factor from 25% to 100% primarily reduces empty hash table slots, leading to better caching behavior. The reason for the second exception is that the MWHC computation overhead for each tuple is constant [102], regardless of the used load factor.

For linear probing, the throughput depends on the length of the sequential scan needed to handle collisions. We observe that the throughputs achieved by using Murmur, MultiplyPrime, and MWHC have the same trend as in bucket chaining. In contrast, RMI has the following three notable changes. First, RMI yields a bit worse throughput than traditional hashing in *normal* (10% less throughput). Although the number of collisions using RMI is slightly smaller in this case (Section 4.7.2), the effect of this difference can be hidden by the sequential scan benefits (e.g., prefetching) of linear probing, and hence the overhead of RMI hash computation becomes more significant. Second, RMI results in worse throughput than traditional hashing in *wiki* (average 40% less throughput than MultiplyPrime). This was a bit surprising as *LMH* functions result in significantly fewer collisions than traditional hashing. However, we found that in a few parts of the *wiki* dataset RMI maps up to 100 keys to the same slot, creating clusters that result in long sequential scans during probing.

For cuckoo hashing, we observe that the throughputs achieved by any hash function are pretty much similar within the same dataset, regardless of the load factor used. This is expected as handling collisions in cuckoo hashing is typically performed in constant time (two cache misses at most). Even better, we employ a biased kicking strategy, in which most of the tuples are placed in their primary hash slots (i.e., one

cache miss for most of the probes). This makes the hash function computation (model prediction in case of RMI) has a great impact on the probe latency in cuckoo hashing, and explains why the throughput using RMI is a bit worse than using traditional hashing in *normal*, and almost similar in *gap_10* and *wiki*. Note that we failed to construct the cuckoo hash table for *fb* using RMI because the resulting number of collisions is extremely high, and the required number of kicks to handle them exceeds the maximum threshold. Also, the construction failed using MultiplyPrime at load factor 95% because of high number of collisions. In general, cuckoo hashing significantly reduces the impact of collisions, regardless of the hash function used, and hence the performance improvement of *LMH* over traditional hashing becomes negligible.

In conclusion, the potential performance improvement of *LMH* functions (e.g., RMI) over traditional and perfect hash functions is greatly affected by the used hashing scheme. It is strongest with bucket chaining, and weakest with cuckoo hashing.

Insert Throughput. Here, we use the same setup in the probe throughput experiment, while changing the query workload. To generate the insert workload, we first uniformly and randomly sample 101 million tuples from an input dataset. Then, we initialize the hash table by bulk-inserting 100 million tuples from this sample as in the probe throughput experiment, and use the remaining 1 million tuples as the query workload. Figure 4-5 shows the results of this experiment for two input datasets only, *wiki* and *fb* (the remaining datasets show similar performance trends).

In general, the relative ranking and throughput trends remain the same as in the probe throughput. We also observe that, in *wiki*, the performance benefit that RMI offers over MultiplyPrime - when used with bucket chaining - in insertion is not as high as in probing (only 10% throughput improvement in insertion compared to 30% in probing). Probing time mainly depends on the length of the chain to be traversed whereas insertion requires allocating and adding new buckets to the chain, and hence the collision reduction improves only a portion of the total insert time. Another interesting observation is that at load factor 95% using cuckoo hashing with MWHC is the best because the overhead of kicking operations becomes higher than

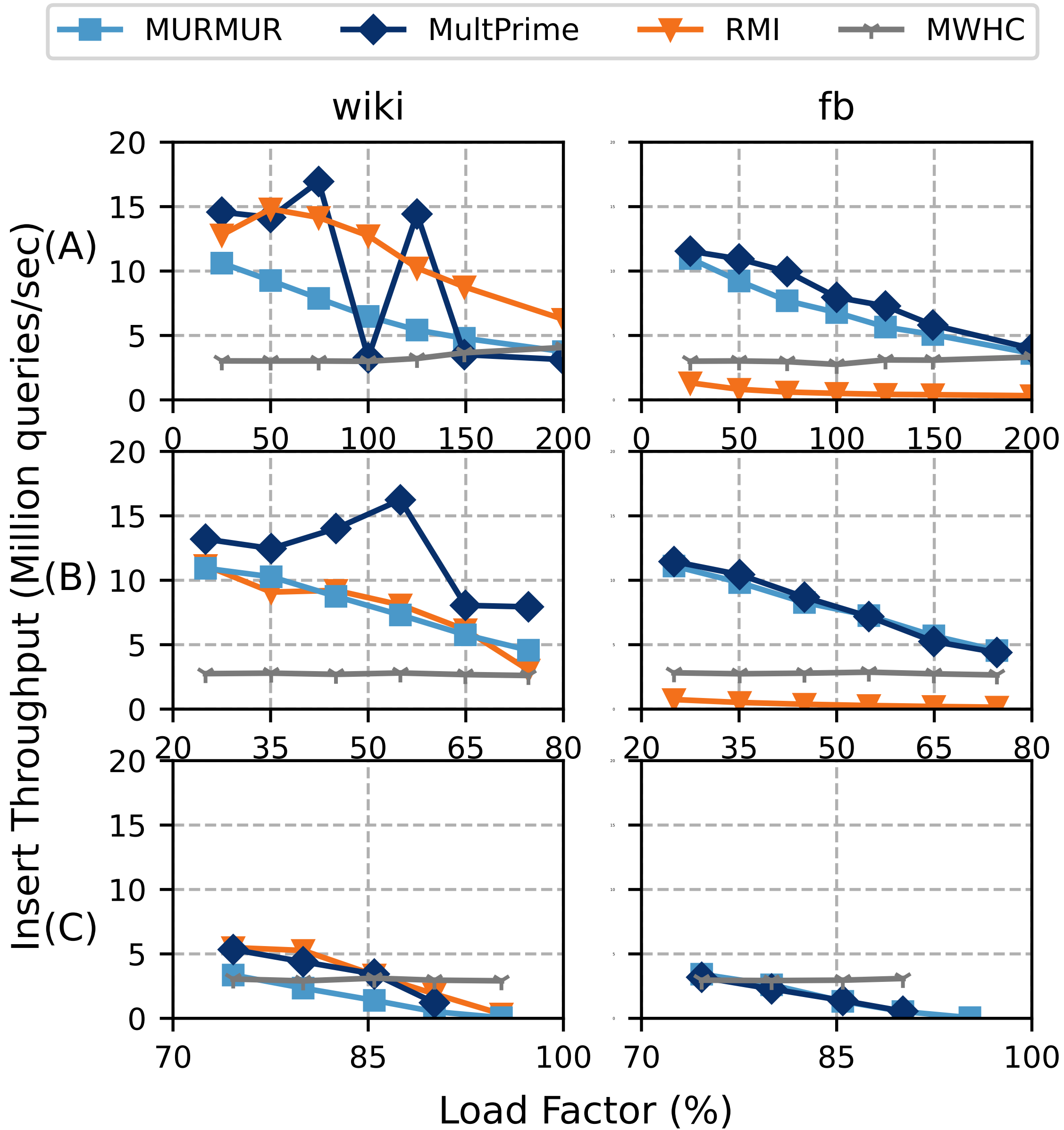


Figure 4-5: Insert throughput for combinations of 4 hash functions and 3 hashing schemes: (A) bucket chaining, (B) linear probing, and (C) cuckoo hashing. Results are shown for 2 different datasets, and various load factors for each scheme.

the complex computation of MWHC.

Performance Counters. To deeply understand what happens on the hardware level, we investigate the following four performance counters: cycles, L1 cache misses, last-level cache (LLC) misses, and branch misses. Figure 4-6 shows the average values of these counters per tuple for the probe throughput experiment in Figure 4-4 at load factor 80% and only for two datasets *gap_10* (first row) and *fb* (second row). For

Figure 4-6: Performance counters per tuple for the probe experiment in Figure 4-4 using the *gap_10* (first row) and *fb* (second row) datasets at load factor 80%.

