Proteus: A Self-Designing Range Filter

Eric R. Knorr*[1], Baptiste Lemaire*[1], Andrew Lim*[1]
Siqiang Luo[2], Huanchen Zhang[3], Stratos Idreos[1], Michael Mitzenmacher[1]
[1]Harvard University, [2]Nanyang Technological University, [3]Tsinghua University

ABSTRACT

We introduce Proteus, a novel self-designing approximate range filter, which configures itself based on sampled data in order to optimize its false positive rate (FPR) for a given space requirement. Proteus unifies the probabilistic and deterministic design spaces of state-of-the-art range filters to achieve robust performance across a larger variety of use cases. At the core of Proteus lies our Contextual Prefix FPR (CPFPR) model —a formal framework for the FPR of prefix-based filters across their design spaces. We empirically demonstrate the accuracy of our model and Proteus' ability to optimize over both synthetic workloads and real-world datasets. We further evaluate Proteus in RocksDB and show that it is able to improve end-to-end performance by as much as 5.3x over more brittle state-of-the-art methods such as SuRF and Rosetta. Our experiments also indicate that the cost of modeling is not significant compared to the end-to-end performance gains and that Proteus is robust to workload shifts.

CCS CONCEPTS

• **Information systems** → **Database design and models**; **Unidimensional range search**; • **Theory of computation** → **Bloom filters and hashing**.

KEYWORDS

Range Filter, Self-Designing, Sample-Based Modeling, Bloom Filter

ACM Reference Format:
Eric R. Knorr, Baptiste Lemaire, Andrew Lim, Siqiang Luo, Huanchen Zhang, Stratos Idreos, Michael Mitzenmacher. 2022. Proteus: A Self-Designing Range Filter. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22), June 12–17, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3514221.3526167

1 INTRODUCTION

The Importance of Range Filters: Range queries are a fundamental operation in big data applications. Given a set S, a range query $[a,b]$ returns the members of S within the query interval, i.e. $S \cap [a,b]$. Example applications that need range queries and handle large amounts of data include social media platforms using spatio-temporal queries to aggregate user events [3], pattern discovery and anomaly detection in time-series data streams [31], scientific spatial models [47], graph databases [16, 33] and Blockchain analytics [46]. Range queries over such data sets are expensive due to the disk or network costs required to process the data. Using a filter data structure to determine when no elements are in the query range can vastly improve performance by preventing unnecessary IO operations.

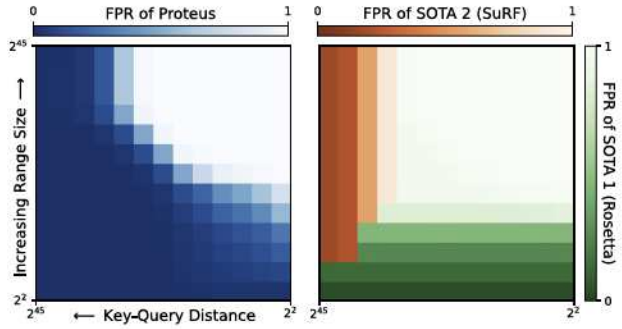


Figure 1: A self-designing filter achieves superior performance in a wide variety of workloads. (Darker is better, lower FPR.)

As an important unifying application, large-scale data systems keep large volumes of data on cheap but high latency storage devices. Answering range queries requires checking every data page that intersects with the queried range to retrieve the relevant data. For example, in widely used Log Structured Merge (LSM) tree-based key-value stores, one or more pages must be checked per level [2, 13, 19, 34, 43]. However, explicitly reading in all intersecting data pages would be far too expensive. In this setting, a compact range query data structure can be set up on a per page basis. Range query structures then act as filters, which can determine that no data in a range exists on a page (that is, we have an empty query) before resorting to accessing disk. As such, much of our evaluation considers the effectiveness of range filters in LSM-trees.

Strong Guarantees are Impractical: Most work on filtering has been focused on single key queries. Such filters are referred to as Approximate Membership Query structures (AMQs). AMQs such as Bloom filters [10] and their many variants are used to avoid the majority of unnecessary lookups for individual items in diverse applications. AMQs tolerate a small probability of false positives in order to achieve a compact representation of the key set that supports membership queries. The false positive rate (FPR) is the primary metric of a filter's performance, as false positives incur unnecessary lookups to verify their emptiness. Similar techniques can be applied to the approximate range emptiness problem to avoid unnecessary range lookups, but guaranteeing a low FPR for all potential queries can be expensive. Prior work has shown that to guarantee an FPR ϵ for range emptiness queries of range size R requires $\Omega\left(\log_2(R) + \log_2(1/\epsilon)\right)$ bits per key (BPK) [23]. Achieving

*These authors contributed equally.

SIGMOD '22, June 12–17, 2022, Philadelphia, PA, USA

ACM ISBN 978-1-4503-9249-5/22/06…$15.00
https://doi.org/10.1145/3514221.3526167

a small FPR for all queries including for large ranges therefore requires an undesirable memory budget.

Design Tradeoffs are Necessary For Performance: Current range filters aim to use less memory than theoretical worst-case guarantees require by *not* providing false positive guarantees for all queries. Instead, state-of-the-art solutions prioritize particular types of workloads, such as those with large ranges [48] or queries that are correlated to the key set [36], by making hard-coded design decisions that cater to their target use case. These range filters use heuristic methods where the FPR depends on the relationship between the key and query sets, and provable guarantees are weak or restricted to specific situations. These decisions limit their usefulness as deviation from the intended use cases results in increasingly sub-optimal performance, as seen in Figure 1. This is an issue for use cases not covered by current range filters, for example, particle physics workloads that contain long ranges and correlated queries when cross referencing time series data from multiple sensors to identify events of interest [4]. Additionally, deviations can also arise from workloads that shift over time. For instance, applications with different data based on language, such as Wikipedia, exhibit temporal skew in query distribution due to the correlation between time zone and language [22, 42]. Therefore, a robust range filter requires the ability to prioritize the desired use case by navigating the range filter design space. However, to our knowledge, there has been no prior work on formalizing the parameters of the range filter design space and the tradeoffs therein.

A Formal Framework Allows for Informed Designs: The standard metric for FPR analysis is worst-case performance which fails to capture the nuances of range filter design choices because they optimize for specific use cases. We introduce the Contextual Prefix FPR (CPFPR) model which formalizes how FPR varies across the design space of prefix-based filters. We focus on the prefix filter design space as this encompasses all state-of-the-art range filter designs [1, 27, 36, 48]. The nuances of each design are captured by expressing their FPR in terms of use case features, such as query range size and the proximity between keys and queries. These features can be concisely described by the characteristics of shared prefixes and the number of unique prefixes of a given length. We then apply the CPFPR model to techniques used by state-of-the-art range filters in order to understand their design tradeoffs.

Self-Designing Approximate Range Filters: We use insights from the CPFPR model to develop a novel class of readily optimized range filters, that we have dubbed Protean Range Filters (PRFs). PRFs navigate some portion of the range filter design space and aim to choose the best design available for a given use case. We introduce a novel PRF filter, Proteus, that spans a large portion of the prefix-filter design space and makes use of the CPFPR model to navigate it. Not only does Proteus encompass the design spaces of state-of-the-art range filters, but it is also able to combine their designs in a complementary fashion for even greater effect. The user need only supply a sample of example queries to be fed into the CPFPR model. Increased design flexibility paired with automated optimization allows Proteus to outperform more constrained designs in the vast majority cases, even when said designs are optimally tuned. We can see in Figure 1 that Proteus achieves a low FPR on a larger region of the workload space as compared to the state-of-the-art range filters which are only optimal within confined, mostly disjoint regions.

Contributions: Our contributions are as follows:

- *Novel Range Filter:* We present a novel range filter that can instantiate configurations from across the current range filter design space to meet the needs of various workloads within a limited memory budget.
- *Formalization of the Range Filter Design Space:* We introduce the CPFPR model which captures the tradeoffs in the design space of prefix-based range filters. We use this model to break down and incorporate the design elements of state-of-the-art range filters into our novel range filter **(Section 3)**.
- *Efficiently Leveraging Context:* We demonstrate how to practically navigate the prefix filter design space **(Section 4)**.
- *Model Validation:* We validate the accuracy of our model and demonstrate its ability to optimize across the prefix filter design space in a wide variety of settings using both synthetic workloads and real world datasets **(Section 5)**.
- *Robust End-to-End Gains:* We show how PRFs can be integrated in a real world system, RocksDB, and demonstrate PRFs' robustness to changing workloads as well as end to end latency improvements of up to 3.9x on 64-bit integers **(Section 6)** and 5.3x on strings **(Section 7)**.

2 THE RANGE FILTER DESIGN SPACE

Current solutions to the approximate range emptiness problem, or Approximate Range Emptiness structures (AREs), use one of two fundamental design elements: probabilistic Approximate Membership Query data structures (AMQs) and deterministic search trees. Both strategies use prefixes of the key set to encode the key space at different granularities.

AMQs are natural building blocks as they are designed to reduce unnecessary lookups by using a compact representation of the data set that is small enough to fit in memory and supports fast membership queries. A low non-zero false positive probability is tolerated in order for the representation to be sufficiently compact, but never a false negative. The search tree approach is more novel and explicitly encodes prefixes of the key set as a trie.

2.1 Probabilistic Prefix Filters

One method used by current AREs is to include one or more AMQs that encode regions of the key space with a single hashed value. There are many examples of AMQs, such as Bloom Filters, Cuckoo Filters, Quotient Filters, Xor Filters and Ribbon Filters [8, 10, 18, 20, 24], which we will collectively refer to as Bloom filters. Though their specifics vary, Bloom filters generally make use of one or more hash functions to encode the key set as a compact array. Hashing allows for fast individual item lookups with a low probability of false positives; however, valuable ordering information is lost in the hashing process. In order to know whether a range $[l, r]$ contains members of the key set, every key from l to r would need to be queried individually and the probability that at least one of them results in a false positive is then proportional to $r - l$. As such, the usefulness of Bloom filters for such queries rapidly declines with the size of the range being queried.

This can be compensated for by hashing prefixes of the key set rather than each individual key. Hashing a given prefix encodes that there is at least one member of the key set with the given prefix.

This retains the benefits of hashing while allowing queries to the Bloom filter to rule out entire regions of the key space at a time.

Prefix Bloom Filters: Prefix Bloom filters have been in use for some time now, particularly in the context of network routing [17]. By hashing a key prefix of length l bits, the prefix Bloom filter encodes a region of the key space of size 2^{k-l}, where k is the maximum key length. Using this strategy, a range of the key space can be queried by querying the Bloom filter for each region that overlaps with the desired range. This is well suited to situations with clustered target ranges, like IP addresses in network routing, as the prefix length used can be tuned to the cluster sizes. Prefix Bloom filters are also used in key-value stores, but the constraint of encoding the key space as fixed size, prefix defined regions limits their usefulness when the target ranges are not well known. In particular, prefix Bloom filters are very sensitive to the choice of prefix length which we will discuss in more detail in Section 3.

