Rosetta: A Robust Space-Time Optimized Range Filter for Key-Value Stores

Siqiang Luo, Subarna Chatterjee, Rafael Ketsetsidis
Niv Dayan, Wilson Qin, Stratos Idreos
Harvard University

ABSTRACT

We introduce Rosetta, a probabilistic range filter designed specifically for LSM-tree based key-value stores. The core intuition is that we can sacrifice filter probe time because it is not visible in end-to-end key-value store performance, which in turn allows us to significantly reduce the filter false positive rate for every level of the tree.

Rosetta indexes all binary prefixes of a key using a hierarchically arranged set of Bloom filters. It then converts each range query into multiple probes, one for each non-overlapping binary prefix. Rosetta has the ability to track workload patterns and adopt a beneficial tuning for each individual LSM-tree run by adjusting the number of Bloom filters it uses and how memory is spread among them to optimize the FPR/CPU cost balance.

We show how to integrate Rosetta in a full system, RocksDB, and we demonstrate that it brings as much as a 40x improvement compared to default RocksDB and 2-5x improvement compared to state-of-the-art range filters in a variety of workloads and across different levels of the memory hierarchy (memory, SSD, hard disk). We also show that, unlike state-of-the-art filters, Rosetta brings a net benefit in RocksDB's overall performance, i.e., it improves range queries without losing any performance for point queries.

ACM Reference Format:
Siqiang Luo, Subarna Chatterjee, Rafael Ketsetsidis and Niv Dayan, Wilson Qin, 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 (SIGMOD'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3318464.3389731

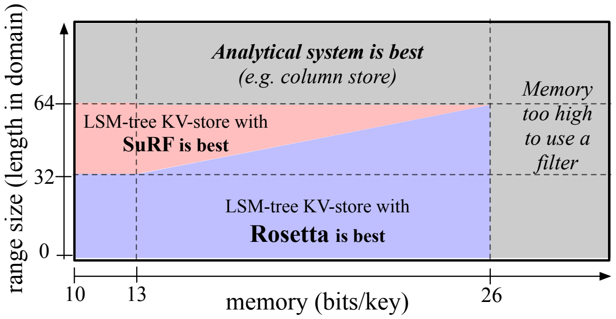


Figure 1: Rosetta brings new performance properties improving on and complementing existing designs.

1 INTRODUCTION

Log Structured Merge (LSM)-Based Key-value Stores. LSM-based key-value stores in industry [2, 4, 5, 15, 30, 36, 43, 53, 55, 71, 73] increasingly serve as the backbone of applications across the areas of social media [6, 12], stream and log processing [14, 16], file structures [47, 65], flash memory firmware [26, 69], and databases for geo-spatial coordinates [3], time-series [50, 51] and graphs [32, 52]. The core data structure, LSM-tree [58] allows systems to target write-optimization while also providing very good point read performance when it is coupled with Bloom filters. Numerous proposals for variations on LSM-tree seek to strike different balances for the intrinsic read-write tradeoff [7], demonstrated in a multitude of academic systems [8, 11, 24, 25, 27, 28, 42, 45–47, 54, 56, 61, 62, 64, 67, 72].

Filtering on LSM-trees. As LSM-based key-value stores need to support fast writes, the ingested data is inserted 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 [48], 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

SIGMOD'20, June 14–19, 2020, Portland, OR, USA

ACM ISBN 978-1-4503-6735-6/20/06…$15.00
https://doi.org/10.1145/3318464.3389731

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 [45].

Range Queries Are Hard. Range queries are increasingly important to modern applications, as social web application conversations [18], distributed key-value storage replication [63], statistics aggregation for time series workloads [49], and even SQL table accesses as tablename prefixed key requests [17, 35] are all use cases that derive richer functionality from building atop of key-value range queries. While LSM-based key-value stores support efficient writes and point queries, they suffer with range queries [25, 27, 28, 34, 57]. 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.

State of the Art. Fence pointers can rule out disk blocks with contents that fall outside the target key range. However, once the qualifying data blocks are narrowed down, the queried range within a data block may be empty – a concern especially for short range queries. To improve range queries, modern key-value stores utilize Prefix Bloom filters [33] that hash prefixes of a predefined length of each key within Bloom filters [36]. Those prefixes can be used to rule out large ranges of data that are empty. While this is a major step forward, a crucial restriction is that this works only for range queries that can be expressed as fixed-prefix queries.

The state of the art is Succinct Range Filter (SuRF) [74] that can filter arbitrary range queries by utilizing a compact trie-like data structure. SuRF encodes a trie of n nodes with maximum fanout of 256. The trie is culled at a certain prefix length. 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).

Problem 1: Short and Medium Range Queries are Significantly Sub-Optimal. Although trie-culling in SuRF is effective for long range queries, it is not very effective on short ranges as (i) short range queries have a high probability of being empty and (ii) the culled prefixes may not help as the keys differ mostly by the least significant bits. Similarly, fence pointers or Prefix Bloom filters cannot help with short range queries unless the data is extremely sparse. On the other hand, if a workload is dominated by (very) long range queries, then a write-optimized multi-level key-value store is far from optimal regardless of the filter used: instead, a scan-optimized columnar-like system is appropriate [1].

Problem 2: Lack of Support for Workloads With Key-Query Correlation or Skew. Unlike hash-based filters[1] such as Bloom and Cuckoo filters [10, 37], both Prefix Bloom filters and SuRF are dependent on the distribution of keys and queries, and as such, are affected by certain properties of the distribution. Specifically, a major source of false positives in these filters is when a queried key does not exist but shares the prefix with an existing one. Such workloads are common in applications that care about the local properties of data. For example, a common use case of an e-commerce application is to find the next order id corresponding to a given order id. This translates to querying the lexicographically smallest key that is greater than the current key. Other examples include time-series applications that often compare events occurring within a short time range for temporal analysis or geo-spatial applications that deal with points-of-interest data which are queried to search a local neighborhood. In fact, the RocksDB API [36] accesses keys through an iterator that steps through the keys in lexicographical order.

Similarly, existing filters do not take advantage of skew, for instance when key distributions are sparse and target ranges contain empty gaps in key range.

Overall Problem: Sub-Optimal Performance for Diverse Workloads and Point Queries. Overall, existing filters in key-value stores suffer in FPR performance due to variation of one or more of the following parameters – range sizes, key distributions, query distributions, and key types. Another critical problem is that neither indexing prefixes with Prefix Bloom filters nor using SuRF (even SuRF-Hash) can help (much) with point queries. In fact, point query performance degrades significantly with these filters. Thus, for workloads that contain both point queries and range queries, an LSM-tree based key-value store with such filters needs to either maintain a separate Bloom filter per run to index full keys or suffer a high false positive rate for point queries.

Our Intuition: We Can Afford to Give Up CPU. All state-of-the-art filters such as, Bloom filters, Quotient filters [9], and SuRF trade among the following to gain in end-to-end performance in terms of data access costs.