MWHC, we only show chained results as other schemes have similar performance.

For *gap_10*, the three RMI-based variants achieve the lowest performance counter values (e.g., one L1/LLC miss per tuple for RMI-CHAIN and RMI-LP) compared to other variants. For *fb*, we found that scanning very large clusters, as in RMI-LP or MULT-LP, significantly increases both cache and branch misses, and in turn increases cycles (high cache and branch misses lead to an excessive increase in the amount of CPU stalls and wasted cycles, respectively). In contrast, RMI-CHAIN significantly reduces the effect of the high collisions produced by RMI in *fb* (RMI-CHAIN has at least 3X less LLC misses and cycles than RMI-LP). Even in *gap_10*, RMI-CHAIN still has at least 2X and 4X less cycles than RMI-LP and RMI-CUCKOO, respectively. This confirms our conclusion about the impact of hashing schemes on the probe throughput using *LMH* functions. Another interesting observation in *fb* is that MULT-CHAIN and MULT-CUCKOO have close values in all counters, yet MULT-CUCKOO is a bit better in cycles and branch misses. This shows that bucket chaining can provide a competitive performance at challenging datasets and high load factors.

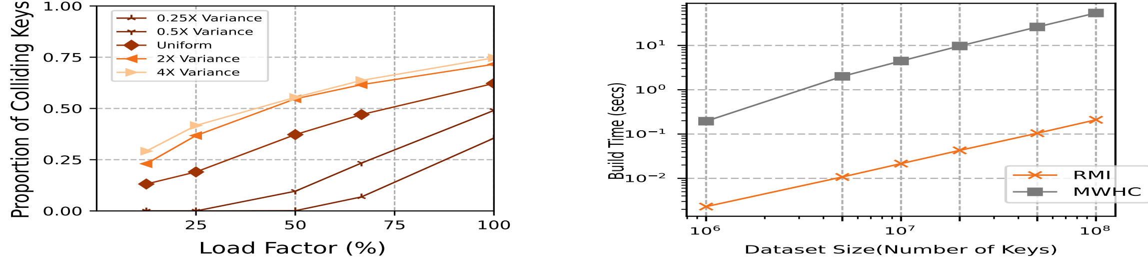


Figure 4-7: Effect of (1) gap distribution on *LMH* collisions (left), and (2) dataset size on building time (right).

4.7.4 More Performance Analysis

In this section, we study more parameters related to *LMH* functions and their performance in hash tables.

Gap Distribution. In this section, we vary the gap distribution to display that gaps concentrated around the mean have lower collisions. Assuming that the variance of the gap distribution of *uniform* keys is X, we generate 4 different variations of the *uniform* dataset, such that the gap distribution variances of their keys are $2X$, $4X$, $0.5X$ and $0.25X$ (i.e., scaled variances) [3]. Then, we insert the keys of each dataset variation in a hash table using RMI, and calculate the proportion of colliding keys. The left part of Figure 4-7 shows the proportion of colliding keys with varying load factors. As expected, the amount of collisions can be decreased by decreasing either decreasing the gap variance or the load factor. Lower gap variance cause the gap distribution to concentrate around the mean value resulting in lower collisions.

Build Time. Unlike traditional hash functions, *LMH* and perfect hash functions require a building stage. In right part of Figure 4-7, we show the building time for RMI and MWHC, as examples for *LMH* and perfect hashing, respectively, while using the *uniform* dataset and vary the number of keys between 10^6 and 10^8. We can see that the building time of MWHC is consistently two orders of magnitude slower than the building time of RMI. Although MWHC has an *expected* $\mathcal{O}(n)$ construction time [102], its hypergraph building process requires an excessive amount of random memory accesses, and hence cache misses (check Section 4.4). In contrast, building an

[3] Each dataset variation is generated by scaling the gaps between the *uniform* keys with the corresponding factor (e.g., the "$2X$ Variance" dataset scales the gaps between *uniform* keys by a factor of 2).

RMI requires only sorting the data once and doing multiple sequential passes over it, which is a cache-friendly process.

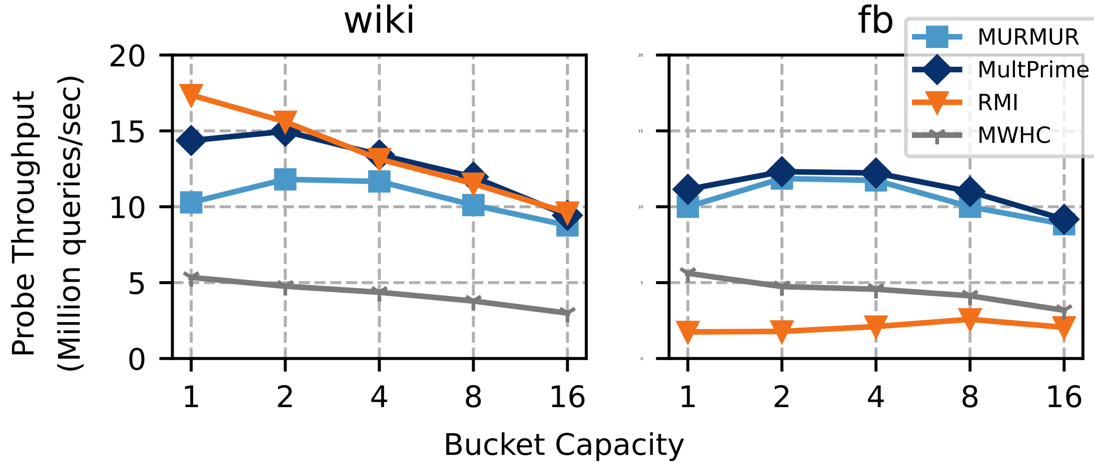


Figure 4-8: Effect of increasing the bucket capacity on the probe throughput of a chained hash table at load factor of 50%.

Bucket Capacity. In this experiment, we study how increasing the bucket capacity (i.e., number of tuples in the bucket) affects the probe throughput. For each dataset, we build different hash tables with a load factor of 50%, and are bulk-loaded with 100 million tuples. Note, since we fix the load factor, increasing the bucket capacity by a factor X reduces the number of buckets by a factor $\frac{1}{X}$. We use the same inserted tuples as a probe workload, after randomly shuffling them, and measure the throughput as in Figure 4-8.

In bucket chaining, increasing the bucket capacity reduces the length of needed chains (i.e., extra buckets) to handle collisions, as any colliding key now has a high probability to be in the main hash table bucket. However, this increases the probe time as well because finding a key in the bucket requires larger scan overhead as the bucket becomes larger. In *wiki*, RMI already produces a low number of collisions, and hence increasing the bucket capacity will not benefit chaining, yet causes probes to scan unnecessary keys, and hence the throughput significantly decreases (this is also true for MWHC as it has no collisions by definition). In *fb*, RMI produces a lot of collisions that result in longer chains. In this case, increasing the bucket capacity improves the probe throughput. In the case of Murmur and MultiplyPrime, we can