Rosetta: Rosetta [36] is an ARE aimed specifically at range queries in database systems using LSM trees, such as RocksDB [27]. These systems have used prefix Bloom filters with shorter prefixes to filter out large range queries; however, if they are not properly tuned, these prefix filters perform very poorly on small to medium range queries as well as individual key queries. This becomes even worse if the queries are correlated to the key set and empty queries tend to fall close to the key set.

Conceptually, Rosetta encodes the nodes of an implicit segment tree, or binary trie, representing the entire key space. All nodes of a given depth present in the key set are encoded by hashing the corresponding prefixes into a single Bloom filter. The prefix filter encoding the leaf nodes of the tree is then equivalent to a Bloom filter populated with full key hashes. A range is queried by checking for the presence of each node in the sub-tree corresponding to the that range in depth first order. If any node is not present in its respective prefix filter, the entire sub-tree rooted at that node is known to be empty and is not queried further. If a given node may be present, the sub-tree continues to be queried in depth-first order until either a leaf node is reported as present, resulting in a positive range query, or the entire sub-tree is found to be empty.

In practice, Rosetta does not encode every level of this tree and is configured by apportioning the total memory budget between the prefix Bloom filters encoding each prefix length. In particular, Rosetta typically allocates all of its memory budget to the last few prefix lengths. This allocation strategy works well for small ranges and point queries, regardless of proximity to the key set, but larger range queries will still require many Bloom filter queries to cover the query range and performance trends towards that of an AMQ.

2.2 Deterministic Prefix Filters

Other solutions forgo the benefits of hashing to retain as much ordering information as possible. These AREs typically make use of an explicitly encoded search tree which performs a similar role to the implicit tree encoded by Rosetta. In particular, tries are a well studied method of encoding prefixes in a succinct search tree [6, 9, 21]. In order to fit these trees in memory, they must be pruned down to prefixes of the key set. As before, each prefix found in the tree represents a range that has at least one member of the key set, while any prefix not found in the tree will not be present in the key set. The pruning therefore introduces the possibility of false positives due to common prefixes between keys and non-keys.

These trees are more flexible than prefix Bloom filters in certain aspects as they can easily encode prefixes of different lengths within the same structure and all sub-prefixes are also retained. This comes with the downside of having to pay more memory for longer prefixes, since the entire prefix must be explicitly stored, as opposed to prefix Bloom filters which can dedicate available memory to any individual prefix length.

SuRF: The Succinct Range Filter (SuRF) [48] is a state-of-the-art ARE that encodes prefixes of the key set as a Fast Succinct Trie (FST). It is a static filter and does not adapt to queries. The FST in SuRF combines the LOUDS-Dense and LOUDS-Sparse representations from [29] to achieve an efficient trie encoding (LOUDS-DS) which allows SuRF to encode longer prefixes than other trie-based prefix filters on the same memory budget. This encoding can also be searched in constant time, so the query time is independent of the size of the range. Despite this, encoding every full key in the FST is still typically too expensive, so SuRF's base configuration prunes the branch for each key to the minimum length prefix that uniquely identifies it in the key set. If there is additional memory available it can be used to extend these prefixes. It can also be used to store hashes of the key suffixes to help rule out individual keys, though these do not provide any additional benefit for range queries.

Because SuRF is configured purely based on the key distribution, sparsely populated regions of the key space are encoded more coarsely. As with other prefix filters, these coarsely encoded regions are not well suited for filtering queries that are close to the key set. The requirement that each prefix be at least long enough to be uniquely identified in the key set also means that SuRFs minimum memory usage is determined by the distribution of the keys. Despite the encoding being very compact, this can still pose an issue in situations where tight memory budgets must be strictly maintained.

3 A CONTEXTUAL PREFIX FPR MODEL

In this section we will formalize how different aspects of a workload affect the performance of prefix-based range filters in order to understand the tradeoffs of different designs. We use this framework to break down the fundamental components of state-of-the-art AREs and realize a unified design space.

3.1 The Importance of Prefix Lengths

We begin by modeling a standard prefix Bloom filter as this forms the basis for our more complex structures. When considering a single prefix Bloom filter, the only parameter to configure is the choice of prefix length. Figure 2 shows how a set of 4-bit keys can be encoded using 1-4 bit prefixes. Each prefix hashed into the filter encodes that at least one member of the key set contains that prefix; therefore, a short prefix encodes many values while a long prefix may only encode a few.

Consider a key space $\mathcal{K}$ with total order $\preceq$ [1] and a prefix Bloom filter encoding a key set $K \in \mathcal{K}$ with prefix length l and an FPR of p. We use the following terms and notation:

[1] The total order $\preceq$ depends on the types of keys being used. For instance, integer keys would use the standard less-than-or-equal relation, while string keys would use lexicographical ordering.

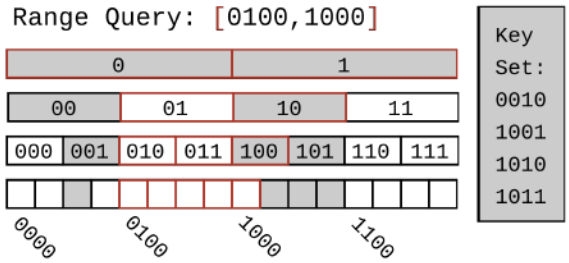


Figure 2: Encodings of a 4-bit key space using 1-4 bit prefixes and the respective prefixes required to cover the range `[4,8]` (outlined in red). Regions containing at least one key are shaded grey.

Q: An empty range query, i.e. an interval of $\mathcal{K}$ s.t. $Q \cap K = \emptyset$.

X_i: For an arbitrary set of values X and integer i, X_i is the set of unique prefixes y s.t. $len(y) = i$ and y is a prefix of some $x \in X$. Figure 2 shows Q_l bordered in red for $Q =$ `[0100,1000]` and each $l \in \{1, 2, 3, 4\}$.

LCP: The longest common prefix. For two arbitrary sets, X, Y, we define $lcp(X, Y)$ as the longest LCP between any pair $\{x, y | x \in X, y \in Y\}$.

As described in Section 2, a prefix Bloom filter returns negative only if each prefix in Q_l returns negative. If $Q_l \cap K_l$ is non-empty, then Q is guaranteed to result in a false positive because it contains a prefix used to encode K. Whether this happens is dependent on the proximity of Q and K which we can express in terms of $lcp(Q, K)$. If $lcp(Q, K)$ is greater than the prefix length used to encode the key set, then the corresponding members of the key and query set will be indistinguishable from each other. This can be seen in Figure 2 where only the 4 bit encoding of the key space can distinguish `1000` from `1001` as they only differ in the 4th bit. Putting all of this together, we can express the probability of a range query false positive in terms of the prefix filter's point query FPR, the range size and proximity of the query to any key, as shown in Equation 1.

$$P_{fp}(Q) = \begin{cases} 1 - (1-p)^{|Q_l|}, & lcp(Q,K) < l \\ 1, & l \leq lcp(Q,K) \end{cases} \quad (1)$$

With only a single prefix Bloom filter, there is a clear contention between how many regions must be queried (a short l), and whether queries close to the key set can be distinguished from it (a long l). One option to address this would be to split the memory budget between multiple Bloom filters. The downside of this approach is that each Bloom filter may only be able to contribute to a portion of the queries. This can be justified if there is a substantial divergence in types of queries such that a single prefix length will not be able to handle all types of queries effectively. Take for instance a bimodal distribution of small queries in close proximity to the key set and large non-key-correlated queries. A single filter tuned to either half of the query set would be effectively useless for the other half. However, this requires orders of magnitude of difference between range sizes or key-query correlations for even two filters to be justified as a less performant filter that can contribute to all or most of the queries will result in better overall performance. If the distribution is split between additional modes, the possible difference between each decreases and the benefit of further subdividing our memory becomes increasingly marginal. As such, we will only consider up to one additional Bloom filter.

Consider then the same setting as before but with two Bloom filters for prefix lengths $l_1 < l_2$ with point query FPRs p_1 and p_2 respectively. We will have to consider several cases depending on how the regions encoded by the first Bloom filter align with Q, for which we will use the following additional terms and notation:

I_0, I_1: Indicator variables for whether the ranges defined by the first and last members of Q_{l_1} are *not* fully contained in Q.

I_2, I_3: Indicator variables for whether the first and last members of Q_{l_1} are *not* in K_{l_2}. If $|Q_{l_1}| = 1$ and $Q_{l_1} \subseteq K_{l_1}$, let $I_2 = 1$ and $I_3 = 0$. Note that we cannot have $I_0 = I_2 = 0$ or $I_1 = I_3 = 0$ since Q is empty.

L, R: The sets of l_2 prefixes that intersect with Q and contain the first and last prefixes of Q_{l_1} respectively.

$P_{l_1}(i)$: The probability that i of the l_1 prefixes completely within Q return false positives. This can be expressed as the probability mass function for the binomial coefficient with $n = |Q_{l_1}| - I_0 - I_1$ and $p = p_1$ as shown in Equation 3.

$\bar{p_L}, \bar{p_R}$: The probability that L or R respectively are not resolved at the first Bloom filter and return only negatives at the second, given by Equation 2.

We now determine the probability of a false positive using two Bloom filters. As before, if $l_1 < l_2 \leq lcp(Q, R)$, then Q is guaranteed to result in a false positive. We therefore consider the case when $lcp(Q, R) < l_2$. Since $Q \cap K$ is empty, any prefix of Q_{l_1} such that the corresponding set of values is fully contained in Q either yields a false positive or a true negative. When a false positive for such a prefix occurs, this yields $2^{l_2 - l_1}$ queries that need to be done at the second Bloom filter. However, Q_{l_1} can also share at most two common prefixes with K, those at either end of the query; therefore, these end prefixes may yield false positives or true positives. Also, these end prefixes, being the only ones that may not be completely within Q, result in a number of queries at the second Bloom filter that depends on the overlap between Q and the prefix whenever a positive (false or true) is returned.