1. Memory cost for storing the filter
2. Probe cost, i.e., the time needed to query the filter
3. False positive rate, which reflects how many empty queries are not detected by the filter, thus leading to unnecessary storage I/O

These tradeoffs make sense because an evolving hardware trend is that contemporary speed of CPU processing facilitates fast in-memory computations in a matter of tens of

[1] SuRF-Hash uses hashes to improve point-queries, not for range filtering.

nanoseconds, whereas even the fastest SSD disks take tens of microseconds to fetch data. In this paper, we use an additional insight which we also demonstrate experimentally later on. When a range filter is properly integrated into a full system, the probe cost of the filter is, in fact, a small portion of the overall key-value store system CPU cost (less than 5%). This is because of additional CPU overhead emanating from range queries traversing iterators (requiring a series of fence pointer comparisons) for each relevant run, and resulting deserialization costs for maintaining the system block cache for both metadata (filters and fence pointers) and data. Thus, since we target specifically LSM-tree based key-value stores, we have even more leeway to optimize the range filter design with respect to both probe cost (CPU) we can afford and FPR we can achieve.

Rosetta. We introduce a new filter, termed Rosetta (A Robust Space-Time Optimized Range Filter for Key-Value Stores), which allows for efficient and robust range and point queries in workloads where state-of-the-art filters suffer. Rosetta is designed with the CPU/FPR tradeoff from the ground up and inherently sacrifices filter probe cost to improve on FPR. The core of the new design is in translating range queries into prefix queries, which can then be turned into point queries. Rosetta stores all prefixes for each key in a series of Bloom filters organized at different levels. For every key, each prefix is hashed into the Bloom filter belonging to the same level as that of the prefix length. For example, when the key 6 (which corresponds to 0110) is inserted to Rosetta, all possible prefixes $\varnothing$, 0, 01, 011, and 0110 are inserted to the first, second, third, fourth, and fifth Bloom filter respectively. A range query of size R (i.e., with the form $[A, A+R-1]$) is decomposed into at most $2\log_2(R)$ dyadic ranges (i.e., ranges of size 2^r that share the same binary prefix for some length r). In essence, this design builds a series of implicit Segment Trees on top of the Bloom filters. Due to the multiple probes required (for every prefix) Rosetta has a high probe cost. Rosetta also adaptively auto-tunes to sacrifice as much probe cost as possible in favor of FPR by monitoring workloads' patterns and deciding optimal bit allocation across Bloom filters (by flattening the implicit Segment Tree when possible).

Our **contributions** are as follows:

- We introduce a new range filter design for key-value stores which utilizes auxiliary CPU resources to achieve a low false positive rate. The core design is about storing all prefixes of keys in a series of Bloom filters organized in an implicit Segment Tree structure.
- We show that Rosetta approaches the optimal memory lower bound for any false positive rate within a constant factor.
- We show that Rosetta can be configured in versatile ways to balance the trade-off between filter probe cost, memory cost, and false positive rate. This is done by controlling the number of Bloom filters being used as well as the memory used at each Bloom filter.
- We show how to integrate Rosetta in a full system, RocksDB and the effect of several design factors that affect full system performance including the number of iterators per query, deserialization costs, seek cost, and SST file size.
- We demonstrate that compared to RocksDB (with fence pointers and Prefix Bloom filters), Rosetta brings an up to 40x improvement for short and medium range queries, while compared to RocksDB with SuRF, Rosetta improves by a factor of 2-5x. These improvements are observed across diverse workloads (uniform, correlated, skewed) and across different memory hierarchy layers (memory, SSD, hard disk).
- As components of end-to-end performance, we show the fundamental tradeoffs between filter construction latency, filter probe latency, and disk access latency for a given memory budget. We also relate these costs to CPU and I/O of the whole key-value store and demonstrate that by utilizing more CPU in accessing filters, Rosetta significantly improves the overall key-value store performance compared to state-of-the-art filters.
- We also demonstrate that Rosetta brings a net benefit in RocksDB by not hurting point query performance. Rosetta can process worst case point queries as well as point-query optimized RocksDB or better (while Prefix Bloom filters and SuRF incur a 60% and 40% overhead for point queries, respectively).
- We also show the holistic positioning of Rosetta compared to existing methods so applications can decide when to use which filter: Figure 1 shows this summary. Given the same memory budget, Rosetta is robust across workloads and beneficial with short range queries. SuRF is beneficial for medium range queries while once query range gets bigger a key-value store is not beneficial anymore.

2 ROSETTA

Rosetta in an LSM-tree KV-Store. A Rosetta instance is created for every immutable run of an LSM-tree. For every run of the tree, a point or range query, first probes the corresponding Rosetta filter for this run, and only tries to access the run on disk if Rosetta returns a positive. As with Bloom filters and fence pointers in LSM-trees, every time there is a merge operation, the Rosetta filters of the source runs are dropped and a new Rosetta instance is created for the new run. We discuss in detail the integration of Rosetta in a full key-value store in Section 4. The terms used in the following discussion can be referred to Table 1.

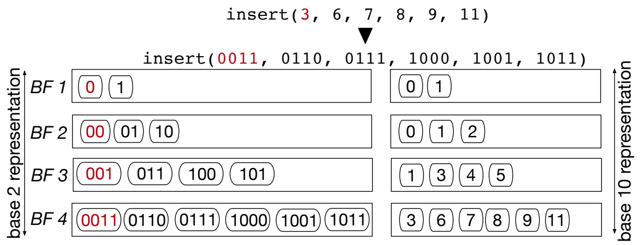


Figure 2: Rosetta indexes all prefixes of each key.

Term	Definition
n	Number of keys
R	Maximum range query size
M	Memory budget
$\mathcal{B}_i$	Bloom filter corresponding to length-i prefix
ϵ_i	False positive rate of $\mathcal{B}_i$
M_i	Memory allocated for $\mathcal{B}_i$
ϵ	Resulting false positive rate for a prefix query after considering recursive probes
ϕ	General reference of the FPR of a Bloom filter at Rosetta

Table 1: Terms used.

2.1 Constructing Rosetta for LSM-tree Runs

For every run in an LSM-tree, a Rosetta exists which contains information about all the entries within the run. A Rosetta is created either when the in-memory buffer is flushed to disk or when a new run is created because two or more runs of older data are merged into a new one. Thus, insertion in Rosetta is done once all the keys which are going to be indexed by this Rosetta instance are known. All keys within the run are broken down into variable-length binary prefixes, thereby generating $\mathcal{L}$ distinct prefixes for each key of length $\mathcal{L}$ bits. Each i-bit prefix is inserted into the i^{th} Bloom filter of Rosetta. The insertion procedure is described in Algorithm 1, and an example is shown in Figure 2. Let us take inserting 3 (corresponding to 0011) as an example. As shown in Figure 2 (the red numbers), we insert prefixes 0, 00, 001 and 0011 into $\mathcal{B}_1$, $\mathcal{B}_2$, $\mathcal{B}_3$ and $\mathcal{B}_4$ respectively. Here insertion means that we index every prefix in the Bloom filter by hashing it and setting the corresponding bits in the Bloom filter. The prefix itself is not stored. Figure 3 shows the final state of Rosetta after inserting all keys in a run (keys 3, 6, 7, 8, 9, 11 in our example). We do not need to take care of dynamic updates to the data set (e.g., a new key arriving with a bigger length) since each separate instance of Rosetta is created for an immutable run at a time. In addition, given that every run of the LSM-tree might have a different set of keys, different Rosetta instances across the tree can have a different number of Bloom filters.