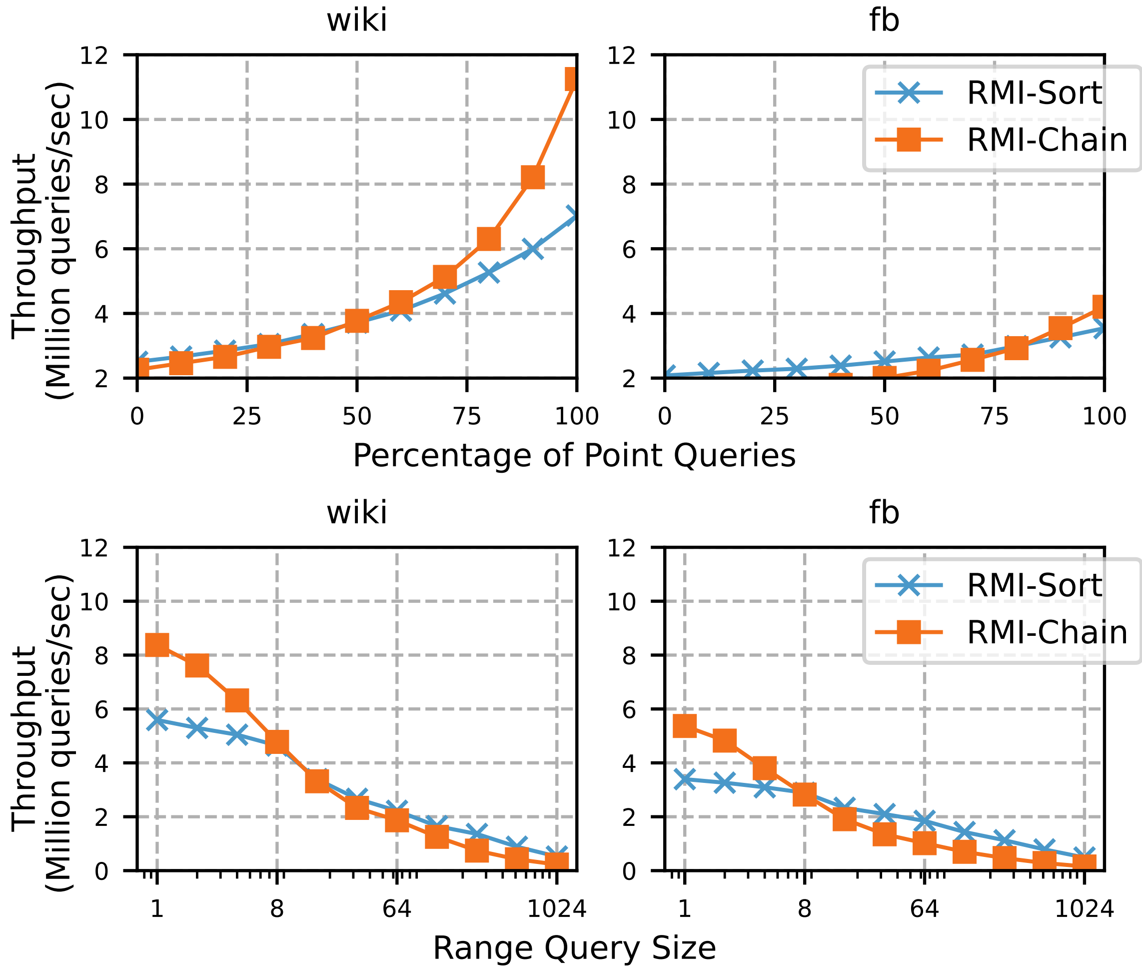


Figure 4-9: Effect of both point queries percentage (first row), and range query size (second row) on the point/range queries throughput.

see that benefit from the capacity increase initially, then they suffer from the extra scan overhead within the bucket.

4.7.5 Range Queries Performance

Hash tables support fast point queries only, and do not support range queries. On the other hand, index structures like B-Tree, ART [94], and RMI [87] support both point and range queries. However, the performance of index structures in point queries is not as efficient as hash tables. In case of having a mixed workload of point and range queries, where range queries represent only a small proportion, one cannot use a hash table and is forced to use an index to be able to answer the range queries. This results

in a huge performance degradation for the majority of the point queries. Fortunately, we can use *LMH* functions along with bucket chaining to build a hash table that supports range queries in addition to its natural support for fast point queries. In this case, a range query can be processed by scanning the buckets between the locations corresponding to the query lower and upper bound keys. This is possible as *LMH* functions are order-preserving, and hence the target keys are bound to be within the bucket locations.

Point Queries Percentage. In this experiment, we study the throughput of a mixed workload of point and range queries (the percentage of point queries is variable) using (1) an RMI-CHAIN hash table (bucket size of 8 and load factor of 50%), which is our proposed solution, and (2) a sorted array of the input data with a typical RMI on top of it (we refer to it here as RMI-SORT). We use *wiki* and *fb*, where we sample 100 million tuples from each one of them as input data. We generate a mixed query workload by first randomly sampling $X\%$ of the input data to be used as point queries, and then for the rest of the workload (i.e., 100-$X\%$) we generate random range queries that retrieve about 25-50 tuples. The upper part of Figure 4-9 shows the results for this experiment, where we vary X between 0 and 100. As expected, in both *wiki* and *fb*, RMI-CHAIN has faster throughput than RMI-SORT when the workload has a majority of point queries, and vice versa. This is explainable as, for a point query, RMI-CHAIN just needs to scan the bucket pointed out by the model whereas, RMI-SORT needs a local search to find the relevant key. For a range query, RMI-CHAIN scans the buckets that fall within the range query and also the additional chains associated with them. This leads to excessive random memory accesses, and hence a decrease in the throughput. In contrast, RMI-SORT is more suitable for range queries as it only needs to sequentially scan the relevant keys in the sorted range.

Range Query Size. Here, we reuse the setup of the previous experiment, while focusing only on 100% range queries workload. We vary the range query size from 1 to 1024. The below part of Figure 4-9 shows the results for this experiment (x-axis has a logscale). With increasing the range size, RMI-CHAIN becomes slower than RMI-SORT as RMI-CHAIN needs to scan additional chained buckets.

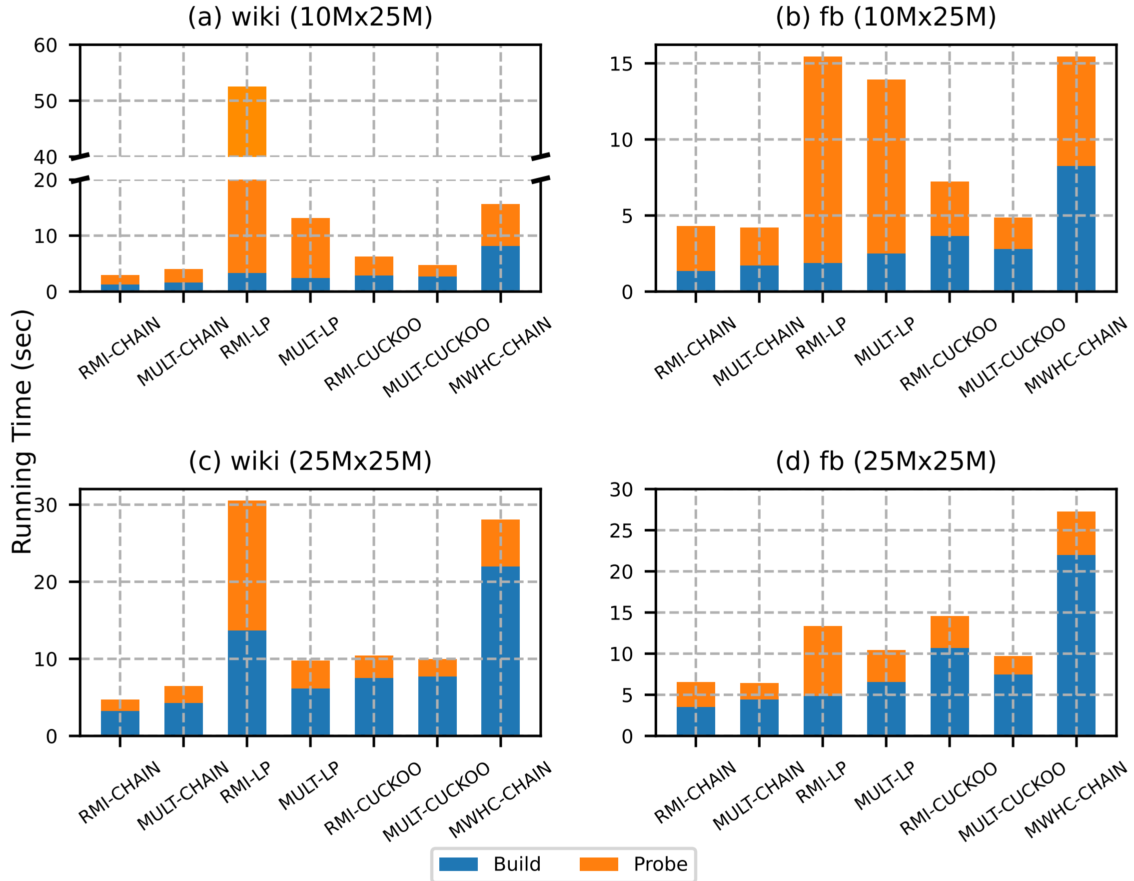


Figure 4-10: Runtime breakdown for the different implementations of non-partitioned hash join (NPJ) using various combinations of hashing functions and schemes.