The end prefix cases are covered by the $\bar{p_L}$ and $\bar{p_R}$ terms defined in Equation 2. We then must also sum over the remaining possible combinations of false positives resulting from Q_{l_1}. Here Equation 3 gives the probability $P_{l_1}(i)$ of having i false positives that each result in $2^{l_2 - l_1}$ queries at l_2. For each of these we use the complement of the probability that all l_2 queries return negative. Combining these we obtain Equation 4.

$$\bar{p_L} = p_1^{I_2} \cdot I_0 \, (1-p_2)^{|L|}, \bar{p_R} = p_1^{I_3} \cdot I_1 \, (1-p_2)^{|R|} \quad (2)$$

$$P_{l_1}(i) = \binom{|Q_{l_1}| - I_0 - I_1}{i} p_1^i (1-p_1)^{|Q_{l_1}| - I_0 - I_1 - i} \quad (3)$$

$$P_{fp}(Q) = 1 - \bar{p_L} - \bar{p_R} - \sum_{i=0}^{|Q_{l_1}| - I_0 - I_1} P_{l_1}(i) \left((1-p_2)^{i 2^{l_2 - l_1}} \right) \quad (4)$$

3.2 Tractable Tries

By using two Bloom filters with different prefix lengths, one can reasonably address divergent query workloads, but their probabilistic nature can still pose issues. As discussed before, the longer prefix length is not well suited for large ranges, and any l_1 prefix fully

within Q that results in a false positive will require $2^{l_2-l_1}$ prefixes queries at l_2. If there is much difference between l_1 and l_2, which is likely if two Bloom filters are justified, then the second Bloom filter will have a very low probability of catching larger ranges missed by the first. We could reduce the number of false positives at the first Bloom filter by allocating it a larger proportion of the memory, but this also reduces the filter's ability to deal with correlated queries since this memory must be taken from the second Bloom filter.

Alternatively, using a uniform depth trie at l_1 puts a hard limit on the number of prefix queries that may be required at l_2 for any range query. Consider then the same situation as before but l_1 now represents the prefix length of the trie and p is the FPR of the Bloom filter. Since the trie is deterministic, only prefixes that match the leaf nodes will ever make it to the Bloom filter. As discussed prior, only the first and last members of Q_{l_1} can ever match the key set, so no more than $2^{l_2-l_1+1}-2$ prefixes will need to be queried at l_2 for a given range query. The probability of a false positive is then given by Equation 5.

$$P_{fp}(Q) = \begin{cases} 0, & lcp(Q,K) < l_1 < l_2 \\ 1-(1-p)^{I_2|L|+I_3|R|}, & l_1 \le lcp(Q,K) < l_2 \\ 1, & l_1 < l_2 \le lcp(Q,K) \end{cases} \quad (5)$$

Not only is Equation 5 simpler to compute than Equation 4, but it will achieve a better FPR for any combination of l_1 and l_2 assuming that the l_2 Bloom filter receives the same amount of memory in each. This does come with the limitations that l_1 now has a fixed memory cost for each possible length. There is then a hard limit on how long l_1 can be and the longer it is, the less memory is available for the l_2 Bloom filter. Despite this, a short l_1 is often cheaper to store explicitly as a trie when compared to the memory a Bloom filter would require to perform comparably. Tries are particularly efficient when representing clustered data as there will be fewer unique prefixes, but even sparse data sets are relatively cheap to represent as an FST when using a sufficiently short prefix length.

4 PROTEAN RANGE FILTERS

Protean (pro·te·an) adj. —having a varied nature or ability to assume different forms [38]

We define a Protean Range Filter (PRF) as a filter that supports approximate range emptiness queries and configures its own design to optimize performance for any given use case. We have presented the CPFPR models for three PRFs: 1PBF —a standard prefix Bloom filter (Equation 1), 2PBF —a pair of prefix Bloom filters (Equation 4) and Proteus —a hybrid filter that uses both a trie and a prefix Bloom filter (Equation 5). Other than the use of the respective CPFPR models, 1PBF operates as described in Section 2 while 2PBF is equivalent to an instance of Rosetta that uses only 2 filters. As such, this section will focus primarily on the structure of Proteus and its respective model. The models for 1PBF and 2PBF are implemented in a similar fashion. We also provide a breakdown of the costs associated with using the model and how these compare for each. We assume fixed length, integer keys in this section and discuss variable length keys in Section 7.

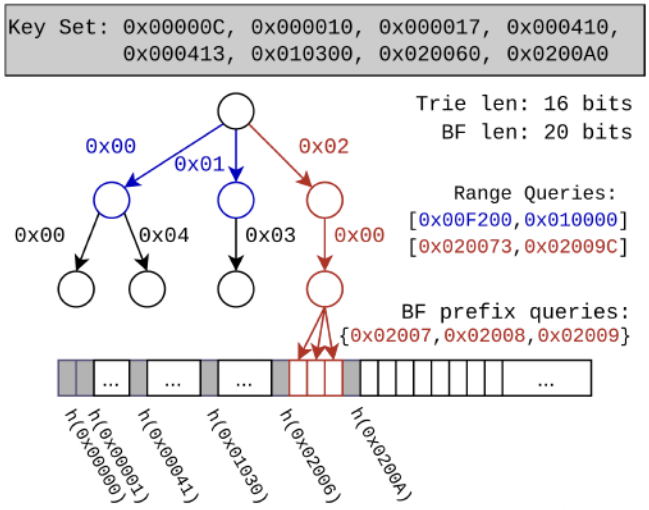


Figure 3: An example of Proteus using 24 bit keys with a trie depth of 16 bits and a Bloom filter prefix length of 20 bits. The blue and red show the logical paths of two empty range queries, the first of which is resolved in the trie while the second is resolved in the Bloom filter and could result in a false positive.

4.1 Hybrid Architecture

Unlike the other filters we have discussed, Proteus uses both probabilistic and deterministic encodings of the key space: an FST and a prefix Bloom filter. As opposed to SuRF, the FST in Proteus does not encode a unique prefix for every key, but rather all unique key prefixes of a fixed length, resulting in a FST with uniform depth. Recall that SuRF prunes the branch for each key to the minimum length prefix that uniquely identifies it in the key set and can extend these prefixes with explicitly stored key bits. Similarly, in the Proteus FST, any trie branch encoding a single key is extended to the chosen trie depth by explicitly storing the additional key bits, rather than using the LOUDS-DS trie encoding. A toy example of a Proteus encoding a 24 bit key space is shown in Figure 3 using a trie depth and Bloom filter prefix length of 16 and 20 bits respectively.

A uniform trie depth is used in part because it simplifies the modeling process, but also because we believe it to be a better use of memory. The intuition here is that sparse regions of the key space that use a coarse encoding are highly susceptible to false positives since they encode very large empty sub-regions. Similarly, encoding a densely populated portion of the key space at a coarse granularity will only result in small empty sub-regions that are less likely to cause false positives. SuRF requires a unique prefix for each key in order to support queries other than range emptiness such as range counts and sums, but this also imposes a minimal memory requirement and limits its usefulness for situations with strict memory constraints. Alternatively, a uniform depth can be adjusted to fit within any memory constraint while any leftover memory can still be put to use in the Bloom filter. While Proteus does not support range queries other than emptiness queries, replacing the Bloom filter with a counting Bloom filter would provide this functionality [11]. The Bloom filter is assigned a prefix length between the maximum key length and the depth of the trie.

Algorithm 1 The modeling process for prefix length selection.

```
Input K - the key set
Input k - the maximum key length in bits
Input m - the memory budget in bits
Input S - the set of empty sample queries
Output (l1, l2) - prefix lengths for the best FPR
Function trieMem(l) - returns size of trie with depth l.
Function BFfpr(m, n) - returns Bloom filter FPR for m bits and n elements.
1:  procedure MODEL(K, k, m, S)
2:      unsigned int resolvedInTrie[k]
3:      map nRegionsQueried[k][k]                  ▷ stores pairs (nRegions, count)
4:      for Q in S do
5:          minLen ← lcp(K, Q)                     ▷ LCP is the min granularity to filter Q
6:          for tLen ← 0 such that trieMem(tLen) ≤ m do
7:              if tLen > minLen then
8:                  resolvedInTrie[tLen]++
9:              for bLen ← tLen + 1 up to k do
10:                 if tLen < minLen && bLen > minLen then
11:                     nRegionsQueried[tLen][bLen].increment(I2|L| + I3|R|)
12:     minFPR ← 1
13:     for tLen ← 0 such that trieMem(tLen) ≤ m do
14:         tFPR ← 1 - (resolvedInTrie[tLen] / |S|)
15:         if tFPR ≤ minFPR then                  ▷ Can be changed to tFPR < minFPR
16:             minFPR ← tFPR
17:             (l1, l2) ← (tLen, 0)
18:         for bLen ← tLen + 1 up to k do
19:             bFPR ← BFfpr(m - trieMem(tLen), |K_bLen|) ▷ Any BF can be used
20:             FPR ← 0
21:             queries ← resolvedInTrie
22:             for (nRegions, count) in nRegionsQueried[tLen, bLen] do
23:                 FPR ← FPR + nRegions * bFPR * count
24:                 queries ← queries + count
25:             FPR ← FPR + (|S| - queries) / |S|
26:             if FPR ≤ minFPR then               ▷ Can be changed to FPR < minFPR
27:                 minFPR ← FPR
28:                 (l1, l2) ← (tLen, bLen)
29:     return (l1, l2)
```

Proteus reaps the benefits of both encodings while also mitigating their shortcomings. The trie is able to efficiently rule out large ranges in constant time, while the prefix Bloom filter can be positioned to catch most of the queries that would fall within the empty sub-regions encoded by the trie. Careful choice of the Bloom filter's prefix length will also improve the prefix Bloom filter's performance independent of the trie. When exploring the configuration space, Proteus is free to dedicate its entire memory budget to either encoding and so can be either entirely probabilistic or deterministic depending on the context. Configuring Proteus amounts to choosing the prefix lengths for the trie and Bloom filter. The possible prefix lengths of the trie are limited by the memory budget while any memory not used in the trie can be assigned to any single prefix length using the Bloom filter.

4.2 Operations

Queries in Proteus are carried out by searching the combined structure for any members of Q_{l_2} in depth-first order. If any prefix $x \in Q_{l_1} \cap K_{l_1}$ is found in the trie, then the prefixes $y \cap Q_{l_2}$ s.t. x is a prefix of y are queried in the Bloom filter. If any of these prefixes are present in the Bloom filter or return a false positive, the query ends and returns positive. If all of these prefixes return negative at the Bloom filter, the query continues to the next matching leaf node in the trie. If there are no more valid leaves in the trie and all queried Bloom filter regions have returned negative, the query returns negative. Queries that land sufficiently far from the key set are then ruled out in the trie as $lcp(Q, K)$ will be short, while queries that land closer to the key set must rely on the Bloom filter. In Figure 3, the query Q = `[0x00F200, 0x010000]` (blue) is an example of the former. The trie is searched for any member of Q_{l_1} = `[0x00F2, 0x0100]`, but none are found so no prefixes are queried at the Bloom filter and the query returns negative. However, for Q = `[0x020073, 0x02009C]` (red), a matching prefix is found in the trie, `0x0200`, so the members of Q_{l_2} with this prefix, {`0x02007, 0x02008, 0x02009`}, must all be queried at the Bloom filter. In this case we have that $lcp(Q, K) < l_2 \implies Q_{l_2} \cap K_{l_2} = \emptyset$, so the query will return negative so long as none of the prefixes queried at the Bloom filter result in false positives.