Algorithm 1: Insert

```
Function Insert($K$):
    /* $K$ is the key to be inserted.          */
    $\mathcal{L} \leftarrow$ number of bits in a key
    $\mathcal{B} \leftarrow \mathcal{L}$ Bloom filters
    for $i \in \{\mathcal{L}, \ldots, 1\}$ do
        $\mathcal{B}_i$.Insert($K$)
        $K \leftarrow \lfloor K/2 \rfloor$
```

Implicit Segment Tree in Rosetta. Rosetta effectively forms an implicit Segment Tree [68] (as shown in Figure 3) which is used to translate range queries into prefix queries. Segment Trees are full binary trees and thus perfectly described by their size, so there is no need to store information about the position of the left and right children of each node.

2.2 Range and Point Queries with Rosetta

2.2.1 Range Lookups. We now illustrate how a range lookup is executed in Rosetta. For every run in an LSM-tree, we first probe the respective Rosetta instance. Algorithm 2 shows the steps in detail and Figure 3 presents an example.

First, the target key range is decomposed into dyadic intervals. For each dyadic interval, there is a sub-tree within Rosetta where all keys within that interval have been indexed. For every dyadic interval, we first search for the existence of the prefix that covers all keys within that interval. This marks the root of the sub-tree for that interval. In the example of Figure 3, the range query `range(8, 12)` breaks down into two dyadic ranges: [8, 11] (the red shaded area) and [12, 12] (the green shaded area). We first probe the Bloom filter residing at the root level of the sub-tree. The Segment Tree is implicit, so the length of the prefix determines which Bloom filter to probe. For example, the prefix 10∗ contains every key in [8, 11] and nothing more (i.e., 1000, 1001, 1010, 1011), so for the range [8, 11] we will probe 10 from the second Bloom filter. If this probe returns a negative, we continue in the same way for the next dyadic range. If the probe from every dyadic range returns negative, the range of the query is empty. As such, in the example of Figure 3, we will probe the second Bloom filter for the existence of 10 and, if that returns negative, we will probe the fourth Bloom filter for the existence of 1100. If both probes return negative, the range is empty (when inserting a key we insert all of its prefixes to the corresponding Bloom filter, so for example if there was a key in the range [8, 11], we would have inserted 10 to the second Bloom filter).

If at least one prefix in the target range returns positive, then Rosetta initiates the process of *doubting*. We probe

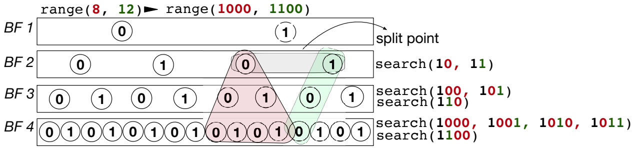


Figure 3: Rosetta utilizes recursive prefix probing.

Bloom filters residing at deeper levels of the sub-tree and in turn, determine the existence of all possible prefixes within the particular dyadic interval. In the example of Figure 3, when querying the range [8, 11], if the probe for prefix $10*$ is positive, we further probe for $100*$ and $101*$ the third Bloom filter. If both of these probes are negative, we can mark the sub-range as empty, since any key with prefix $10*$ will either have the prefix $100*$ or the prefix $101*$.

In general, all Bloom filters at the subsequent levels of Rosetta are recursively probed to determine the existence of intermediate prefixes by traversing the left and right subtrees in a pre-order manner. The left sub-tree corresponds to the current prefix with 0 appended, while the right sub-tree corresponds to the current prefix with 1 appended. Recursing within a sub-tree terminates if either (a) a Bloom filter probe corresponding to a left-leaf in the sub-tree returns positive which marks the whole subrange as positive, or (b) case (a) does not happen while any Bloom filter probe corresponding to a right node returns negative in which case the whole sub-range is marked as negative.

In the event that the range is found to be a positive, there is still a chance for Rosetta to reduce I/O by "tightening" the effective range to touch on storage. That is, if Rosetta finds that intervals at the edge of the requested range are empty, it will proceed to do I/O only for the narrower, effective key range which is determined as positive.

2.2.2 Point Lookups. The last level of Rosetta indexes the whole prefix of every key in the LSM-tree run that it represents. This is equivalent to indexing the whole key in a Bloom filter which is how traditional Bloom filters are used in the default design of LSM-trees to protect point queries from performing unnecessary disk access. In this way, Rosetta can answer point queries in exactly the same way as point-query optimized LSM-trees: For every run of the LSM-tree, a point query only checks the last level of Rosetta for this run, and only tries to access disk if Rosetta returns a positive.

2.2.3 Probe Cost vs FPR. The basic Rosetta design described above intuitively sacrifices CPU cost during probe time to improve on FPR. This is because of the choice to index all prefixes of keys and subsequently recursively probe those prefixes at query time. In the next two subsections, we show how to further optimize FPR through memory allocation and further CPU sacrifices during probe time.

Algorithm 2: Range Query

```
Function RangeQuery($Low, High, P = 0, l = 1$):
    /* $[Low, High]$: the query range;    */
    /* $P$: the prefix.                   */
    if $P > High$ or $P + 2^{\mathcal{L}-l+1} - 1 < Low$ then
        /* $P$ is not contained in the range    */
        return false
    if $P \geq Low$ and $P + 2^{\mathcal{L}-l+1} - 1 \leq High$ then
        /* $P$ is contained in the range    */
        return Doubt($P, l$)
    if RangeQuery($Low, High, P, l + 1$) then
        return true
    return RangeQuery($Low, High, P + 2^{\mathcal{L}-l}, l + 1$)
Function Doubt($P, l$):
    if $\neg\mathcal{B}_l$.PointQuery($P$) then
        return false
    if $l > \mathcal{L}$ then
        return true
    if Doubt($P, l + 1$) then
        return true
    return Doubt($P + 2^{\mathcal{L}-l}, l + 1$)
```

2.3 FPR Optimization Through Memory Allocation

A natural question arises: how much memory should we devote to each Bloom filter in a Rosetta instance for a given LSM-tree run? Alternatively, what is the false positive rate ϵ_i that we should choose for the filters in order to minimize the false positive rate for a given memory budget? Equally distributing the memory budget among all Bloom filters overlooks the different utilities of the Bloom filters at different Rosetta levels. In this section, we discuss this design issue.