4.7.6 Hash-based Join Performance

In this experiment, we are interested in understanding the performance of non-partitioned hash join (NPJ) over two input relations [92, 153], when implementing it using different combinations of hashing functions and schemes. Note that we do not investigate the hashing effect on partitioned hash join [153, 142] as it employs small cache-fit hash tables. In this situation, using any traditional hash function to build such small tables will be the best choice. In contrast, NPJ builds a large global hash table for the smaller input relation and the probability of having performance degradation, due to large number of collisions, is high. Therefore, employing an efficient hash function is crucial to improve the join performance. We use *wiki* and *fb* datasets, where we uniformly and randomly sample two variations from each dataset

with 10M and 25M tuples. These variations will be used to perform the NPJ on. Figure 4-10 shows the running time of the NPJ build and probe phases.

Interestingly, we can observe that RMI-CHAIN and MULT-CHAIN have the best join performance in both *wiki* and *fb*, where RMI-CHAIN has 28% less runtime than MULT-CHAIN. Looking at the build phase, we can see that RMI-CHAIN and RMI-LP build the hash table more efficiently than other solutions in most of the cases. This is mainly because RMI sorts the data to build its submodels, and then uses them to insert each tuple from the sorted data into the hash table. Although sorting the data is a bit expensive, it helps the model-based insertion to happen in a cache-friendly manner, and the overall overhead, including both sorting and model-based insertion, is still significantly less than randomly inserting tuples using MultiplyPrime and Murmur. This observation was confirmed in a previous study before [89]. Due to the efficiency of RMI in building the hash table, the total time of NPJ using RMI-CHAIN becomes more competitive with MULT-CHAIN in a challenging dataset like *fb* because the performance gain in building compensates for the performance degradation in probing, and the total running time becomes very close.

4.8 Related Work

Traditional Hashing. Traditional hash functions can be categorized as either non-cryptographic [52] or cryptographic [1]. *Non-cryptographic* hash functions [32, 83, 136, 3, 140, 29, 139, 37], which we mainly focus on in our study, are mostly used in building data structures (e.g., hash maps, Bloom filters) and algorithms (e.g., searching) due to their good balance between computation time and collision rates. More recent work has focused on optimizing the performance of non-cryptographic hashing on modern hardware by either proposing new hash functions [154] (e.g., CLHash [95], and tabulation hashing [131]) or customizing the existing ones to utilize the underlying hardware (e.g., GPU [96] and SIMD vectorization [11, 71]). *Cryptographic* hash functions have the property of being computationally hard to invert. These functions can still be used in building data structures, yet their performance can be much slower

than non-cryptographic ones [29]. Examples of cryptographic hash functions include MD5 [138], SHA1 [50] and SipHash [8].

Perfect Hashing. Perfect hashing has been widely studied; see, e.g., the survey in [102]. In general, obtaining a perfect hashing scheme, especially for large datasets, is known to be difficult [93, 108]. Although perfect hashing schemes provide competitive probe performance, due to the lack of collisions, they incur significant overhead to support incremental updates. Perfect hashing solutions can be divided into two categories: *static* and *dynamic*. When inserting new tuples to the hash table, the static solutions (e.g., [51, 19, 102, 125]) reconstruct the whole table from scratch, while the dynamic solutions (e.g., [160, 38]) reconstruct the table parts that are related to the update only. Another interesting line of work is improving the perfect hashing computation using modern hardware, such as GPU (refer to a survey in [96]).

Learned Models for Indexing and Hashing. During the last few years, the idea of using CDF-based learned models to replace traditional indexes has been investigated extensively including single-dimension (e.g., [87, 81, 57]), multi-dimensional (e.g., [118, 43]), updatable (e.g., [47]), and spatial (e.g., [97, 133, 127]) indexes. Interestingly, the authors of [87] also discussed the idea of using learned models as order-preserving hash functions. A recent study [140] initially investigated whether learned models are better than traditional hash functions in performing hash table lookups or not. In contrast, our proposed study is more comprehensive as it spans additional hash function types, hashing schemes, workload types, and hash-based operations. Another recent interesting work [74] employs an entropy-learned approach to reduce the hashing overhead by choosing how much and which parts of the input data we need to hash, instead of hashing the whole input.

Hashing Experimental Studies and Analysis. Previous experimental studies for the performance of different hash functions and schemes have been provided. SMHasher [154] is a widely-known test-suite for evaluating the performance of non-cryptographic hash functions. [152] provided both theoretical and experimental analysis for cryptographic hash functions. [61] and [9] focused on experimentally studying the performance of non-cryptographic and cryptographic hash functions, respectively, on

embedded devices. [116] did an experimental evaluation for the major hash functions that are designed for sampling packets during network traffic measurements. [158] experimentally investigated how choosing a different hash function might significantly affect the blockchain performance. [3] provided a detailed experimental comparison between the performance of two hashing schemes (cuckoo hashing and quadratic probing) and two radix tree variations. [144] micro-benchmarked the performance of SIMD-aware variations of different hashing schemes. [136] is another recent comprehensive experimental study for the different combinations of hash functions and schemes. However, it only focused on non-cryptographic traditional hash functions and hash table operations. For learned models, they have been extensively benchmarked in [106] for indexing only, and not for hashing. In this paper, we try to fill this gap.

Chapter 5

Discussion

5.1 Summary

In this thesis, we saw instance-optimized variants of exact and approximate membership query data structures. The variants used ML models, capturing the data/workload distribution, in order to specialize to a particular use case.

First, we saw an instance-optimized variant of Bloom filters which are data structures for approximate point membership queries. These variants used a model to answer queries with a backup Bloom filter to correct the false negatives. In PLBF, we proposed a design to utilize the model more efficiently along with a framework to automatically tune the parameter values. The score of the model indicates how likely is it for the input to be in the set and PLBF essentially treats the inputs differently based on the score of the model. PLBF frames an optimization problem and solves it to obtain the optimal parameter values. Using these techniques, PLBF is able to achieve a better space v/s error rate trade-off compared to its competitors as shown empirically.

Second, we presented SNARF which is an instance-optimized variant of range filters, which are data structures for approximate range membership queries. SNARF uses a model to map the keys to a small domain and stores the mapped values. The model helps SNARF compress the data set while also allowing range queries. Using this structure, SNARF is able to provide a better space v/s error rate trade-off than

its competitors.

Last, we saw how in certain scenarios one can use models instead of hash functions in hash tables to improve point query latencies. For datasets with evenly spaced keys, models have lower collisions than hash functions leading to faster point lookups. Another interesting aspect of these model-based hash tables is that they are able to support range queries. For workloads with majority point queries (<10% range queries), model-based hash tables turn out to be faster than using state-of-art range indexes.

5.2 Future Work

The data structures presented in this thesis show that instance optimization is a promising direction. There are a lot more components in systems like schedulers, query optimizers, sorting, etc that can be instance optimized. We believe this is going to be an active research direction in the coming years. Various challenges need to be addressed before these instance-optimized components can become production ready. Some of these challenges are mentioned below:

Handling Data/Workload Shifts:

Instance-optimized components specialize for a particular data/workload distribution and are able to provide huge performance gains by doing so. In practice, data and workload patterns might change over time leading to a drop in the performance of these components. Sensing capabilities need to be added to these systems which can detect shifts in the distribution and then trigger the re-optimization algorithm. In the context of PLBF, the performance depends on both the data/workload distribution, hence it is important to monitor both shifts. When such shifts are identified one can trigger the parameter tuning framework of PLBF.