4.3 Using the CPFPR Model

Proteus determines its configuration by calculating the expected FPR for each possible configuration and choosing the one resulting in the lowest FPR, as shown in Algorithm 1. This involves extracting and storing the necessary information from a set of empty sample queries and the key set, then using it to compute Equation 5 for the desired memory budget.

Bloom Filter FPR: The false positive probability p in Equation 5 is dependent on the type of Bloom filter being used. We implemented Proteus using a standard Bloom filter [10] for simplicity and calculate p according to Equation 6, where n is the number of key prefixes in the filter, m is the number of bits allocated to it, and $\lceil m/n \cdot \ln(2) \rceil$ hash functions are used.[2]

$$p = \left(1 - e^{-\ln(2)}\right)^{\lceil m/n \cdot \ln(2) \rceil} \tag{6}$$

Note that both Equation 5 and Algorithm 1 are AMQ-agnostic. The Bloom filters in our PRFs can be replaced with any AMQ, and we need only use the corresponding FPR formula.

Count Key Prefixes: As the Bloom filter FPR is dependent on the number of elements, we must count the number of unique key prefixes, $|K_l|$, for all possible prefix lengths l. This can be done by computing the successive LCPs of the sorted key set as each successive LCP indicates the minimum prefix length at which a key is uniquely represented. This is an $O(|K|)$ operation, assuming the keys are already sorted. In our example application, RocksDB —described in Section 6 —the keys are sorted internally for the filter.

Calculate Trie Memory: The number of unique key prefixes $|K_l|$ is also used to estimate the size of the trie at each depth l. This estimation is based on the implementations of LOUDS-Sparse and LOUDS-Dense as described in [48]. We also implemented a method to accurately calculate the trie size, but this dominated the combined modeling and construction cost of the filter and provided little benefit. As is, we overestimate the cost of the trie because we do not consider the memory saved by using explicitly stored key bits after a key has been uniquely represented in the FST. When the trie is short, this has little to no effect as very few keys will be uniquely represented. This error then grows with the depth of the trie, but any leftover memory is simply allocated to the Bloom filter. We also use this to approximate the ideal number of FST levels that should be encoded with LOUDS-Dense and LOUDS-Sparse respectively,

[2] A max limit of 32 hash functions is imposed since m/n can be quite high for short prefix lengths resulting in very large hash function counts that are not practical when making multiple prefix queries. We use the MurmurHash3 and CLHASH hash functions for integer and string workloads respectively [35].

$N\delta^2$	1	2	3	4	5
Bound	0.00425	0.00132	0.00005	0.000002	0.0000001

Table 1: Bounds for $e^{-N\delta^2/(2p)} + e^{-N\delta^2/(3p)}$ for different values of $N\delta^2$, $p \leq 0.1$.

rather than relying on a fixed ratio as SuRF does. This allows our FST to be even more memory-efficient than SuRF.

Count Query Prefixes: Here we obtain the relevant metrics from our sample queries. For each $Q \in S$, we must determine which trie depths will resolve the query as well as the number of regions required to cover Q for each possible prefix length l, $|Q_l|$. The first of these requires $lcp(K, Q)$ which entails searching K for the nearest member to Q. In the worst case, computing this for all queries is $O(|S| \log_2 |K|)$, but we sort the left query bounds, which costs $O(|S| \log_2 |S|)$, and start each search from the key found for the previous query. For the second, we simply shift the left and right bounds of Q to each prefix length and take their difference. This is a constant amount of work for each query.

Calculate Configuration FPRs: Once we have gathered the above information from K and S, we have everything we need to compute Equation 5 for each Q and configuration. Averaging these across a given configuration gives us our corresponding expected FPR. The advantage of gathering all the information first is that the false positive probabilities for the individual queries can be batched together. Specifically, all queries with the same $|Q_l|$ for a given prefix length l have the same false positive probability and can thus be calculated together. The number of such calculations is then dependent on the number of queries with distinct prefix counts for each configuration. If range query sizes vary significantly, then most queries will have a distinct number of prefixes and the number of calculations per configuration approaches $O(|S|)$, which is not ideal.

To counteract this, we bin the query prefix counts into k bins of exponentially increasing size, where k is the maximum key length in bits. Bin i contains the number of queries with prefix counts in $[2^{i-1}, 2^i)$ as well as the average number of prefix counts for those queries and bin 0 contains the number of queries resolved in the trie. A single batch FPR calculation is then performed for each non-empty bin using the average prefix query count. Calculating the total FPR for a given configuration therefore requires at most k batch calculations. This significantly reduces the amount of modeling work and has little effect on the accuracy. This is because the probability of returning negative for an empty query decays exponentially with the number of prefix queries required. Despite containing more disparate values, the bins with larger ranges will still batch together queries with similar false positive probabilities.

Sample Size: We based our sample size on confidence intervals derived using a Chernoff bound. Using N queries, we obtain an estimate $\hat{p}$ of the FPR of a given configuration by dividing the number of false positives found by N. Assuming the N queries are independently false positives with probability p, a standard Chernoff bound (see, e.g., Chapter 4 of [41]) yields the following bound for the probability that our estimate, $\hat{p}$, is within δ of p:

$$Pr(p \in [\hat{p}-\delta, \hat{p}+\delta]) \geq 1-\min\left(2e^{-2N\delta^2}, e^{-N\delta^2/(2p)} + e^{-N\delta^2/(3p)}\right).$$

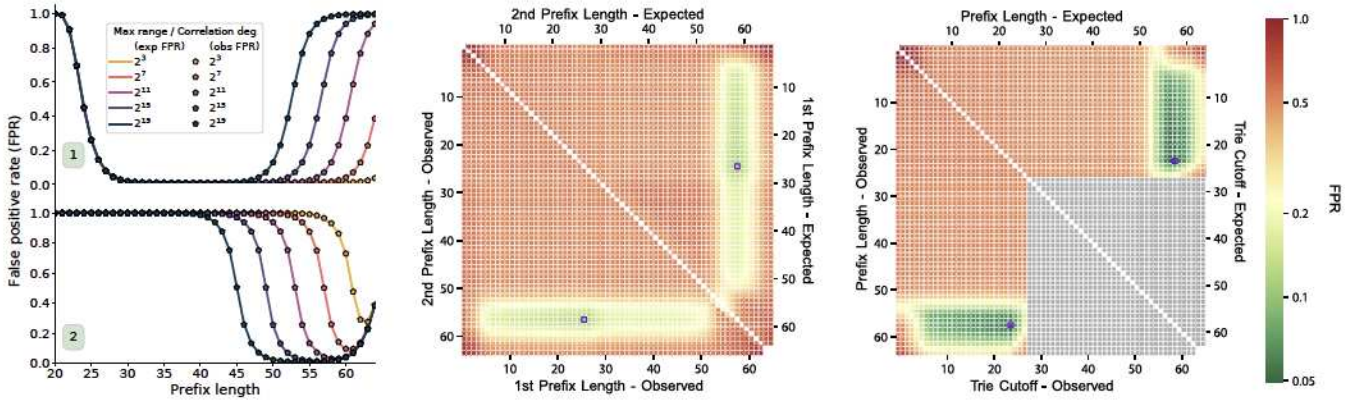
	Count Key Prefixes	*Calc. Trie Mem.*	*Count Query Prefixes*	*Calc. Config FPRs*	*Build Filter*	*Total*
1PBF	32	-	9	1	3139	3181
2PBF	32	-	81	884	3239	4236
Proteus	32	1	71	1	3130	3235
SuRF			-		482	482
Rosetta			3		4775	4778

Table 2: Breakdown and comparison of filter construction times, including modeling. Values are rounded up to nearest millisecond.

In our setting, typically $p \leq 0.1$. Table 1 provides (upper bounds for) the largest value of $e^{-N\delta^2/(2p)} + e^{-N\delta^2/(3p)})$ for $p \leq 0.1$, for different values of $N\delta^2$. For example, with sample sizes of 10,000 and 50,000 queries, the probability that $\hat{p}$ differs from p by more than 0.01 will be at most 0.00425 and 0.0000001 respectively. Note that this is an upper bound and so the actual probability will likely be smaller in practice. Furthermore, accurately estimating the FPR of each configuration is less consequential than finding a good configuration. So long as our estimates are close, we will end up with a configuration that is close to ideal. We show in Section 5.1 the accuracy of our FPR estimates over the space of possible PRF configurations for a sample size of 10K queries. In Section 5.2, we compare Proteus configured using 20K sample queries against the state-of-the-art on diverse workloads.

Modeling Cost Breakdown: Table 2 shows a breakdown of these costs for 10M normally distributed keys, a sample of 20K correlated empty queries, and a memory budget of 10 BPK. This workload is designed to maximize the number of possible configurations for Proteus and consequently the computation required for modeling. We use normally distributed keys to increase the number of viable configurations to test as there will be more common key prefixes, thereby making the trie more compact. The queries are correlated to the keys just enough that most will not be resolved in the trie which increases the number of calculations required to compute the expected FPR for each configuration. Lastly, we use range sizes uniformly distributed between 2 and 2^{20} to have a large number of distinct prefix counts. 2PBF uses a maximum range size of 2^{15} due to values overflowing when computing the binomial coefficient in Equation 4 for queries with a large number of prefixes.

Looking at the results in Table 2, we see that the modeling time for 1PBF (~42ms) is about two orders of magnitude smaller than its construction time (~3s), which is just that of a standard Bloom filter. Proteus's modeling is a modestly more expensive (~100ms) but is still dominated by the construction time (~3s). It is worth noting that, without the binning, calculating the expected FPRs for each configuration becomes the dominant factor for all of our PRFs, in the worst case. With the binning, the combined modeling and build cost of both 1PBF and Proteus is comparable to the construction cost of a standard Bloom filter. However, the modeling cost alone for 2PBF is comparable to the construction cost for a Bloom filter, even with reduced range size. This is for a number of reasons. While Proteus's potential configurations are limited by the cost of its trie, 2PBF's choice of l_1 has no such limit. 2PBF therefore considers all combinations of $l_1 < l_2 \in [1, 64]$ for multiple memory allocations.



(a) 1PBF (b) 2PBF (c) Proteus

Figure 4: The CPFPR model accurately predicts the FPR for all possible designs of different Protean Range Filters.

An exhaustive search of all possible memory allocations is infeasible, so we implement 2PBF to test 2 asymmetric allocations (60-40/40-60) and a symmetric allocation. Furthermore, the probabilistic nature of the first filter results in many possible outcomes for each query, all of which must be considered when calculating the query's false positive probability. This puts 2PBF's modeling time (~1s) on the same order of magnitude as it's construction time (~3s). Note that this is a worst case workload for modeling time but not filter build time. This is because 2PBF only ends up using a single Bloom filter while Proteus does not use a trie. However, the worst case build time for Proteus will not be significantly larger since the FST can be built very quickly, as shown by SuRF's build time. 2PBF's build time can as much as double if it has to build a second Bloom filter.