A First-Cut Solution: FPR Equlibrium Across Nodes. Our first-cut solution is to achieve an equilibrium, such that an equal FPR ϵ is achieved for any segment tree node u by considering the filters both at u and u's descendant nodes. As we will explain later in Section 3, this strategy gives very good theoretical interpretation in terms of memory usage– it achieves a 1.44-approximation optimal memory usage. The detailed calculation for the FPR ϕ of the filter at u's level is as follows. By the aforementioned objective equilibrium, we suppose that a node u's children all have false positive rate ϵ, after taking the filters of their respective descendant nodes into account. If we consider the filters of u and its descendants all together, an ultimate false positive is achieved by either 1) the filter at u returns a positive, and u's left sub-tree returns a positive. This case leads to a positive rate $\phi \cdot \epsilon$; or 2) the filter at u returns a positive, u's left sub-tree returns a negative while u' right sub-tree returns a positive. This case gives us a positive rate $\phi \cdot (1 - \epsilon) \cdot \epsilon$. Putting these two cases together gives the following equation.

$$\phi \cdot \epsilon + \phi \cdot (1 - \epsilon) \cdot \epsilon = \epsilon \Rightarrow \phi \cdot (2 - \epsilon) = 1 \Rightarrow \quad \phi = \frac{1}{2 - \epsilon}$$

This means that we may set each non-terminal node's false positive rate to $1/(2 - \epsilon)$.

Optimized Allocation of Memory Across Levels. Based on the aforementioned idea, there are still many Bloom filters that are treated equally (except for the last level). However, we find that in general, the Bloom filters at shorter prefix Rosetta levels are probed less than deeper levels. Intuitively, we should set lower FPRs for Bloom filters that are probed more frequently. To formalize this idea, we analyze the frequency of a segment tree node u being accessed (i.e., the Bloom filter at node u is probed), if we issue every query of size R once. Our analytical results are as follows.

Let level-r of Rosetta be the level which is r level higher than the leaf level. Then, the frequency of a level-r node being accessed, $g(r)$, is expressed as,

$$g(r) = \sum_{0 \le c \le \lfloor \log R \rfloor - r} g(r + c, R - 1) \tag{1}$$

$$\text{where, } g(x, R-1) = \begin{cases} 1, & x \in [0, \lfloor \log R \rfloor) \\ \frac{R - 2^x + 1}{2^x}, & x = \lfloor \log R \rfloor \\ 0, & x > \lfloor \log R \rfloor \end{cases} \tag{2}$$

To incorporate the idea of achieving lower FPRs for less frequently accessed filters, we aim to minimize the objective of $\sum_{0 \le r \le \lfloor \log R \rfloor} g(r) \cdot \epsilon_r$ (where ϵ_r is the FPR of the level-r Bloom filter of Rosetta), subject to the overall memory budget M. Let M_r be the number of bits assigned to the level-r Bloom filter. Then the following expression minimizes the aforementioned function.

$$M_r = -\frac{n}{(\ln 2)^2} \ln \frac{C}{g(r)} \tag{3}$$

$$\text{where, } C = \left(\prod_{0 \le r \le \lfloor \log R \rfloor} g(r) \right)^{\frac{1}{\lfloor \log R \rfloor}} \cdot \left(e^{-\frac{M}{n}} \right)^{\frac{(\ln 2)^2}{\lfloor \log R \rfloor}} \tag{4}$$

The derivation of the Equation 3 is as follows. First, we transform the objective function [2]:

$$\sum_{0 \le r \le \lfloor \log R \rfloor} g(r) \cdot \epsilon_r = \sum_{0 \le r \le \lfloor \log R \rfloor} g(r) \cdot e^{-\frac{M_r}{n} \cdot (\ln 2)^2} \tag{5}$$

Then, by the AM-GM inequality [3] and let $h = 1 + \lfloor \log R \rfloor$, we have,

$$\sum_{0 \le r \le \lfloor \log R \rfloor} g(r) e^{-\frac{M_r}{n} \cdot (\ln 2)^2} \ge h \left(\prod_{0 \le r \le h-1} g(r) e^{-\frac{M_r}{n} \cdot (\ln 2)^2} \right)^{\frac{1}{h}}$$

$$\Rightarrow \sum_{0 \le r \le \lfloor \log R \rfloor} g(r) e^{-\frac{M_r}{n} \cdot (\ln 2)^2} \ge h \left(\prod_{0 \le r \le h-1} g(r) \right)^{\frac{1}{h}} \left(e^{-\frac{M}{n} \cdot (\ln 2)^2} \right)^{\frac{1}{h}}$$

$$\Rightarrow \sum_{0 \le r \le \lfloor \log R \rfloor} g(r) \cdot e^{-\frac{M_r}{n} \cdot (\ln 2)^2} \ge (1 + \lfloor \log R \rfloor) \cdot$$

$$\left(\prod_{0 \le r \le \lfloor \log R \rfloor} g(r) \right)^{\frac{1}{\lfloor \log R \rfloor}} \cdot \left(e^{-\frac{M}{n}} \right)^{\frac{(\ln 2)^2}{\lfloor \log R \rfloor}} \tag{6}$$

Given parameters R, r, M, n, the last value in Equation 6 is a constant. Hence, the minimum achieves when the equality condition of the arithmetic-mean-inequality holds, i.e.,

$$g(r) \cdot e^{-\frac{M_r}{n} \cdot (\ln 2)^2} = C \tag{7}$$

where, $C = \left(\prod_{0 \le r}^{\lfloor \log R \rfloor} g(r) \right)^{\frac{1}{\lfloor \log R \rfloor}} \cdot \left(e^{-\frac{M}{n}} \right)^{\frac{(\ln 2)^2}{\lfloor \log R \rfloor}}$.

Transforming Equation 7 gives the optimal memory bits allocation as shown in Equation 3. When M_r solved gives a negative value, we reset $M_r = 0$ and rescale the other $M_{r'}$ for $r' \neq r$ to maintain the sum of memory bits to be M.

2.4 FPR Optimization Through Further Probe Time Sacrifice

As we will show in Section 5, filter probe time is a tiny portion of overall query time in a key-value store. This means that we can explore opportunities to further sacrifice filter probe time if that might help us to further improve on FPR. In this section, we discuss such an optimization.

The core intuition of the optimization in this section is that the multiple levels of Rosetta require valuable bits. Effectively, all levels but the last one in each Rosetta instance help primarily with probe time as their utility is to prune the parts of the range we are going to query for at the last level of Rosetta.

The fewer such ranges the less the probe cost of the filter. At the same time, though, these bits at the top levels of Rosetta could be used to store prefixes with fewer conflicts at the bottom level, improving FPR. This gives a balance between FPR and probe cost. We present two new strategies based on this observation.

(i) *Single-level filter for small ranges.* This is an extreme design option of using only a single level Rosetta. This means

[2] For ease of analysis, we assume each Bloom filter at level in $[0, \lfloor \log R \rfloor]$ has n unique keys.

[3] $x_1 + x_2 + \ldots + x_h \ge \frac{1}{h} \cdot \prod_{1 \le i \le h} {x_i}^{\frac{1}{h}}$, For $x_i \ge 0$, the equality holds when all x_i are equal.

that all of Rosetta is stored in a single Bloom filter maximizing the utility of the available memory budget but increasing probe time as all ranges need to be queried. Thus the probe time is linear to the range size.