SNARF and learned hash tables use a model that approximates the data distribution. Both these structures operate independently of the workload distribution and hence, are robust to workload shifts. Data distribution shifts can occur over time due to inserts/deletes of keys in the set. The models used by these structures are static

and hence when data distribution shifts the performance might drop for them. This can be mitigated by triggering a rebuild in case of data shifts or by the use of an updatable model similar to ALEX[46] and PGM[56].

Supporting Complex Data Types

SNARF and learned hash tables use piece-wise linear models for approximating the data distribution. This works well for numerical keys but does not extend to other data types such as strings, embeddings, multi-dimensional values, etc. Developing models that work with such data types is an interesting research direction and initial efforts in this direction have shown a lot of promise[147, 134].

Theoretical Results and Robustness

The data structures proposed in this thesis show great empirical improvements over real-world datasets along with some preliminary theoretical results explaining the performance. However, more work needs to be done to quantify the robustness of these techniques. More rigorous theoretical analysis of these structures would help us identify scenarios where they do or do not work well. Such analysis will help in making an informed decision on when/where it makes sense to use such instance-optimized components. Further, this analysis would also help us design data structures that are more robust than current designs while offering similar performance benefits.

System Integration

These prototype data structures have been largely built in isolation and there have been efforts to integrate them into an end-to-end system. It is unclear how multiple instance-optimized components would work together. It is easy to imagine a number of learned components destructively interfering with each other. Also, it is unclear how much benefit would this specialization provide compared to a general-purpose system. SageDB[44] is a preliminary effort to build an instance-optimized system. Integrating these data structures in such a system and measuring their performance benefits is still an open challenge.

Bibliography

[1] Mohammad Alahmad and Imad Fakhri Taha Alshaikhli. Broad View of Cryptographic Hash Functions. *International Journal of Computer Science Issues*, 2013.

[2] Karolina Alexiou, Donald Kossmann, and Paul Larson. Adaptive range filters for cold data: Avoiding trips to siberia. In *Proceedings of the VLDB Endowment, Vol. 6, No. 14*, 2013.

[3] Victor Alvarez, Stefan Richter, Xiao Chen, and Jens Dittrich. A Comparison of Adaptive Radix Trees and Hash Tables. In *ICDE*, pages 1227–1238, 2015.

[4] Hyrum S. Anderson and Phil Roth. Ember: An open dataset for training static pe malware machine learning models, 2018.

[5] Austin Appleby. Murmurhash3 64-bit finalizer. `https://code.google.com/p/smhasher/wiki/MurmurHash3`.

[6] Austin Appleby. MurmurHash. `https://sites.google.com/site/murmurhash/`, 2011.

[7] Audouin Audouin and Brongniart Brongniart. Annales des sciences naturelles-vol. 7 (series-2). In *Annales des Sciences Naturelles*, volume 7, pages 42–110. Crochard, 1837.

[8] Jean-Philippe Aumasson and Daniel J. Bernstein. SipHash: A Fast Short-Input PRF. In *Progress in Cryptology - INDOCRYPT*, 2012.

[9] Josep Balasch, Bariş Ege, Thomas Eisenbarth, Benoit Gérard, Zheng Gong, Tim Güneysu, Stefan Heyse, Stéphanie Kerckhof, François Koeune, Thomas Plos, Thomas Pöppelmann, Francesco Regazzoni, François-Xavier Standaert, Gilles Van Assche, Ronny Van Keer, Loïc van Oldeneel tot Oldenzeel, and Ingo von Maurich. Compact Implementation and Performance Evaluation of Hash Functions in ATtiny Devices. In *Smart Card Research and Advanced Applications*, 2013.

[10] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Özsu. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *ICDE*, pages 362–373, 2013.

[11] Tobias Behrens, Viktor Rosenfeld, Jonas Traub, Sebastian Breß, and Volker Markl. Efficient SIMD Vectorization for Hashing in OpenCL. In *EDBT*, 2018.

[12] Djamal Belazzougui, Paolo Boldi, Rasmus Pagh, and Sebastiano Vigna. Theory and practice of monotone minimal perfect hashing. *Journal of Experimental Algorithmics (JEA)*, 16:3–1, 2008.

[13] Michael A Bender, Martin Farach-Colton, Rob Johnson, Russell Kraner, Bradley C Kuszmaul, Dzejla Medjedovic, Pablo Montes, Pradeep Shetty, Richard P Spillane, and Erez Zadok. Don't thrash: How to cache your hash on flash. *Proc. VLDB Endow.*, 5(11):1627–1637, 2012.

[14] Michael A. Bender, Bradley C. Kuszmaul, and William Kuszmaul. Linear Probing Revisited: Tombstones Mark the Death of Primary Clustering. In *IEEE Symposium on Foundations of Computer Science*, 2021.

[15] C++ Team Blog. Linker Throughput Improvement in Visual Studio 2019. `https://devblogs.microsoft.com/cppblog/linker-throughput-improvement-in-visual-studio-2019/`, 2019.

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

[17] bloom filter. Python bloom filter.

[18] Prosenjit Bose, Hua Guo, Evangelos Kranakis, Anil Maheshwari, Pat Morin, Jason Morrison, Michiel Smid, and Yihui Tang. On the false-positive rate of bloom filters. *Information Processing Letters*, 108(4):210–213, 2008.

[19] Fabiano C. Botelho, Rasmus Pagh, and Nivio Ziviani. Simple and Space-Efficient Minimal Perfect Hash Functions. In *Proceedings of the International Conference on Algorithms and Data Structures*, 2007.

[20] Fabiano C Botelho and Nivio Ziviani. External perfect hashing for very large key sets. In *Proceedings of the sixteenth ACM conference on Conference on information and knowledge management*, pages 653–662, 2007.

[21] Alex D. Breslow and Nuwan S. Jayasena. Morton filters: Faster, space-efficient cuckoo filters via biasing, compression, and decoupled logical sparsity. *Proc. VLDB Endow.*, 11(9):1041–1055, May 2018.

[22] Andrei Z. Broder and Michael Mitzenmacher. Survey: Network applications of bloom filters: A survey. *Internet Math.*, 1(4):485–509, 2003.

[23] Jehoshua Bruck, Jie Gao, and Anxiao Jiang. Weighted bloom filter. In *Proceedings 2006 IEEE International Symposium on Information Theory, ISIT 2006, The Westin Seattle, Seattle, Washington, USA, July 9-14, 2006*, pages 2304–2308. IEEE, 2006.

[24] Pedro Celis. *Robin Hood Hashing.* PhD thesis, University of Waterloo, CAN, 1986.

[25] Samy Chambi, Daniel Lemire, Owen Kaser, and Robert Godin. Better bitmap performance with roaring bitmaps. *Software: practice and experience*, 46(5):709–719, 2016.

[26] Efficient Lab CMU. https://github.com/efficient/cuckoofilter, 2020.

[27] Efficient Lab CMU. https://github.com/efficient/surf, 2020.

[28] Alessandro Colantonio and Roberto Di Pietro. Concise: Compressed 'n'composable integer set. *Information Processing Letters*, pages 644–650, 2010.

[29] Yann Collet. xxHash repository. `https://cyan4973.github.io/xxHash/`.

[30] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[31] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, page 143–154, 2010.

[32] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* The MIT Press, 2nd edition, 2001.

[33] Andrew Crotty. Hist-tree: Those who ignore it are doomed to learn. In *CIDR*, 2021.

[34] Zhenwei Dai and Anshumali Shrivastava. Adaptive learned bloom filter (ada-bf): Efficient utilization of the classifier. *arXiv preprint arXiv:1910.09131*, 2019.

[35] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. Optimal bloom filters and adaptive merging for lsm-trees. *ACM Trans. Database Syst.*, 43(4), December 2018.