5 MODEL VALIDATION

Here we demonstrate the accuracy of the described CPFPR models and evaluate our PRFs (1PBF, 2PBF, and Proteus) as in-memory standalone filters. We show that Proteus selects optimal designs across a variety of workloads and achieves better FPRs than state-of-the-art AREs due to its broader design space. In Section 6, we evaluate these AREs in RocksDB with end-to-end system metrics. We use 64-bit unsigned integers for our experiments in Section 5 and Section 6 and focus on strings in Section 7.

Datasets: The real world datasets come from the Searched on Sorted Data (SOSD) benchmark for index structures [30, 37].

- UNIFORM: Keys are generated uniformly from `[0, 2^64 - 1]`.
- NORMAL: Keys are generated with a mean of 2^{63} and a standard deviation of $2^{64} \cdot 0.01$.
- BOOKS: Amazon booksale popularity data for 800M books. This data has a fairly heavy skew with many more low popularity scores than high.
- FACEBOOK: A set of 200M upsampled Facebook user IDs. This data is fairly dense and covers a relatively small range of values with uniformly distributed gaps.

Workloads: We test variations of YCSB Workload E, a range-scan intensive workload. Queries are of the form `[left, left+offset]`, where `offset` is chosen uniformly at random from `[2, RMAX]` unless otherwise stated. For point queries, `offset` is set to 0.

- UNIFORM: `left` is taken uniformly at random from `[0, 2^64 − RMAX]`.
- CORRELATED: A `key` is chosen uniformly at random from the dataset and then `left` is chosen uniformly at random from `[key+1, key+CORRDEGREE]`. We use a default `CORRDEGREE` of 2^{10}.
- SPLIT: An even split of UNIFORM and CORRELATED queries, similar to the particle physics workload mentioned in Section 1.
- REAL: For a real world dataset, we uniformly sample 10M integers to use as keys and another 1M integers to use as the `left` bounds.

Experimental Setup: All experiments were run on Linux kernel 5.13.12-arch1-1 with an AMD Ryzen 7 1800X 8-core processor, 16GB RAM, and 1TB Samsung 850 EVO SSD. For each experiment, 1M queries were executed serially on a single thread with a sample of 20K queries and a data set of 10M keys.

5.1 Model Accuracy

We validate the accuracy of the CPFPR models for 1PBF, 2PBF, and Proteus by comparing the expected FPR —as calculated by the corresponding CPFPR model —with the observed FPR on all possible designs in the respective filter design spaces. We use a sample size of 10K queries for these experiments, demonstrating our accuracy at the lowest $N\delta^2$ in Table 1.

1PBF: We run two experiments to highlight the impact of each of our contextual parameters, range size and correlation between keys and queries, as shown in Figure 4a. The top graph varies `RMAX` on UNIFORM-UNIFORM, while the bottom graph varies `CORRDEGREE` on UNIFORM-CORRELATED. The `RMAX` for the bottom graph is fixed at 2^7 and the prefix length on the x-axis represents the different possible designs for 1PBF.

In Figure 4a.1, we see that the observed FPR quickly increases once the prefix length passes $64 - \log_2$ `RMAX`. Before this threshold, a given range query will not require more than 2 regions to be queried in the prefix Bloom filter as the range queries are all smaller than the regions encoded the prefix filter. Since we use UNIFORM keys and queries, the significant majority of queries do not fall close to keys. As such, empty sub-regions only become an issue for prefix lengths shorter than 30. The same effect becomes more prevalent in Figure 4a.2 where we see that any prefix length shorter than $64 - \log_2$ `CORRDEGREE` is affected by empty sub-regions. Since

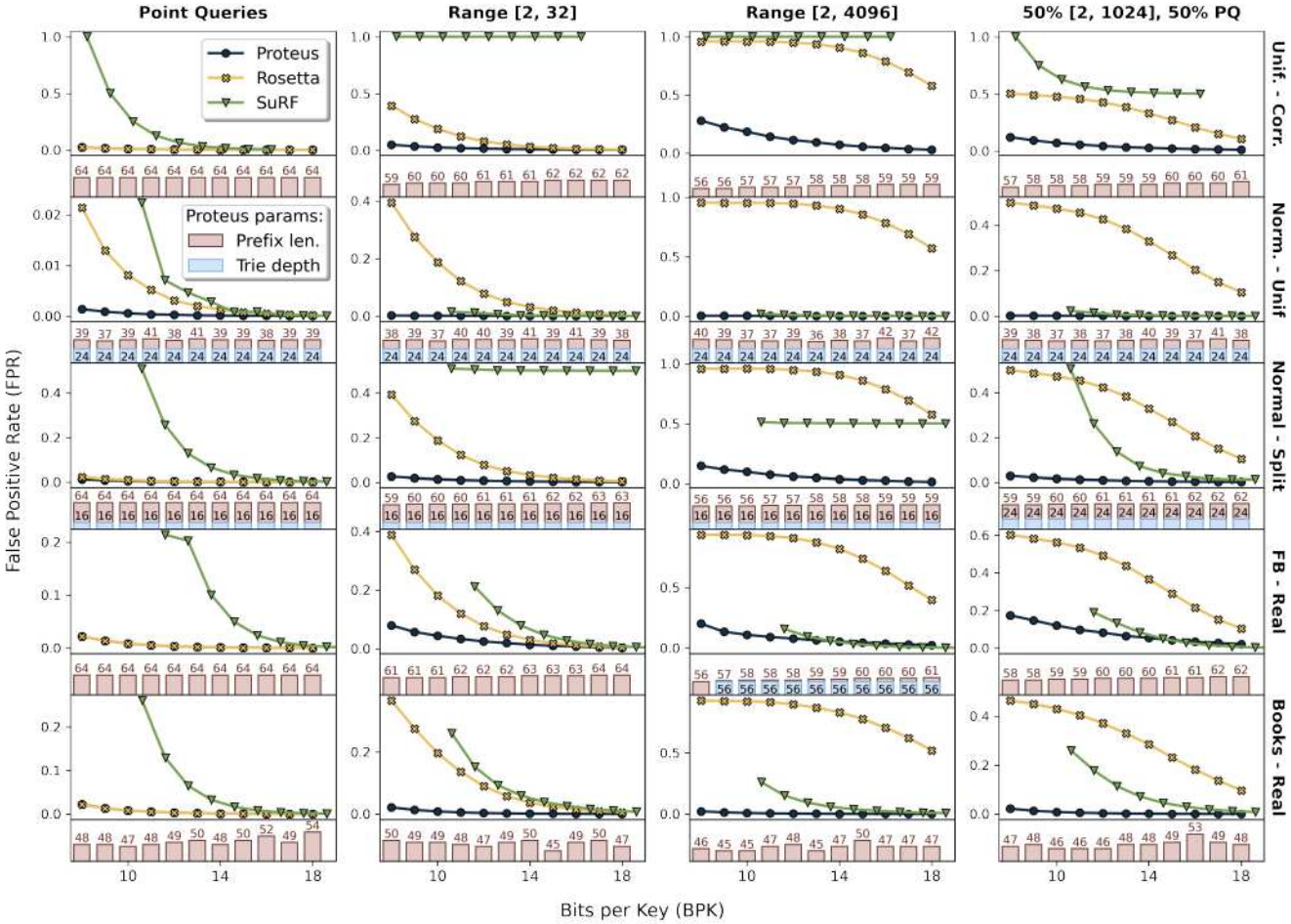


Figure 5: Proteus optimally configures its design on diverse workloads with varying range sizes and memory budgets.

Figure 4a.b uses a range size of 2^7, any prefix length longer than 57 will also result in more false positives due to range size. When $\log_2$ CORRDEGREE $>$ 7, the prefix filter must contend with false positives resulting from both empty sub-regions and range size. In both experiments, the 1PBF CPFPR model is able to accurately capture the effects of both conditions as they pertain to the FPR.

2PBF & Proteus: For both of our PRFs that use two prefix lengths, we focus on a situation that calls for multiple prefix lengths, as described in Section 3. We use NORMAL-SPLIT with short range CORRELATED and long range UNIFORM queries to necessitate the use of two prefix lengths. The FPRs for each design are shown in Figure 4b and 4c for 2PBF and Proteus respectively. The expected FPRs are shown in the upper right triangle of the matrix while the corresponding observed FPRs are reflected in the lower left.

For both filters, we correctly predict the optimal design, and accurately predict the range of FPRs over the entire design space. The optimal design for 2PBF is uses prefix lengths of 26 and 57 bits while Proteus's optimal design uses a 24 bit trie and a prefix length of 58 bits. The corresponding expected and observed FPRs are 11.4% and 12.3% for 2PBF and 5.17% and 4.91% for Proteus respectively. These values may seem high in terms of the FPRs typically achieved by AMQs, but this is also a highly adversarial case which current AREs are unable to handle. We re-visit this use case in Section 6.3 in the context of end-to-end system performance. The gray region corresponds to the part of the parameter space where the trie is too large for the memory budget (10 BPK). Despite its more limited design space, Proteus achieves lower FPRs than 2PBF.

Figure 4b shows the results for the best memory division between the two Bloom filters, a 50-50 split. This works best as our SPLIT query distribution is evenly split between small, key-correlated queries and large, uniformly distributed queries. 2PBF also considers asymmetric memory allocations, as described in Section 4.

Both Proteus and 2PBF fully encompass 1PBF's design space and will always achieve an equivalent or lower FPR. Moreover, even though Proteus and 2PBF occupy slightly different design spaces, in the situations where a second Bloom filter is helpful, it is outperformed by Proteus's trie.

5.2 Optimizing Across the Design Space

We now demonstrate Proteus's ability to select optimal designs across a variety of workloads. We also contrast Proteus flexibility against state-of-the-art AREs Rosetta and SuRF for each, as shown

in Figure 5. Each row in Figure 5 shows the results for a Dataset-Workload pair for point queries, small range queries, large range queries and a combination of point and range queries. The SuRF results show the lowest FPR for all possible configurations of real and hash-suffix bits, but in practice users will need to implement a policy to choose the appropriate SuRF configuration. In these experiments, Proteus and Rosetta both use 20K empty sample queries which gives us a bound of 0.00425 for $\delta = 0.1$, as per Table 1. This will give us higher confidence in the optimality of Proteus's chosen design as we compare its performance across workloads.

Effective Navigation of Design Space: For nearly all use cases, Proteus is able to choose a design that achieves a low FPR. This is less true for large Correlated queries —an adversarial case for any prefix-based filter —as Proteus must rely entirely on a Bloom filter design. Even so, Proteus is able to achieve a much lower FPR than any state-of-the-art ARE by picking a prefix length that balances the number of prefixes per query and the number of range queries distinguishable from the key set. In situations where SuRF and Rosetta are optimal, Proteus takes on a similar design to the respective filter and achieves similar performance. For instance, Rosetta and Proteus are virtually indistinguishable in terms of FPR in point query workloads. Similarly, Proteus and SuRF achieve very similar FPRs on Facebook-Real as the keys lie in a narrow range which causes both Proteus and SuRF to have extremely deep tries.