(ii) *Variable-level filter for large ranges.* An alternative approach to the extreme single-level design is to selectively push more bits at lower levels (e.g., more aggressively pushing bits at lower levels to the point where a level might be empty). The amount of bits chosen determines the CPU/FPR balance. We achieve this by associating a Bloom filter with a weight as the sum of its probe frequency and all the probe-frequencies of its above Bloom filters. The new weight can be expressed as follows.

$$w(\mathcal{B}_i) = \sum_{i \le r \le \lfloor \log R \rfloor} g(i)$$

Then we apply Equation 3 to compute bits-allocation based on the new weights (i.e., by replacing $g(r)$ with $w(\mathcal{B}_r)$). In this way, we shift more bits from upper levels to bottom levels. Also, when M_r solved by Equation 3 gives a negative value, we set $M_r = 0$ and update $w(\mathcal{B}_i)$ by adding up its current weight with all the weights of its above Bloom filters.

To evaluate the FPR performance and probe time of the original mechanism (by Equation 3), single-level filter, and new variable-level filter, we perform a test on 10 million keys with a Rosetta with 10 bits-per-key. Figure 4 varies a range size from 2 to 512 (x-axis). Each bar represents FPR (right figure) or probe cost (left figure). While the single-level filter has the best FPR, its probe cost is significantly higher than the other two methods starting from a range size of 32. This range size is also the turning point where the FPR performance of the new variable-level filter starts to overtake the original mechanism, while achieving a more reasonable probe cost compared with the single-level filter. This observation motivates a hybrid mechanism of both single-level and variable-level filters depending on the workload: for workloads where small ranges (i.e., at most size 16) are dominating, we employ the single-level filter, whereas in other cases, we build Rosetta with the mediated filter.

Finally, note that our solution to distribute the bits across the different Rosetta levels based on actual runtime filter use, requires workload knowledge. This is knowledge attainable by extending the native statistics already captured in a key-value store. Given that a Rosetta instance is created for every run, we keep counters and histograms for query ranges, invoked Rosetta instances and the corresponding hit rates on the underlying runs. More specifically we record the query ranges seen and also record the utilization and performance of each Bloom filter in the invoked Rosetta instances. At compaction time, we reconcile these statistics and create the post-compaction Rosetta instances using this workload information to extract the weights and to decide on single vs variable levels for each run.

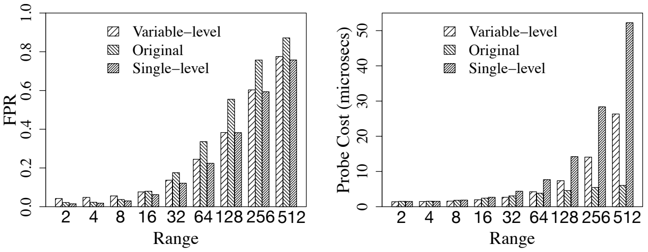


Figure 4: Depending on range size, a different bits-allocation mechanism can be used.

3 THEORETICAL INSIGHTS

We show that Rosetta is very close to be space optimal, and the query time is moderate.

3.1 Space Complexity is Near-optimal

It has been shown in [44] that in order to achieve a FPR of ϵ for any range query of size R, any data structure should use at least $n \log \left(\frac{R^{1-O(\epsilon)}}{\epsilon} \right) - O(n)$ of bits, where n is the number of keys.

For memory allocation in Rosetta, we have introduced the first-cut solution and the optimized approaches. The first-cut solution is relevant to the given FPR guarantee ϵ, while the spirit of the optimized approaches consider the utilities of each filter thereby being workload aware. As the space lower bound is given regarding ϵ, our discussion of the space complexity will focus on the first-cut solution, which is with regard to ϵ directly.

The first-cut solution of memory allocation gives an ϵ FPR for the last level but $\frac{1}{2-\epsilon}$ FPR for the other levels. Since each level in Rosetta may have at most n nodes, we get that the total memory required is:

$\log e \cdot (n \log(1/\epsilon) + n \log(u) \log(2 - \epsilon)) \approx 1.44 \cdot n \log (u/\epsilon)$, where u is the size of the key space. If we have a bound R on the maximum size of range queries, we may disregard some levels of the Rosetta, which gives us a memory usage of $1.44 \cdot n \log (R/\epsilon)$. This means Rosetta is very close to the aforementioned theoretical lower bound.

3.2 Computational Complexity is Low

Construction. Given that each run in an LSM-tree contains sorted keys, we can avoid inserting common prefixes of consecutive keys, which means that we only insert unique prefixes to the Bloom filters. Thus, the number of insertions to Bloom filters that we perform is equal to the size of a binary trie containing the keys, which is upper bounded by $n \cdot \mathcal{L}$. This complexity is very close to the lower bound, which barely reads every bit of the keys.

Querying. The *query* time complexity is proportional to the maximum number of prefixes that we break the range down into. In a Segment Tree that number is equal to $2\mathcal{L}$ [29]. Similarly, a range of size R will need up to $2\log_2(R)$ queries since it can break into at most that many prefixes. In the worst-case scenario, the number of probes needed for a range query is equal to the number of Segment Tree nodes under the sub-trees of every dyadic range, which is $O(R)$. However, computation will end if there are no positive probes that we can recurse over any longer. Our analysis shows that in expectation, the number of probes is typically much less than the worst case. We divide the analysis into multiple cases.

Uniform Bits-Allocation. Consider a Rosetta with each of its Bloom filters of FPR equal to $p = 0.5 + \theta, \theta \neq 0, |\theta| < 0.5$. We analytically show that the expected runtime of a query is less than $O(\frac{\log R}{\theta^2})$ for an empty range. For an empty range, the process of doubting ends once either no positives are remaining, or the level of recursion reaches the last level of Rosetta. Let E be the expected number of Bloom filter probes under the condition that the level of recursion never reaches the last level (i.e., level $\mathcal{L}$), and F the expected number of probes under the condition that it does. By linearity of expectation, we have that the total expected number of probes is equal to $E + F$. To upper bound $E + F$, we consider a Rosetta containing an infinite number of Bloom filters where each has an FPR equal to p. Querying with such an "infinite" Rosetta is at least as costly as querying with the original Rosetta. To see this, if a query never reaches level $\mathcal{L}$ or any probe in level $\mathcal{L}$ always returns a negative, then probe cost with either Rosetta will be the same. However, if a query reaches level $\mathcal{L}$ and a probe in this level returns a positive, then with the original Rosetta the query stops immediately by returning positive, whereas with the infinite Rosetta the probe can further recurse to deeper levels, thereby incurring more probe cost. Next, we analyze the probe cost within each dyadic range in the infinite Rosetta, and we reuse the term E defined previously for ease of presentation.