[36] François Deliège and Torben Bach Pedersen. Position list word aligned hybrid: Optimizing space and performance for compressed bitmaps. EDBT '10, page 228–239, 2010.

[37] Martin Dietzfelbinger, Torben Hagerup, Jyrki Katajainen, and Martti Penttonen. A Reliable Randomized Algorithm for the Closest-Pair Problem. *Journal of Algorithms*, 25(1):19–51, 1997.

[38] Martin Dietzfelbinger, Anna Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans Rohnert, and Robert E. Tarjan. Dynamic Perfect Hashing: Upper and Lower Bounds. *SIAM J. Comput.*, 23(4):738–761, 1994.

[39] Martin Dietzfelbinger, Michael Mitzenmacher, and Michael Rink. Cuckoo hashing with pages. In Camil Demetrescu and Magnús M. Halldórsson, editors, *Algorithms - ESA 2011 - 19th Annual European Symposium, Saarbrücken, Germany, September 5-9, 2011. Proceedings*, volume 6942 of *Lecture Notes in Computer Science*, pages 615–627. Springer, 2011.

[40] Martin Dietzfelbinger and Christoph Weidling. Balanced Allocation and Dictionaries with Tightly Packed Constant Size Bins. In *Theortical Computer Science*, 2007.

[41] Peter C. Dillinger and Panagiotis Manolios. Bloom filters in probabilistic verification. In Alan J. Hu and Andrew K. Martin, editors, *Formal Methods in Computer-Aided Design, 5th International Conference, FMCAD 2004, Austin, Texas, USA, November 15-17, 2004, Proceedings*, volume 3312 of *Lecture Notes in Computer Science*, pages 367–381. Springer, 2004.

[42] Peter C Dillinger and Stefan Walzer. Ribbon filter: practically smaller than bloom and xor. *arXiv preprint arXiv:2103.02515*, 2021.

[43] Jialin Ding et al. Tsunami: A Learned Multi-Dimensional Index for Correlated Data and Skewed Workloads. In *Proc. VLDB Endow.*, 2020.

[44] Jialin Ding, Ryan Marcus, Andreas Kipf, Vikram Nathan, Ani Nrusimha, Kapil Vaidya, Alexander van Renen, and Tim Kraska. SageDB: An Instance-Optimized Data Analytics System.

[45] Jialin Ding, Umar Farooq Minhas, Badrish Chandramouli, Chi Wang, Yinan Li, Ying Li, Donald Kossmann, Johannes Gehrke, and Tim Kraska. *Instance-Optimized Data Layouts for Cloud Analytics Workloads*, page 418–431. Association for Computing Machinery, New York, NY, USA, 2021.

[46] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, David Lomet, and Tim Kraska. ALEX: An Updatable Adaptive Learned Index. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 969–984, New York, NY, USA, 2020. Association for Computing Machinery.

[47] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, David Lomet, and Tim Kraska. ALEX: An Updatable Adaptive Learned Index. In *SIGMOD*, 2020.

[48] Jialin Ding, Vikram Nathan, Mohammad Alizadeh, and Tim Kraska. Tsunami: A Learned Multi-Dimensional Index for Correlated Data and Skewed Workloads. *Proc. VLDB Endow.*, 14(2):74–86, oct 2020.

[49] Kayhan Dursun, Carsten Binnig, Ugur Cetintemel, and Tim Kraska. Revisiting Reuse in Main Memory Database Systems. In *SIGMOD*, 2017.

[50] D. Eastlake and P. Jones. US Secure Hash Algorithm 1 (SHA1). RFC 3174, IETF, 9 2001.

[51] Emmanuel Esposito, Thomas Mueller Graf, and Sebastiano Vigna. Recsplit: Minimal perfect hashing via recursive splitting. In *2020 Proceedings of the Twenty-Second Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 175–185. SIAM, 2020.

[52] César Estébanez, Yago Saez, Gustavo Recio, and Pedro Isasi. Performance of the Most Common Non-Cryptographic Hash functions. *Softw. Pract. Exper.*, 44(6):681–698, 2014.

[53] Facebook. http://myrocks.io/, 2015.

[54] Bin Fan, Dave G. Andersen, Michael Kaminsky, and Michael 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, page 75–88, New York, NY, USA, 2014. Association for Computing Machinery.

[55] Bin Fan, David G. Andersen, Michael Kaminsky, and Michael D. Mitzenmacher. Cuckoo filter: Practically better than Bloom. In *Proc. CoNEXT*, December 2014.

[56] Paolo Ferragina and Giorgio Vinciguerra. The pgm-index. *Proceedings of the VLDB Endowment*, 13, 2020.

[57] Paolo Ferragina and Giorgio Vinciguerra. The PGM-Index: A Fully-Dynamic Compressed Learned Index with Provable Worst-Case Bounds. *Proc. VLDB Endow.*, 13(8):1162–1175, 2020.

[58] Dimitris Fotakis, Rasmus Pagh, Peter Sanders, and Paul G. Spirakis. Space Efficient Hash Tables with Worst Case Constant Access Time. In *Proceedings of the Annual Symposium on Theoretical Aspects of Computer Science*, 2003.

[59] Michael L Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with 0 (1) worst case access time. *Journal of the ACM (JACM)*, 31(3):538–544, 1984.

[60] Frederick N Fritsch and Ralph E Carlson. Monotone piecewise cubic interpolation. *SIAM Journal on Numerical Analysis*, 17(2):238–246, 1980.

[61] Matthew Fritter, Nadir Ould-Khessal, Scott Fazackerley, and Ramon Lawrence. Experimental Evaluation of Hash Function Performance on Embedded Devices. In *IEEE Canadian Conference on Electrical Computer Engineering, CCECE*, pages 1–5, 2018.

[62] Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. Fiting-tree: A data-aware index structure. SIGMOD '19, page 1189–1206, 2019.

[63] R. Gallager and D. van Voorhis. Optimal source codes for geometrically distributed integer alphabets (corresp.). *IEEE Transactions on Information Theory*, 21(2):228–230, 1975.

[64] Simon Gog and Matthias Petri. Optimized succinct data structures for massive data. *Softw. Pract. Exper.*, 44(11):1287–1314, November 2014.

[65] Rodrigo González, Szymon Grabowski, Veli Mäkinen, and Gonzalo 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.

[66] Mayank Goswami, Allan Grønlund, Kasper Green Larsen, and Rasmus Pagh. Approximate range emptiness in constant time and optimal space, 2014.

[67] Thomas Mueller Graf and Daniel Lemire. Xor filters. *ACM Journal of Experimental Algorithmics*, 25(1):1–16, Jul 2020.

[68] Thomas Mueller Graf and Daniel Lemire. Xor filters: Faster and smaller than bloom and cuckoo filters. *Journal of Experimental Algorithmics (JEA)*, 25:1–16, 2020.

[69] Roberto Grossi, Alessio Orlandi, Rajeev Raman, and S. Srinivasa Rao. More haste, less waste: Lowering the redundancy in fully indexable dictionaries, 2009.

[70] Shay Gueron. Intel Advanced Encryption Standard (AES) New Instructions Set. `https://www.intel.com/content/dam/doc/white-paper/advanced-encryption-standard-new-instructions-set-paper.pdf`.

[71] Bala Gurumurthy, David Broneske, Marcus Pinnecke, Gabriel Campero Durand, and Gunter Saake. SIMD Vectorized Hashing for Grouped Aggregation. In *Advances in Databases and Information Systems*, 2018.

[72] Ali Hadian and Thomas Heinis. Shift-table: A low-latency learned index for range queries using model correction, 2021.

[73] Torben Hagerup and Torsten Tholey. Efficient minimal perfect hashing in nearly minimal space. In *Annual Symposium on Theoretical Aspects of Computer Science*, pages 317–326. Springer, 2001.

[74] Brian Hentschel, Utku Sirin, and Stratos Idreos. Entropy-Learned Hashing: 10x Faster Hashing with Controllable Uniformity. In *SIGMOD*, 2022.