Impact of Restricted Design Spaces: However, if the optimal design lies outside the restricted design spaces of SuRF or Rosetta, then the corresponding filter's performance will be limited. For example, consider the Uniform-Correlated small range query workload. Despite only a small increase in range size from the point query workload, a full length prefix filter is no longer optimal for lower memory budgets as the benefit from distinguishing all queries is outweighed by the benefit of querying fewer prefixes, as corroborated by Proteus's design. Even so, Rosetta will always dedicate the majority of its memory to the full length prefix filter. For all point query workloads as well as the mixed query workloads for Uniform-Correlated and Normal-Split, SuRF achieves its best FPR with the use of hash-suffix bits and only achieves good FPR with high memory budgets. In such cases, a Bloom filter is a more efficient use of memory and can be tuned to optimize for arbitrary range sizes, in contrast to the hash-suffix bits which are only used for point queries. We can also observe SuRF's minimum memory requirement across the various workloads. In most cases, it requires 11-12 BPK, while Proteus can always achieve equivalent if not better performance at 8BPK.

Additional Benefits of the Hybrid Design Space: The combination of complementary design elements in Proteus allows it to achieve better FPR than designs which rely only on a single technique. Even for some point query workloads, such as Normal-Uniform, a hybrid design can leverage a short, memory-efficient trie to achieve a better FPR-memory tradeoff than a standard Bloom filter. Furthermore, on Split workloads, Proteus is able to gracefully handle both types of queries, but more brittle structures may have to sacrifice performance on a certain portion of the queries. For instance, on the mixed Normal-Split workload at a low memory budget, Rosetta and SuRF can only filter the Correlated point queries and the Uniform range queries respectively.

6 SYSTEM EVALUATION: ROCKSDB

We integrate Proteus into RocksDB —a popular key-value store —and demonstrate that it improves end-to-end range query performance by up to 3.9x and 3.3x over Rosetta and SuRF respectively across a variety of workloads, with consistently better performance at low BPKs (8-12). This speed-up is due to a reduction in I/O operations as a result of the lower FPR achieved. Furthermore, we show that the additional cost of modeling in filter construction does not significantly impact the end-to-end performance of RocksDB and that Proteus is able to adapt smoothly to changes in the query workload distribution, unlike Rosetta and SuRF which suffer drastic declines in performance on certain adversarial distributions.

6.1 Proteus System Integration

RocksDB uses an LSM tree architecture which organizes data on disk into levels of increasing size, where each level L_i (except L_0) is range partitioned into multiple sorted runs or Static Sorted Table (SST) files that occupy disjoint key ranges. The SST files in L_0 are directly flushed to disk from MemTables —in-memory structures that buffer writes —and thus typically have overlapping key ranges. Static filters (e.g. Bloom filters) are built on every SST file to reduce unnecessary accesses for non-existent keys. When a level L_i reaches its maximum capacity, RocksDB selectively merges SST files from L_i into L_{i+1}, triggering the construction of new filters on the merged data in the new L_{i+1} SST files. This process is called compaction.

Range Query Implementation: Similar to [36], we modify the RocksDB closed Seek logic to first check all filters for the existence of keys in the queried range. If all filters return false, then Seek returns an invalid iterator. For the filters that return true, RocksDB proceeds to retrieve the smallest keys from the associated SST files that are greater than or equal to the lower query bound. This is done by binary searching over the index block which stores min/max information in each SST file and fetching the corresponding data block. If the global smallest key is smaller than the upper query bound, an iterator pointing to that key is returned. Otherwise, an invalid iterator is returned for the empty range query.

Sample Query Queue: Since Proteus (and Rosetta) need sample queries, we create a fixed size query queue and seed it with an initial query sample. Older queries are evicted with a FIFO policy. This changing set of sample queries is used in conjunction with the keys in each SST file to determine the optimal filter design for each SST file at construction time. In our experiments, we use a queue size of 20K queries (~320KB) and update the queue with every 100th executed empty query.

Tuning RocksDB: Since all filters have to be queried during a closed Seek, we curtail this CPU cost by tuning RocksDB to fit more data in each SST file, thereby reducing the number of filters queried. This also helps to control the overall cost of modeling from filter construction when there are heavy compactions. We increase the SST file size from the default of 64MB to 256MB and scale up the size of L_1 and the MemTables to maintain the same number of SST files that fit in them with default settings.[3] Similarly, we selectively enable data compression for certain levels. We leave the few SST files in L_0 and L_1 uncompressed to maintain the speed of MemTable flushes and $L_0 \rightarrow L_1$ compactions. For SST files in L_2, we use LZ4

[3] `write_buffer_size = 256MB, max_bytes_for_level_base = 1024MB`

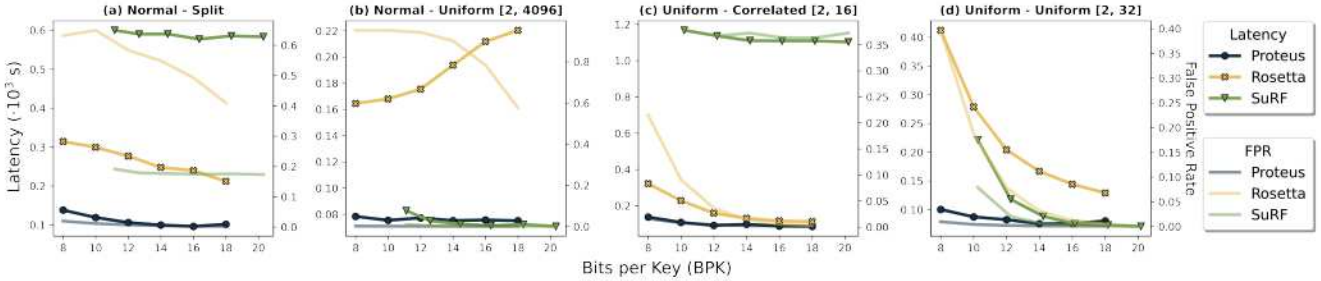


Figure 6: Proteus improves end-to-end RocksDB performance on low memory budgets across diverse workloads.

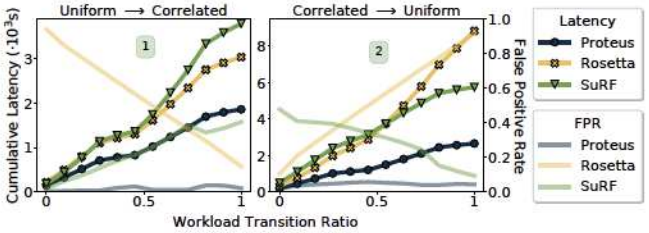


Figure 7: Proteus is robust against extreme workload shifts.

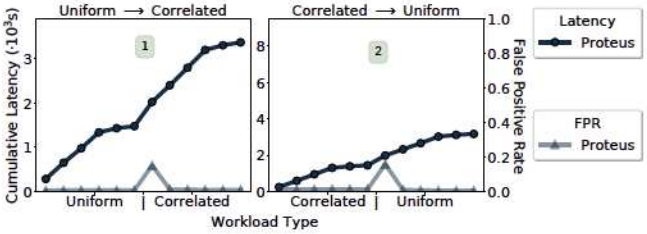


Figure 8: Proteus is robust to immediate, extreme workload shifts.

compression —a light-weight compression algorithm recommended by RocksDB [28] that balances CPU cost and compression size. We use heavy-weight ZSTD compression for SST files in L_3 onwards as they are less frequently modified and contain bulk of the data in the LSM tree. We ensure that the filters are cached in the RocksDB block cache[4] and assign RocksDB 6 background threads for flushes and compactions.

6.2 Experimental Setup

We use the same datasets, range query workloads, and machine described in Section 5 for experiments carried out in RocksDB v6.20.3. For each 8 byte integer key, we generate an associated 512 byte value. The first half of all values are zeroed out, while the second half is randomly generated which yields a constant compression ratio of 0.5. To ensure that all experiments start from a consistent LSM tree state, we manually flush the MemTable after populating the initial, empty database, and wait for all background compactions to finish before executing the benchmark. In Section 6.3, the initial database has 50M key-value pairs which yields a 4 level (~14GB compressed) LSM tree with ~70 SST files. In Section 6.4, we first insert 20M key-value pairs to get a 3 level (~6GB compressed) LSM tree with ~40 SST files, and subsequently Put an additional 40M key-value pairs over the course of the experiments. In both cases, L_0 is empty in the initial database state as we set RocksDB to compact all L_0 SST files to L_1 for sake of consistency. Lastly, we warm the RocksDB block cache (1GB) and the OS page cache by running 1M uniformly distributed point queries of existing keys to ensure that all indexes and filters are loaded into memory.

6.3 End-to-End Performance

We measure the end-to-end range query performance in terms of workload execution latency for Proteus, SuRF, and Rosetta on four use cases targeting different points in the design space. As shown in Figure 6, Proteus achieves the lowest latency across all workloads on low memory budgets, improving upon SuRF and Rosetta by as much as 3.3x and 3.9x respectively. In RocksDB, one of the primary sources of latency is I/O when accessing the SST files. As such, a lower FPR generally results in lower latency because we can avoid unnecessary I/O. However, situations such as thrashing in RocksDB's internal cache or excessive prefix queries during Bloom filter probes can result in higher latency despite a low FPR. For example, Rosetta improves its FPR on Normal-Uniform as its memory budget increases by using more hash functions per Bloom filter, but the resulting latency increases. This is because a large range query requires Rosetta to query many prefixes and the CPU cost for each is proportional to the number of hash functions used. Proteus is able to frequently avoid this as the trie limits - and sometimes eliminates all - prefix queries made at the Bloom filter. SuRF has an effectively constant computational cost for queries, but can suffer due to pressure on RockDB's internal cache. This can be seen in both the Normal-Split and Uniform-Correlated workloads. We observed that SuRF puts more pressure on RocksDB's internal cache which results in thrashing after passing a certain FPR threshold (~0.1 to 0.2 in our experiments).[5] This thrashing severely impacts the overall system latency. We also observed that SuRF's memory footprint varies by as much as 3 BPK across SST files, while Proteus and Rosetta maintain consistent BPKs and do not result in thrashing. Naturally, I/O savings would be magnified for larger datasets.

6.4 Robustness to Shifting Query Distribution

We now evaluate the robustness of Proteus: its ability to maintain good end-to-end performance when the workload shifts. We examine incrementally shifting workloads which mimic real-life applications with temporal skew, such as Wikipedia. As shown in

[4] `cache_index_and_filter_blocks = true, pin_l0_filter_and_index_blocks_in_cache = true`

[5] `rocksdb.block.cache.add` shows that more data blocks are added to the cache when using SuRF compared to Proteus with similar FPR.