Let E_i be the expected number of Bloom filter probes, and has encountered exactly i positives, which means that by linearity of expectation, $E = \sum_{i=0}^{\infty} E_i$. Each positive probe either recurses into two negative probes (which then recurse no further), two positive probes (both of which recurse), or one positive (which recurses) and one negative (which does not). Therefore the probes that we perform form a binary tree whose leaves are the negative probes, and every other node is a positive probe. From this property, we have that there are $i + 1$ negatives, and as such $2i + 1$ probes in total. Then, we have that $E_i = P_i(2i+1)$, where P_i is the probability that the number of positive probes is equal to i. Let C_i be the ith Catalan number, which is equal to the number of unique binary trees with $2i + 1$ nodes [66]. We have that $P_i = C_i \cdot p^i \cdot (1-p)^{i+1}$ since for each unique binary tree, the probability of Rosetta making the probes exactly in the shape of that tree is equal to $p^i \cdot (1-p)^{i+1}$, since a positive occurs with probability p and a negative with probability $(1-p)$. By the approximation $C_i \simeq \frac{4^i}{i^{3/2}\sqrt{\pi}}$ [66] we have that

$$P_i \simeq \frac{(1-4\theta^2)^i}{i\sqrt{i\pi}} \cdot (1-p) \leq \frac{(1-4\theta^2)^i}{i\sqrt{i\pi}}$$

Then, $E_i = P_i(2i+1) \leq \frac{(1-4\theta^2)^i(2i+1)}{i\sqrt{i\pi}}$. We have that,

$$E - E_0 = \sum_{i=1}^{\infty} E_i \leq \sum_{i=1}^{\infty} \frac{(1-4\theta^2)^i(2i+1)}{i\sqrt{i\pi}} \leq \sum_{i=1}^{\infty} \frac{3i(1-4\theta^2)^i}{i\sqrt{i\pi}}$$
$$\leq \sum_{i=1}^{\infty} \frac{3(1-4\theta^2)^i}{\sqrt{\pi}} \leq \frac{3}{4\theta^2\sqrt{\pi}}$$

Finally, since E_0 is a constant and we have at most $2\log R$ dyadic ranges, we conclude that the probe cost to be at most $O(\frac{\log R}{\theta^2})$, with the infinite Rosetta. This also upper bounds the probe cost of the original Rosetta, and thus the overall expected complexity of a range query is less than $O(\frac{\log R}{\theta^2})$.

Non-Uniform Bits-Allocation. We can relax the assumption by allowing unequal FPRs among the Bloom filters. We denote the FPR of level-i Bloom filter by p_i. Let $p_{max} = \max\{p_i\}$ and $p_{min} = \min\{p_i\}$. Similar to the above derivation, we have

$$P_i \leq C_i p_{max}^i (1-p_{min})^{i+1}$$
$$\simeq \frac{4^i}{i^{3/2}\sqrt{\pi}} (p_{max}(1-p_{min}))^i \cdot (1-p_{min}) \tag{8}$$

To give the same complexity analysis as in the case where all p_i are equal, it is sufficient to let $p_{max}(1-p_{min}) < \frac{1}{4}$. Let $\theta' = \sqrt{\frac{1}{4} - p_{max}(1-p_{min})}$. Following our complexity analysis, we show that the expected probe time is $O(\frac{\log R}{\theta'^2})$.

4 INTEGRATING RANGE FILTERS TO AN LSM-BASED KEY-VALUE STORE

Standarization of Filters in RocksDB. We integrated Rosetta into RocksDB v6.3.6, a widely used LSM-tree based key-value store. We constructed a master *filter* template that exposes the API of the fundamental filter functionalities – *populating* the filter, *querying* the filter about the existence of one or more keys (point lookups and range scans), and *serializing* and *deserializing* the filter contents and its structure.

Implementing Full Filters. Each run of the LSM-tree is physically stored as one or more SST files - stands for Static Sorted Table and is effectively a contiguously stored set of ordered key-value pairs. A Rosetta instance is created for every SST file[4]. Filters are serialized before they are persisted

[4] Using block-based filters is also deprecated in RocksDB.

on disk[5]. During background compactions, a new filter instance is built for the merged content of the new SST, while the filter instances for the old SSTs are destroyed.

Minimizing Iterator-Overhead Through Configuration. For each range query, RocksDB creates a hierarchy of iterators. The number of iterators is equal to the number of SST files and thus the number of filter instances. Each such iterator is called a two-level iterator as it further maintains two child iterators to iterate over the data and non-data (filters and fence pointers) blocks (in the event that the corresponding filter returns a positive).

We found that a big part of the CPU cost of range queries in RocksDB [13] is due to the maintenance and consolidation of data coming from the hierarchical iterators. Unlike other levels, L0 in RocksDB comprises of more than one run flushed from the in-memory buffer. This design decision helps with writes in the key-value store. But at the same time, it means that we need to create multiple iterators for L0 causing a big CPU-overhead for empty queries (queries that do not move data). To reduce this overhead, we bound the number of L0 files (in our experiments we use 3 runs at L0[6]). Setting the number of L0 iterators too low may throttle compaction jobs and create long queues of pending files to be compacted. Hence, this value needs to be tuned based on the data size. We also restrict the growth of L0 thereby enforcing compaction and spawning most iterators only per level (and not per file) by setting `max_bytes_for_level_base`.

Implementation Overview of a Range Query. For each range query in `[low, high]`, we first probe all filter instances to verify the existance of the range. If all filters answer negative, we delete the iterator and return an empty result. If one or more filters answer positive, we probe the fence pointers and seek the lower end of the query using `seek(low)` which incurs I/O. If `seek()` returns a valid pointer, we perform a `next()` operation on the two-level iterator which advances the child iterators to read the corresponding fence pointers followed by the data blocks. When the iterator reaches (or crosses) the upper bound `high`, `next()` returns false.

Minimizing Deserialization-Overhead. To perform a filter probe for an SST file, we first need to retrieve the filter. If the filter comes from disk, we need to first deserialize it. Filter serialization incurs a low, one-time CPU-cost as it takes place once during the creation of a filter instance. However, by default, deserialization takes place as many times as the number of queries. Being a trie with a tree structure, the cost of deserialization in SuRF is low as it can choose to deserialize only a part of the filter as needed. On the other hand, Rosetta being based on Bloom filters cannot take advantage of partial filter bits. To this end, we construct a dictionary containing the mapping of the deserialized bits of each Rosetta instance and its corresponding run in the LSM-tree. Every time a Rosetta instance is constructed, we add this mapping to the dictionary which prevents frequent calls to deserialization at runtime. After each compaction, we delete the corresponding entries from the dictionary.

5 EXPERIMENTAL EVALUATION

We now demonstrate that by sacrificing filter probe cost, Rosetta achieves drastically lower FPR, bringing up to 70% net improvement in end-to-end performance in default RocksDB, and 50% against a SuRF-integrated RocksDB.

Baselines. We use the following baselines for comparison – state-of-the-art range query filter, SuRF[7] [74] integrated into RocksDB, RocksDB with Prefix Bloom filters and vanilla RocksDB with no filters but only fence pointers. For SuRF, we used the open-source implementation[8] with additional stubs to tailor the code path to the master template. For the other baselines (Prefix Bloom filters and vanilla RocksDB with no filters but only fence pointers), we use the default RocksDB implementation of each of them. All baselines use the same source RocksDB version v6.3.6.