[75] Stratos Idreos and Mark Callaghan. Key-value storage engines. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020*, 2020.

[76] Christopher Jonathan, Umar Farooq Minhas, James Hunter, Justin Levandoski, and Gor Nishanov. Exploiting Coroutines to Attack the "Killer Nanoseconds". *Proc. VLDB Endow.*, 11(11):1702–1714, 2018.

[77] Tamer Kahveci and Ambuj Singh. Variable length queries for time series data. In *Proceedings 17th International Conference on Data Engineering*, pages 273–282. IEEE, 2001.

[78] Andreas Kipf, Damian Chromejko, Alexander Hall, Peter A. Boncz, and David G. Andersen. Cuckoo Index: A lightweight secondary index structure. *Proc. VLDB Endow.*, 13(13):3559–3572, 2020.

[79] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter A. Boncz, and Alfons Kemper. Learned Cardinalities: Estimating Correlated Joins with Deep Learning. In *CIDR*, 2019.

[80] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. Radixspline: A single-pass learned index, 2020.

[81] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. RadixSpline: A Single-Pass Learned Index. In *Proc. of aiDM@SIGMOD*, 2020.

[82] Oliver Knill. Probability and stochastic processes with applications. *Havard Web-Based*, page 5, 1994.

[83] Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching.* Addison Wesley Longman Publishing Co., Inc., USA, 1998.

[84] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD ’18, page 489–504, New York, NY, USA, 2018. Association for Computing Machinery.

[85] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures, 2018.

[86] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. The Case for Learned Index Structures. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD ’18, page 489–504, New York, NY, USA, 2018. Association for Computing Machinery.

[87] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. The Case for Learned Index Structures. In *SIGMOD*, page 489–504, 2018.

[88] Ani Kristo, Kapil Vaidya, Ugur Çetintemel, Sanchit Misra, and Tim Kraska. The case for a learned sorting algorithm. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 1001–1016, 2020.

[89] Ani Kristo, Kapil Vaidya, Ugur Çetintemel, Sanchit Misra, and Tim Kraska. The Case for a Learned Sorting Algorithm. In *SIGMOD*, page 1001–1016, 2020.

[90] Ani Kristo, Kapil Vaidya, and Tim Kraska. Defeating duplicates: A re-design of the learnedsort algorithm, 2021.

[91] Cockroach Labs. https://github.com/cockroachdb/cockroach, 2015.

[92] Harald Lang, Viktor Leis, Martina-Cezara Albutiu, Thomas Neumann, and Alfons Kemper. Massively Parallel NUMA-Aware Hash Joins. In *In-Memory Data Management and Analysis, IMDM*, 2015.

[93] Sylvain Lefebvre and Hugues Hoppe. Perfect Spatial Hashing. *ACM Transactions on Graphics.*, 25(3):579–588, 2006.

[94] Viktor Leis, Alfons Kemper, and Thomas Neumann. The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases. In *ICDE*, 2013.

[95] Daniel Lemire and Owen Kaser. Faster 64-bit Universal Hashing Using Carry-less Multiplications. *Journal of Cryptographic Engineering*, 6:171–185, 2015.

[96] Brenton Lessley and Hank Childs. Data-Parallel Hashing Techniques for GPU Architectures. *IEEE Transactions on Parallel and Distributed Systems*, 31(1), 2020.

[97] Pengfei Li, Hua Lu, Qian Zheng, Long Yang, and Gang Pan. LISA: A Learned Index Structure for Spatial Data. In *SIGMOD*, 2020.

[98] Yongkun Li, Chengjin Tian, Fan Guo, Cheng Li, and Yinlong Xu. Elasticbf: elastic bloom filter with hotness awareness for boosting read performance in large key-value stores. In *2019 {USENIX} Annual Technical Conference ({USENIX} {ATC} 19)*, pages 739–752, 2019.

[99] PerfEvent Library. PerfEvent Library. `https://github.com/viktorleis/perfevent`, 2019.

[100] Siqiang Luo, Subarna Chatterjee, Rafael Ketsetsidis, Niv Dayan, Wilson Qin, and Stratos Idreos. Rosetta: A robust space-time optimized range filter for key-value stores. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 2071–2086, New York, NY, USA, 2020. Association for Computing Machinery.

[101] Stephen Macke, Alex Beutel, Tim Kraska, Maheswaran Sathiamoorthy, Derek Zhiyuan Cheng, and EH Chi. Lifting the curse of multidimensional data with learned existence indexes. In *Workshop on ML for Systems at NeurIPS*, 2018.

[102] Bohdan S Majewski, Nicholas C Wormald, George Havas, and Zbigniew J Czech. A family of perfect hashing methods. *The Computer Journal*, 39(6):547–554, 1996.

[103] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. Learning Scheduling Algorithms for Data Processing Clusters. In *SIGCOMM*, pages 270 – 288, 2019.

[104] Ryan Marcus, Andreas Kipf, Alexander van Renen, Mihail Stoian, Sanchit Misra, Alfons Kemper, Thomas Neumann, and Tim Kraska. Benchmarking learned indexes, 2020.

[105] Ryan Marcus, Andreas Kipf, Alexander van Renen, Mihail Stoian, Sanchit Misra, Alfons Kemper, Thomas Neumann, and Tim Kraska. Benchmarking Learned Indexes. In *Proc. VLDB Endow.*, 2020.

[106] Ryan Marcus, Andreas Kipf, Alexander van Renen, Mihail Stoian, Sanchit Misra, Alfons Kemper, Thomas Neumann, and Tim Kraska. Benchmarking Learned Indexes. *Proc. VLDB Endow.*, 14(1):1–13, 2020.

[107] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. Bao: Making Learned Query Optimization Practical. In *SIGMOD*, 2021.

[108] Kurt Mehlhorn. On the program size of perfect and universal hash functions. In *Annual Symposium on Foundations of Computer Science*, 1982.

[109] M. Mitzenmacher. Compressed bloom filters. *IEEE/ACM Transactions on Networking*, 10(5):604–612, 2002.

[110] Michael Mitzenmacher. Compressed bloom filters. *IEEE/ACM Trans. Netw.*, 10(5):604–612, 2002.

[111] Michael Mitzenmacher. A model for learned bloom filters and optimizing by sandwiching. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 464–473. Curran Associates, Inc., 2018.

[112] Michael Mitzenmacher. A model for learned bloom filters and optimizing by sandwiching. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

[113] Michael Mitzenmacher, Salvatore Pontarelli, and Pedro Reviriego. Adaptive cuckoo filters. *J. Exp. Algorithmics*, 25(1), March 2020.

[114] Michael Mitzenmacher and Eli Upfal. *Probability and computing: Randomization and probabilistic techniques in algorithms and data analysis*. Cambridge university press, 2017.

[115] Hamid Mohamadi, Justin Chu, Benjamin P. Vandervalk, and Inanc Birol. ntHash: Recursive Nucleotide Hashing. *Bioinformatics*, 32(22):3492–3494, 2016.

[116] Mayra Molina, Saverio Niccolini, and Nick Duffield. A Comparative Experimental Study of Hash Functions Applied to Packet Sampling. In *International Teletraffic Congress, ITC-19*, 2005.

[117] Michael Molloy. Cores in random hypergraphs and boolean formulas. *Random Structures & Algorithms*, 27(1):124–135, 2005.

[118] Vikram Nathan, Jialin Ding, Mohammad Alizadeh, and Tim Kraska. Learning Multi-Dimensional Indexes. In *SIGMOD*, 2020.

[119] Gonzalo Navarro and Eliana Providel. Fast, small, simple rank/select on bitmaps. In Ralf Klasing, editor, *Experimental Algorithms*, pages 295–306, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[120] Thomas Neumann and Sebastian Michel. Smooth interpolating histograms with error guarantees. In *Sharing Data, Information and Knowledge, 25th British National Conference on Databases*, BNCOD '08, pages 126–138, 2008.