Section 3, a range filter's performance is primarily affected by the relative positioning between the data keys and the queried keys. Thus, changing the queries or the data are essentially equivalent in terms of impact on filter performance and so we focus on changing the query distribution to be able to control the relative key-query proximities. We test with workloads that gradually shift between large range Uniform queries and small range Correlated queries which favor shorter and longer prefix lengths respectively. To magnify the difference in Proteus designs, we maintain a Normal key distribution when shifting from long Uniform queries to short Correlated queries as the trie chosen for Uniform queries is ineffective for Correlated queries and take memory away from the Bloom filter. Similarly, we maintain a Uniform key distribution for a short Correlated to long Uniform query transition which precludes the use of a trie for long Uniform queries.

For each workload, we define the workload transition ratio as the probability of executing a query from the end query distribution. We test 60M closed Seeks with a workload transition ratio increasing linearly from 0 to 1. We start with an initial database of 20M keys and uniformly interleave 40M Puts with the 60M Seeks to trigger periodic compactions and construction of new filters.

In Figure 7, we show the cumulative latency as the respective workloads transition from one type of query to another, and report the FPR for every batch of 5M Seeks. Proteus is resilient to the extreme workload shifts and is able to instantiate new designs to maintain a consistently low Seek latency. As shown in Figure 7.1 and 7.2, Proteus has a smooth increase in cumulative latency which stems from the low FPR maintained as the workload shifts. This is because Proteus can configure itself accurately by relying on the sample query queue to provide an up-to-date view of the query workload. The end-to-end behavior observed also highlights that the additional cost of modeling for Proteus during filter construction does not impact the overall performance despite heavy ongoing compactions (~15-20 for each 5M Seek batch). In contrast, the latencies for SuRF and Rosetta increase sharply when the workload transitions past 0.5 for Uniform→Correlated (Figure 7.1) and Correlated→Uniform (Figure 7.2) respectively. Due to their restricted design spaces, Rosetta and SuRF can only effectively handle one of the two types of queries. We observe the impact of their brittle designs in the FPR which decreases as the workload shifts.

We repeated the same experiments for Proteus with an immediate change in query distribution, simulated by not mixing the two distributions, with results shown in Figure 8. We observe a larger increase in FPR and latency after the drastic transition since the filter designs are not optimal for the new query distribution, but the decrease in performance is temporary as the filters are rebuilt using the updated query cache, giving robust performance.

7 VARIABLE LENGTH KEYS

Database workloads commonly include variable length keys, which often arise from concatenations of various metadata [12, 25]. In this section, we show how Proteus can be used with variable length keys and demonstrate that the CPFPR model extends to any key length. We also show that Proteus reduces end-to-end query latency in RocksDB by as much as 5.3x vs. SuRF on a real-world string dataset.

7.1 Extending the Model and Filter

Modeling: Shifting from a fixed length integer key space to a variable length key space in the CPFPR model is equivalent to changing the total ordering from a less-than-or-equal relation to a lexicographical relation. For longer keys, Proteus also has more potential designs due to the larger range of prefix lengths. In the context of static filters, the length of the longest key in the dataset is known and therefore Proteus's design space is well defined.
Filter Operation: The trie portion of Proteus handles string keys without requiring any modifications. On the other hand, variable length keys give rise to an exploding number of prefixes to query in the prefix Bloom filters. In addition, every Bloom filter query inherently increases the probability of a false positive. Proteus achieves low FPR and query time by padding short keys and queries with trailing null bytes to a chosen prefix length, thus mapping the key space onto a fixed-length key space. This means that the prefix Bloom filter does not distinguish between short keys and their padded equivalents, which will result in false positives if the application does not make the same assumption. Finally, we changed the Bloom filter hash function from MurmurHash3 to CLHASH which can handle strings [5, 35].

7.2 Validation and Evaluation

Experimental Setup: We run in-memory and RocksDB benchmarks similar to Section 5.2 and Section 6.3 respectively. For in-memory experiments, we generate three datasets of fixed-length string keys of size 80, 200, and 1440 that conform to either a Uniform or Normal distribution. Uniform keys are concatenations of uniformly generated key bytes up to the specified key length, while Normal keys are normally distributed around the middle of the key space with standard deviation $\sigma = 0.01 \cdot 2^{64}$. Specifically, the mean key is defined to be the string with a most significant byte value of 128 followed by null bytes up to the key length. We also generated Uniform, Correlated, and Split synthetic string workloads with RMAX 2^{30} and CORRDEGREE 2^{29}. The in-memory experiments were run with 10M keys, 1M queries, and 20K sample queries. For RocksDB experiments, we use an internet domain dataset comprising ~31M crawled .org domains [44]. The domains are 5 to 253 bytes long with a median length of 21 bytes and follow a log-normal distribution ($R^2 = 0.98$). The initial database was populated with 20M domain keys and 512 byte random values, resulting in a 3 level (~6GB compressed) LSM tree with ~30 SST files. Another 10M random domains were used to generate queries with RMAX 2^{30}, as with our other Real workloads. To control the distribution of RMAX, we pad the dataset with null bytes to the max key length. Note that this does not affect the performance of SuRF as it only considers the keys up to their unique prefixes which will be unchanged by the padding. For Proteus, the padding would ideally be done on a per-SST file basis to avoid unnecessary padding and modeling of designs with longer prefix lengths. These costs are incurred at construction time and not measured in the read-only experiment.
FPR is Unaffected by Key Length: In Figure 9a-d, we present in-memory results for 1440-bit keys. As with integers, Proteus outperforms SuRF across distributions and filter sizes. Experiments with other key lengths show the same performance patterns as only the range size and key-query proximity matter.

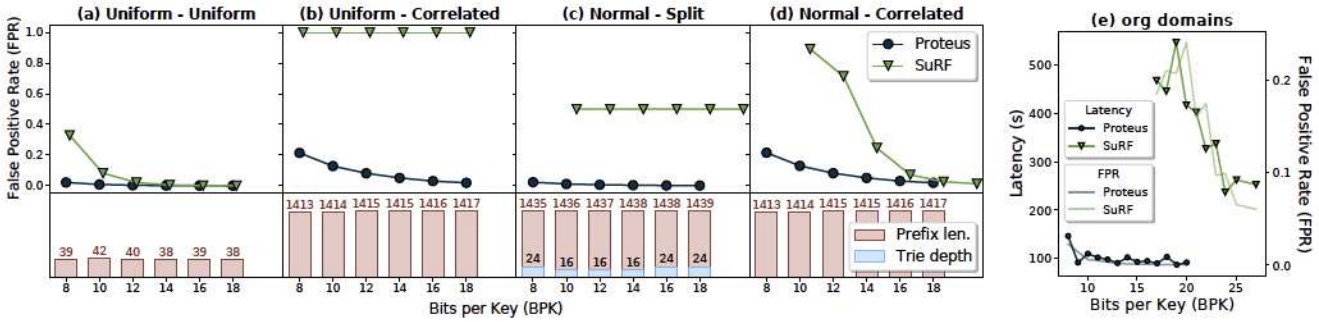


Figure 9: Proteus achieves lower FPR than SuRF on synthetic strings (a-d) and lower RocksDB latency on real-world strings (e).

Modeling Time Increases with Key Length: While the modeling accuracy is preserved across key length, the number of possible designs is $O(k^2)$ with respect to the maximum key length. Longer keys can therefore result in a significant increase in the time required to model the designs. In our in-memory benchmarks, the time to model all possible designs with 1440-bit keys ranged from 2.82 to 149 seconds. The worst case is too expensive for most real world applications; however, we can achieve an order of magnitude speedup by using a coarser search as the difference in performance for similar designs is often quite small. The results for Proteus shown in Figure 9 were obtained by only modeling 128 uniformly spaced Bloom filter prefix lengths for all feasible trie depths. The modeling times with this optimization ranged from 0.86 to 14.3 seconds while still achieving similar performance.

The structure of the modeling also lends itself very well to parallel computation. As shown in Section 4.3, the dominant cost of the sampling is extracting the information needed to model each sample query relative to each design. This can be done independently on both a per-query and per-design basis. Since modern database services are hosted on elastic cloud architectures, occasional increases in CPU usage can easily be amortized at a low cost compared to the benefits of a more performant filter.

Proteus Maintains Strong Performance on Strings: Figure 9 shows the results of our real-world string benchmark in RocksDB. Proteus outperforms SuRF in both end-to-end latency and FPR by an even larger margin using the aforementioned coarse-grained modeling. We see the impact of design tradeoffs amplified for longer keys —SuRF's rigid design requires a large minimum memory budget and limits the effectiveness of additional filter memory. Conversely, Proteus's flexible design allows it to distribute memory between its design elements for more efficient memory use. As shown in Figure 9, SuRF requires at least 16 BPK while Proteus can achieve significant performance gains with as little as 8 BPK.

8 RELATED WORK

Our investigation is part of a broader initiative in the systems community to design contextually-customized data structures.

Adaptive Range Filter: The Adaptive Range Filter (ARF) [1] adapts its binary trie structure in response to queries, extending branches to compensate for false positives and retracting them to maintain its memory footprint. However, ARF's encoding strategy limits its memory efficiency and requires significant time and memory to pre-train [48]. Similar adaptive techniques have been applied to AMQs to deal with adversarial workloads [7, 40].

Stacked Filters: Stacked Filters [15] also use modeling to incorporate workload specific information in their design. However, their model is designed for point rather than range queries.

Data Calculator: Similar to PRFs, the Data Calculator [26] breaks down a complex design space into its fundamental design primitives and models the behavior of designs to determine the optimal design for a given workload. However, the Data Calculator focuses on the design space of key-value structures rather than range filters and merely synthesizes the optimal design rather than instantiating it.

Learned Structures: Several iterations of learned Bloom filters use learned models to leverage patterns in the data for better performance [14, 39, 45]. These filters are designed for single item queries and perform poorly on range queries for the same reasons as other AMQs. This method of fitting to the use case also requires the presence of patterns in the data that are amenable to the model being used. Similar techniques have been applied to indexing structures in databases to speed up searches [32]. Indexes can also be used to answer range queries, but they are larger, general purpose structures which typically require more I/O to answer a query.

9 SUMMARY AND OPPORTUNITIES

This paper introduces Proteus, a self-designing range filter that achieves robust performance across a large variety of workloads. The core idea is that (1) Proteus unifies the design spaces of state-of-the-art range filters to cover a wider range of workloads and (2) is able to instantiate workload-optimal designs. Analysing cutting-edge range filtering techniques through the lens of the CPFPR model reveals adversarial workloads which no current design can handle practically, suggesting the need for further expansion of the range filter design space. Other promising directions include extending the CPFPR model to support higher order optimization strategies by incorporating metrics such as query latency as well as exploring non-uniform memory allocation strategies in RocksDB.

ACKNOWLEDGMENTS

This work is supported in part by NSF grants CCF-2101140, CNS-2107078, CCF-1563710, and DMS-2023528, and by a gift to the Center for Research on Computation and Society at Harvard University. Additional funding was provided by USA Department of Energy project DE-SC0020200. We would also like to thank our reviewers for their helpful and constructive feedback.