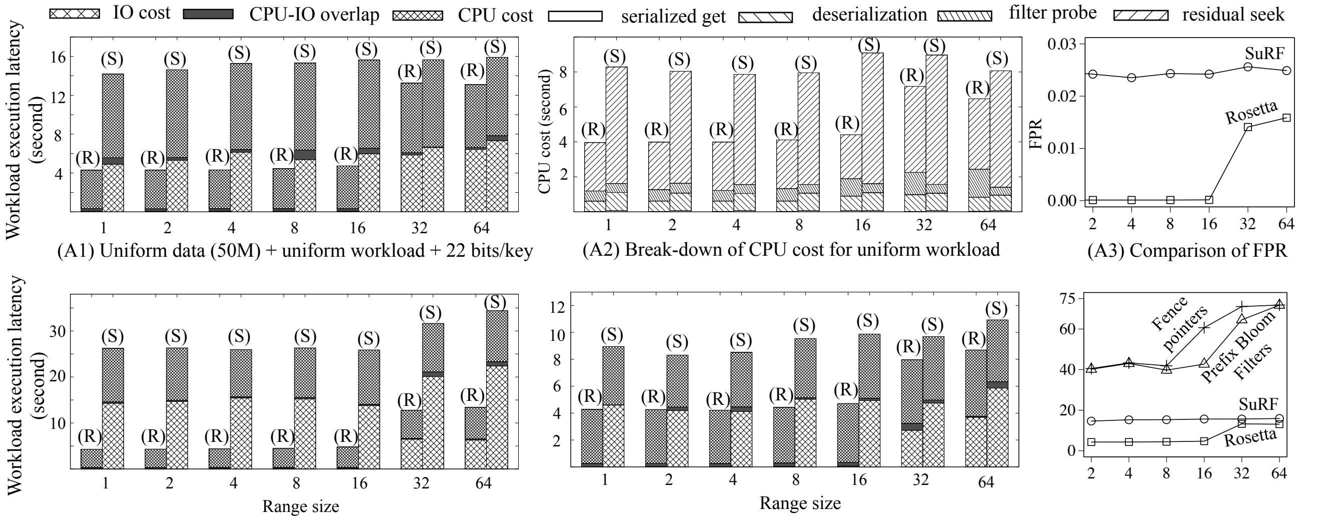
Workload and Setup. We generate YCSB key-value workloads that are variations of Workload E, a majority range scan workload modeling a web application use case [18]. We generate 50 million (50×10^6) keys each of size 64 bits (and 512 byte values) using both uniform, and normal distributions. We vary the maximum size of range queries, and target scan distribution. We set a default budget of 22 bits per key to each filter unless specified otherwise. Rosetta can tune itself to achieve a specific memory budget, while with SuRF we need to implicitly trade memory for FPR by increasing a built-in parameter called suffix length. When given a memory budget, we use a binary search on the suffix length to locate the SuRF version that gives the closest memory cost to the given memory budget. Exceptional cases happen when the minimum possible memory occupied by SuRF (when we set suffix length to 0) is still greater than the memory budget. In this case, we use the SuRF version that achieves its minimum possible memory budget. As filters are meant to improve performance by capturing false positives, and thus reducing the number of I/Os, a filter is best evaluated in presence of empty queries. Therefore, for all our experiments, our workload is comprised of empty range and point queries to capture worse-case behavior.

[5] For caching filters and fence pointers, we set the `cache_index_and_filter_blocks` to true. We also ensure that the fence pointers and filter blocks have a higher priority than data blocks when block cache is used. For this, we set `cache_index_and_filter_blocks_with_high_priority=true` and `pin_l0_filter_and_index_blocks_in_cache=true`.

[6] `level0_file_num_compaction_trigger=3`

[7] https://github.com/efficient/SuRF

[8] https://github.com/efficient/rocksdb



(A1) Uniform data (50M) + uniform workload + 22 bits/key (A2) Break-down of CPU cost for uniform workload (A3) Comparison of FPR

(B) Uniform data (50M) + correlated workload + 22 bits/key (C) Normal data (50M) + uniform workload + 22 bits/key (D) Rosetta outperforms all baselines

Figure 5: Rosetta improves end-to-end RocksDB performance across diverse workloads.

Experimental Infrastructure. We use a NUC5i3ryh server with a 5th Generation Intel Core i3 CPU, 8GB main memory, 1 TB SATA HDD at 7200 RPM, and 120GB SSD with version Samsung 850 EVO m.2 120GB Samsung EVO 850 SSD. The operating system is Ubuntu version 16.04 LTS.

Rosetta Improves the Performance of RocksDB. In our first experiment we demonstrate the benefits of Rosetta in a uniform workload while varying the query range from 1 to 64. For each range size, we measure the the end-to-end performance in terms of workload execution latency, for both Rosetta-integrated RocksDB and SuRF-integrated RocksDB. Figure 5(A1) shows the results denoting the measurements of Rosetta and SuRF using R and S, respectively. Figure 5(A1) also reports how the total cost for each system is spread among disk I/Os, CPU usage, and the overlap between the two. The total time emanating from disk I/Os and overall CPU usage are obtained using RocksDB-reported metric – `block_read_time` and `iter_seek_cpu_nanos`. The cost for I/O and CPU taken together exceeds the total cost due to cycle-stealing within the processor architecture which prevents wasting CPU cycles. Therefore, the CPU-IO overlap contributes to both the CPU and I/O cost. For measuring the sub-costs due to serialization, deserialization, and filter probe, we have embedded custom statistics into RocksDB using the internal `stopwatch()` support.

Figure 5(A1) shows that the performance of Rosetta is up to 4× better compared to SuRF. For shorter ranges till range size of 16, Rosetta incurs a negligible I/O overhead compared to SuRF with 22 bits per key. This behavior is due to the design differences between SuRF and Rosetta – SuRF employs a trie-culling design that prunes a full-trie's bottom elements, which contain important information for answering short range queries, whereas, Rosetta takes advantage of its lower level Bloom filters for answering short range queries. This significantly reduces the FPR achieved by Rosetta for shorter ranges, as shown in Figure 5(A3). The mean FPR of Rosetta for shorter ranges is 0.00012 whereas, it is 0.0242 for SuRF. This leads to a 70% reduction of I/O cost for shorter ranges in Rosetta. The gap closes for bigger ranges as more of the data has to be scanned and more areas in the filters need to be probed. As the range size increases beyond 16, although the FPR of Rosetta increases, it is still 1.5× less than that of SuRF which reduces the I/O cost by 13.6%.

Sacrificing Filter Probe Cost Leads To Better Performance with Rosetta. In Figure 5(A2), we magnify the CPU cost reported in Figure 5(A1) and break it down further into sub-costs: fetching serialized filter bits before a filter probe, deserializing filter bits, probing the filter, and performing residual operations `seek()` which includes 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. Figure 5(A2) shows that the cost of serialization is negligible (about 0.7% for both SuRF and Rosetta), and hence is not visible within the stacked breakdown. The CPU overhead of deserialization is significant yet close across both filters – about 14.5% and 11.1% for Rosetta and SuRF, respectively. Overall, the cost of serialization and deserializtion do not fluctuate across diverse range sizes as both these operations account for constant amount CPU work for each query.