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

[122] Anna Pagh and Rasmus Pagh. Uniform hashing in constant time and optimal space. *SIAM Journal on Computing*, 38(1):85–96, 2008.

[123] Anna Pagh, Rasmus Pagh, and S. Srinivasa Rao. An optimal bloom filter replacement. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '05, page 823–829, USA, 2005. Society for Industrial and Applied Mathematics.

[124] Rasmus Pagh and Flemming Friche Rodler. Cuckoo Hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[125] Shekhar Palit and Kevin A. Wortman. Perfect Tabular Hashing in Pseudolinear Time. In *IEEE Annual Computing and Communication Workshop and Conference (CCWC)*, 2021.

[126] Prashant Pandey, Michael A. Bender, and Rob Johnson. A fast x86 implementation of select. *CoRR*, abs/1706.00990, 2017.

[127] Varun Pandey, Alexander van Renen, Andreas Kipf, Ibrahim Sabek, Jialin Ding, and Alfons Kemper. The Case for Learned Spatial Indexes. In *Proceedings of the AIDB Workshop @VLDB*, 2020.

[128] Mihai Patrascu. Succincter. In *2008 49th Annual IEEE Symposium on Foundations of Computer Science*, pages 305–313, 2008.

[129] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830.

[130] Orestis Polychroniou and Kenneth A. Ross. A Comprehensive Study of Main-Memory Partitioning and Its Application to Large-Scale Comparison- and Radix-Sort. In *SIGMOD*, 2014.

[131] Mihai Pundefinedtraşcu and Mikkel Thorup. The Power of Simple Tabulation Hashing. *Journal of the ACM*, 59(3), 2012.

[132] Felix Putze, Peter Sanders, and Johannes Singler. Cache-, hash-and space-efficient bloom filters. In *International Workshop on Experimental and Efficient Algorithms*, pages 108–121, 2007.

[133] Jianzhong Qi, Guanli Liu, Christian S. Jensen, and Lars Kulik. Effectively Learning Spatial Indices. In *VLDB*, 2020.

[134] Jianzhong Qi, Guanli Liu, Christian S. Jensen, and Lars Kulik. Effectively learning spatial indices. *Proc. VLDB Endow.*, 13(12):2341–2354, jul 2020.

[135] Jack Rae, Sergey Bartunov, and Timothy Lillicrap. Meta-learning neural bloom filters. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 5271–5280, Long Beach, California, USA, 09–15 Jun 2019. PMLR.

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

[137] Christian Riegger, Arthur Bernhardt, Bernhard Moessner, and Ilia Petrov. bloomrf: On performing range-queries with bloom-filters based on piecewise-monotone hash functions and dyadic trace-trees, 2020.

[138] Ronald L. Rivest. The MD5 Message-Digest Algorithm. *RFC*, 1321:1–21, 1992.

[139] J. Andrew Rogers. AquaHash. `https://github.com/jandrewrogers/AquaHash/`.

[140] Ibrahim Sabek, Kapil Vaidya, Dominik Horn, Andreas Kipf, and Tim Kraska. When Are Learned Models Better Than Hash Functions? In *Proceedings of the AIDB Workshop @VLDB*, 2021.

[141] David Salomon. Data compression. In *Handbook of massive data sets*, pages 245–309. Springer, 2002.

[142] Stefan Schuh, Xiao Chen, and Jens Dittrich. An Experimental Comparison of Thirteen Relational Equi-Joins in Main Memory. In *SIGMOD*, 2016.

[143] Russell Sears, Mark Callaghan, and Eric Brewer. Rose: Compressed, log-structured replication. *Proceedings of the VLDB Endowment*, 1(1):526–537, 2008.

[144] Dipti Shankar, Xiaoyi Lu, and Dhabaleswar K. DK Panda. SimdHT-Bench: Characterizing SIMD-Aware Hash Table Designs on Emerging CPU Architectures. In *IEEE International Symposium on Workload Characterization, IISWC*, 2019.

[145] Malte Skarupke. Fibonacci Hashing: The Optimization that the World Forgot (or: a Better Alternative to Integer Modulo). `https://probablydance.com/2018/06/16/fibonacci-hashing-the-optimization-that-the-world-forgot-or-a-better-alternativ`

[146] Vera T Sós. On the theory of diophantine approximations. i 1 (on a problem of a. ostrowski). *Acta Mathematica Hungarica*, 8(3-4):461–472, 1957.

[147] Benjamin Spector, Andreas Kipf, Kapil Vaidya, Chi Wang, Umar Farooq Minhas, and Tim Kraska. Bounding the last mile: Efficient learned string indexing, 2021.

[148] Mike Stonebraker, Daniel J Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, et al. C-store: a column-oriented dbms. In *Making Databases Work: the Pragmatic Wisdom of Michael Stonebraker*, pages 491–518. 2018.

[149] Kazuhiro Suzuki, Dongvu Tonien, Kaoru Kurosawa, and Koji Toyota. Birthday Paradox for Multi-Collisions. In *Proceedings of the International Conference on Information Security and Cryptology*, 2006.

[150] Sasu Tarkoma, Christian Esteve Rothenberg, and Eemil Lagerspetz. Theory and practice of bloom filters for distributed systems. *IEEE Communications Surveys & Tutorials*, 14(1):131–155, 2011.

[151] Nesime Tatbul. Streaming data integration: Challenges and opportunities. In *2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW 2010)*, pages 155–158. IEEE, 2010.

[152] Jacek Tchórzewski and Agnieszka Jakóbik. Theoretical and Experimental Analysis of Cryptographic Hash Functions. *Journal of Telecommunications and Information Technology*, 2019.

[153] Jens Teubner, Gustavo Alonso, Cagri Balkesen, and M. Tamer Ozsu. Main-Memory Hash Joins on Multi-Core CPUs: Tuning to the Underlying Hardware. In *ICDE*, 2013.

[154] Reini Urban. Smhasher. `https://github.com/rurban/smhasher`.

[155] Kapil Vaidya, Subarna Chatterjee, Eric Knorr, Michael Mitzenmacher, Stratos Idreos, and Tim Kraska. Snarf: A learning-enhanced range filter. *Proc. VLDB Endow.*, 15(8):1632–1644, apr 2022.

[156] Kapil Vaidya, Eric Knorr, Michael Mitzenmacher, and Tim Kraska. Partitioned learned bloom filters. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021.

[157] Peter Van Sandt, Yannis Chronis, and Jignesh M. Patel. Efficiently searching in-memory sorted arrays: Revenge of the interpolation search? In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 36–53, 2019.

[158] Fuqin Wang, Yijiang Chen, Ruochen Wang, Akindipe Olusegun Francis, Bugingo Emmanuel, Wei Zheng, and Jinjun Chen. An Experimental Investigation Into the Hash Functions Used in Blockchains. *IEEE Transactions on Engineering Management*, 67(4):1404–1424, 2020.

[159] Minmei Wang, Mingxun Zhou, Shouqian Shi, and Chen Qian. Vacuum filters: More space-efficient and faster replacement for bloom and cuckoo filters. *Proc. VLDB Endow.*, 13(2):197–210, October 2019.

[160] Yuhan Wu, Zirui Liu, Xiang Yu, Jie Gui, Haochen Gan, Yuhao Han, Tao Li, Ori Rottenstreich, and Tong Yang. MapEmbed: Perfect Hashing with High Load Factor and Fast Update. In *SIGKDD*, 2021.

[161] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G. Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. Surf: Practical range query filtering with fast succinct tries. SIGMOD '18, page 323–336, New York, NY, USA, 2018. Association for Computing Machinery.

[162] Wenshao Zhong, Chen Chen, Xingbo Wu, and Song Jiang. Remix: Efficient range query for lsm-trees. In *FAST*, 2021.

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

[164] S. Świerczkowski. On successive settings of an arc on the circumference of a circle. *Fundamenta Mathematicae*, 46(2):187–189, 1958.