REFERENCES

[1] Karolina Alexiou, Donald Kossmann, and Paul Larson. 2013. Adaptive Range Filters for Cold Data: Avoiding Trips to Siberia. In *Proceedings of the VLDB Endowment, Vol. 6, No. 14.* https://www.microsoft.com/en-us/research/publication/adaptive-range-filters-for-cold-data-avoiding-trips-to-siberia/

[2] Sattam Alsubaiee, Yasser Altowim, Hotham Altwaijry, Alexander Behm, Vinayak Borkar, Yingyi Bu, Michael Carey, Inci Cetindil, Madhusudan Cheelangi, Khurram Faraaz, et al. 2014. AsterixDB: A scalable, open source BDMS. *arXiv preprint arXiv:1407.0454* (2014).

[3] Sattam Alsubaiee, Michael J. Carey, and Chen Li. 2015. LSM-Based Storage and Indexing: An Old Idea with Timely Benefits. In *Second International ACM Workshop on Managing and Mining Enriched Geo-Spatial Data* (Melbourne, VIC, Australia) *(GeoRich '15)*. Association for Computing Machinery, New York, NY, USA, 1–6. https://doi.org/10.1145/2786006.2786007

[4] Sabrina Amrouche, Laurent Basara, Paolo Calafiura, Dmitry Emeliyanov, Victor Estrade, Steven Farrell, Cécile Germain, Vladimir Vava Gligorov, Tobias Golling, Sergey Gorbunov, Heather Gray, Isabelle Guyon, Mikhail Hushchyn, Vincenzo Innocente, Moritz Kiehn, Marcel Kunze, Edward Moyse, David Rousseau, Andreas Salzburger, Andrey Ustyuzhanin, and Jean-Roch Vlimant. 2021. The Tracking Machine Learning challenge : Throughput phase. arXiv:2105.01160 [cs.LG]

[5] Austin Appleby. 2008. . https://sites.google.com/site/murmurhash/

[6] Diego Arroyuelo, Rodrigo Cánovas, Gonzalo Navarro, and Kunihiko Sadakane. 2010. Succinct Trees in Practice. In *Proceedings of the Meeting on Algorithm Engineering & Expermiments* (Austin, Texas) *(ALENEX '10)*. Society for Industrial and Applied Mathematics, USA, 84–97.

[7] Michael A. Bender, Martin Farach-Colton, Mayank Goswami, Rob Johnson, Samuel McCauley, and Shikha Singh. 2018. Bloom Filters, Adaptivity, and the Dictionary Problem. In *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*. 182–193. https://doi.org/10.1109/FOCS.2018.00026

[8] Michael A. Bender, Martin Farach-Colton, Rob Johnson, Russell Kraner, Bradley C. Kuszmaul, Dzejla Medjedovic, Pablo Montes, Pradeep Shetty, Richard P. Spillane, and Erez Zadok. 2012. Don't Thrash: How to Cache Your Hash on Flash. *Proc. VLDB Endow.* 5, 11 (July 2012), 1627–1637. https://doi.org/10.14778/2350229.2350275

[9] David Benoit, Erik D. Demaine, J. Ian Munro, Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. 2005. Representing Trees of Higher Degree. *Algorithmica* 43, 4 (Dec. 2005), 275–292.

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

[11] Flavio Bonomi, Michael Mitzenmacher, Rina Panigrahy, Sushil Singh, and George Varghese. 2006. An Improved Construction for Counting Bloom Filters *(ESA'06)*. Springer-Verlag, Berlin, Heidelberg, 684–695. https://doi.org/10.1007/11841036_61

[12] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H.C. Du. 2020. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. USENIX Association, Santa Clara, CA, 209–223. https://www.usenix.org/conference/fast20/presentation/cao-zhichao

[13] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. 2008. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)* 26, 2 (2008), 1–26.

[14] Zhenwei Dai and Anshumali Shrivastava. 2019. Adaptive Learned Bloom Filter (Ada-BF): Efficient Utilization of the Classifier. *CoRR* abs/1910.09131 (2019). arXiv:1910.09131 http://arxiv.org/abs/1910.09131

[15] Kyle Deeds, Brian Hentschel, and Stratos Idreos. 2021. Stacked Filters: Learning to Filter by Structure. *Proceedings of the VLDB Endowment* 14, 4 (2021), 600 – 612.

[16] Dgraph. 2017. *Fast Key-value DB in Go.* https://github.com/dgraph-io/badger

[17] S. Dharmapurikar, P. Krishnamurthy, and D.E. Taylor. 2006. Longest prefix matching using bloom filters. *IEEE/ACM Transactions on Networking* 14, 2 (2006), 397–409. https://doi.org/10.1109/TNET.2006.872576

[18] Peter C. Dillinger and Stefan Walzer. 2021. Ribbon filter: practically smaller than Bloom and Xor. *CoRR* abs/2103.02515 (2021). arXiv:2103.02515 https://arxiv.org/abs/2103.02515

[19] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. 2021. Evolution of Development Priorities in Key-value Stores Serving Large-scale Applications: The {RocksDB} Experience. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. 33–49.

[20] Bin Fan, Dave G. Andersen, Michael Kaminsky, and Michael D. Mitzenmacher. 2014. Cuckoo Filter: Practically Better Than Bloom. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies* (Sydney, Australia) *(CoNEXT '14)*. Association for Computing Machinery, New York, NY, USA, 75–88. https://doi.org/10.1145/2674005.2674994

[21] Edward Fredkin. 1960. Trie Memory. *Commun. ACM* 3, 9 (Sept. 1960), 490–499. https://doi.org/10.1145/367390.367400

[22] Wei Ge, Xianxian Li, Chunfeng Yuan, and Yihua Huang. 2019. Correlation-Aware Partitioning for Skewed Range Query Optimization. *World Wide Web* 22, 1 (Jan. 2019), 125–151. https://doi.org/10.1007/s11280-018-0547-4

[23] Mayank Goswami, Allan Grønlund, Kasper Green Larsen, and Rasmus Pagh. [n.d.]. *Approximate Range Emptiness in Constant Time and Optimal Space.* 769–775. https://doi.org/10.1137/1.9781611973730.52 arXiv:https://epubs.siam.org/doi/pdf/10.1137/1.9781611973730.52

[24] Thomas Mueller Graf and Daniel Lemire. 2019. Xor Filters: Faster and Smaller Than Bloom and Cuckoo Filters. *CoRR* abs/1912.08258 (2019). arXiv:1912.08258 http://arxiv.org/abs/1912.08258

[25] Stratos Idreos and Mark Callaghan. 2020. Key-value storage engines. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data.* 2667–2672.

[26] Stratos Idreos, Kostas Zoumpatianos, Brian Hentschel, Michael S. Kester, and Demi Guo. 2018. The Data Calculator: Data Structure Design and Cost Synthesis from First Principles and Learned Cost Models. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 535–550. https://doi.org/10.1145/3183713.3199671

[27] Facebook Inc. 2012. *RocksDB.* https://github.com/facebook/rocksdb

[28] Facebook Inc. 2020. *Compression.* https://github.com/facebook/rocksdb/wiki/Compression

[29] G. Jacobson. 1989. Space-efficient static trees and graphs. In *30th Annual Symposium on Foundations of Computer Science.* 549–554. https://doi.org/10.1109/SFCS.1989.63533

[30] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. 2019. SOSD: A Benchmark for Learned Indexes. *NeurIPS Workshop on Machine Learning for Systems* (2019).

[31] Haridimos Kondylakis, Niv Dayan, Kostas Zoumpatianos, and Themis Palpanas. 2020. Coconut Palm: Static and Streaming Data Series Exploration Now in your Palm. arXiv:2006.13079 [cs.DB]

[32] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 489–504. https://doi.org/10.1145/3183713.3196909

[33] Aapo Kyrola and Carlos Guestrin. 2014. GraphChi-DB: Simple Design for a Scalable Graph Database System – on Just a PC. arXiv:1403.0701 [cs.DB]

[34] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review* 44, 2 (2010), 35–40.

[35] Daniel Lemire and Owen Kaser. 2016. Faster 64-bit universal hashing using carry-less multiplications. *Journal of Cryptographic Engineering* 6, 3 (2016), 171–185.

[36] Siqiang Luo, Subarna Chatterjee, Rafael Ketsetsidis, Niv Dayan, Wilson Qin, and Stratos Idreos. 2020. 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* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 2071–2086. https://doi.org/10.1145/3318464.3389731

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

[38] Merriam-Webster. [n.d.]. Protean. In *Merriam-Webster.com dictionary.* https://www.merriam-webster.com/dictionary/Protean

[39] Michael Mitzenmacher. 2018. A Model for Learned Bloom Filters and Optimizing by Sandwiching. In *Advances in Neural Information Processing Systems*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.), Vol. 31. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2018/file/0f49c89d1e7298bb9930789c8ed59d48-Paper.pdf

[40] Michael Mitzenmacher, Salvatore Pontarelli, and Pedro Reviriego. 2017. Adaptive Cuckoo Filters. arXiv:1704.06818 [cs.DS]

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

[42] Andrew Pavlo, Carlo Curino, and Stanley Zdonik. 2012. Skew-Aware Automatic Database Partitioning in Shared-Nothing, Parallel OLTP Systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (Scottsdale, Arizona, USA) *(SIGMOD '12)*. Association for Computing Machinery, New York, NY, USA, 61–72. https://doi.org/10.1145/2213836.2213844

[43] Swaminathan Sivasubramanian. 2012. Amazon dynamoDB: a seamlessly scalable non-relational database service. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data.* 729–730.

[44] Bohdan Turkynewych. 2022. *Domains Project.* Retrieved March 20, 2022 from https://domainsproject.org/

[45] Kapil Vaidya, Eric Knorr, Tim Kraska, and Michael Mitzenmacher. 2020. Partitioned Learned Bloom Filter. *CoRR* abs/2006.03176 (2020). arXiv:2006.03176 https://arxiv.org/abs/2006.03176

[46] Cheng Xu, Ce Zhang, and Jianliang Xu. 2019. VChain: Enabling Verifiable Boolean Range Queries over Blockchain Databases. In *Proceedings of the 2019 International*

Conference on Management of Data (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 141–158. https://doi.org/10.1145/3299869.3300083

[47] Eleni Tzirita Zacharatou, Darius Šidlauskas, Farhan Tauheed, Thomas Heinis, and Anastasia Ailamaki. 2019. Efficient Bundled Spatial Range Queries. In *Proceedings of the 27th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems* (Chicago, IL, USA) *(SIGSPATIAL '19)*. Association for Computing Machinery, New York, NY, USA, 139–148. https://doi.org/10.1145/3347146.3359077

[48] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G. Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. 2018. SuRF: Practical Range Query Filtering with Fast Succinct Tries *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 323–336. https://doi.org/10.1145/3183713.3196931