The critical observation from Figure 5(A2) is that filter probe cost is a non-dominant portion of the overall CPU cost. Rosetta does incur a higher filter probe cost compared to SuRF (19% of total CPU cost compared to 5%). The reason is that Rosetta probes its internal Bloom filters once per dyadic interval of the target query range. Within each interval, the deeper Bloom filters are recursively probed until one of them returns a negative. On the other hand, with SuRF a probe involves only checking the existence of (i) a smaller number of (ii) fixed-length prefixes.

However, this CPU-intensive design leads to bigger gains for Rosetta. It allows Rosetta to achieve lower FPR (Figure 5(A3)) resulting in $2-4\times$ end-to-end improvement for RocksDB (Figure 5(A1)). The improved FPR does not only help with I/O but it also affects the fourth CPU-overhead contributing to the residual seek cost. On averaging over different range sizes, the overhead due to residual seek is about 65% and 82% for Rosetta and SuRF, respectively, thereby dominating the overall CPU cost. This overhead comprises of a (i) fixed component (invariable across varied range sizes) due to creation and maintenance of top-level iterators in RocksDB for all queries and (ii) a variable component due to longer usage of child iterators per query due to reading more data or fence pointer blocks in case of false or true positives. For example, as we compare the results across Figures 5(A1), (A2), and (A3), we notice that Rosetta shows a fixed overhead of about 2.7 seconds for shorter ranges, i.e., when FPR is significantly close to 0. The variable cost starts showing up as the FPR increases for range size 32 and 64. Due to high FPR in SuRF, we observe that this part of the CPU-overhead is significantly higher.

Therefore, sacrificing the CPU cost due to filter probe significantly reduces (a) the CPU overhead originating from `seek()` (which also dominates the total CPU cost) and of course (b) the total I/O cost of the workload due to lower FPR. The consolidated effects of (a) and (b) make a significant difference in the end-to-end performance for Rosetta.

Rosetta Enhances RocksDB for Key-Correlated Workloads. In order to investigate the benefits of Rosetta across diverse workload types, we repeat the previous experiment on both correlated and skewed workloads. First, we analyze workloads that exhibit prefix similarity between queried keys and stored keys. For this experiment, we introduce a correlation factor θ such that a range query with correlation degree θ has its lower bound at a distance θ from the lower bound generated using the distribution. Therefore, for a range query of size R, if the generated lower-bound is l, then we set the actual lower-bound to be at key $l+\theta$. For this experiment, we set $\theta = 1$.

Figure 5(B) depicts that when the workload entails a key correlation, Rosetta-integrated RocksDB on average achieves a 75% lower latency than that of SuRF-integrated RocksDB. This is because when the lower bound of a query falls close to a key, even if the range is empty, SuRF will always answer it as positive, since it will have likely culled that key's prefixes which contain the information to answer the query. In contrast, for any query, Rosetta is able to decompose it into prefix checking which is insensitive to the correlation.

Rosetta Enhances RocksDB for Skewed Key Distribution. For this experiment, we generate a dataset comprising of 50M keys following a normal distribution and execute a uniform workload on the dataset. In Figure 5(C), we demonstrate that Rosetta offers up to 2× performance benefits over SuRF with skewed keys for end-to-end performance in RocksDB. As more skewed keys are added to a run, the number of distinct keys within the run decreases. This leads to more prefix collisions which makes the trie-culling approach of SuRF vulnerable to false positives. On the other hand, Rosetta can resolve the collision through more probes at the deeper level Bloom filters. On average, Figure 5(C) shows that Rosetta enhances performance over SuRF by 52% and 19% for short and long range queries, respectively.

Rosetta improves default RocksDB by as much as 40x. We now demonstrate the benefits of Rosetta against the default filters of RocksDB using the 50M uniform workload setup (as in Figure 5(A)). In Figure 5(D), we observe that both Prefix Bloom filters and fence pointers offer significantly higher latency compared to both SuRF and Rosetta. This is because fence pointers cannot detect empty range queries especially when the range size is significantly lower than the size of the key domain (2^{64}). Prefix Bloom filters also cause a high FPR due to frequent prefix collisions as the range size increases. Rosetta offers the lowest latency among all baselines bringing an up to 40× improvement over default RocksDB. While the benefit reduces with larger range sizes, key-value stores are fundamentally meant to support point lookups, writes, and short range queries as opposed to scans and long range queries [38, 57, 60].

Rosetta's Construction Cost Adds Minimal Overhead. All types of filters in an LSM-based key-value store need to be reconstructed after a merge occurs. In our next experiment, we demonstrate the impact of this construction cost as we vary the data size. First, we set the size of L0 to be very high (30 runs) so that the entire data fits into L0 and there is no compaction. This way, we can isolate and measure the filter construction cost without taking into account the overlap with the compaction overhead. We vary the SST file size between 256 MBs and 1 GB which, in turn, increases the number of filter instances created from 5 to 59. Figure 6(A) shows that the average filter construction cost of Rosetta is about 14% less than that of SuRF. This is because Rosetta creates a series of Bloom filters which are densely packed arrays, causing fewer memory misses compared to populating a tree. Next, in order to study the more holistic impact of filters on write costs, we restored the size of L0 to 3 (as in other experiments) and measure the end-to-end workload execution latency for both the filters as well as vanilla RocksDB with only fence pointers (denoted by F in the results). In Figure 6(B), we break down the overall cost into separate costs due to reads and writes and writes are further broken down into sub-costs emanating from both compaction and filter creation. We observe that write-overhead is less for fence pointers as there is no filter construction overhead, but reads are expensive due to high false-positives. On the other hand, both SuRF and Rosetta have higher write-overhead due to the additional work of recreating filters after each compaction. To give more insights, we measure the total compaction time (T), and the bytes read (R) and written (W) during compaction and denote the compaction overhead to be T/(R+W). We observe this overhead to be 0.006μs/byte, 0.008μs/byte, and 0.007μs/byte for Rosetta, SuRF, and fence pointers respectively. Thus, the overall overhead in Rosetta due to the more complex filter structure is minimal compared to SuRF or even compared to having no filter at all. Critically, this overhead is overshadowed by the improved FPR which, in turn, drastically improves query performance on every run of the tree. Of course, for purely insert-heavy workloads, no filters should be maintained (Rosetta, SuRF, or Bloom filters) as they would be constructed bur never (or hardly ever) be used.

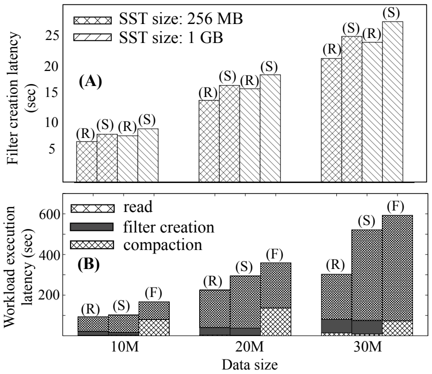


Figure 6: Rosetta incurs minimal filter construction overhead during compactions.

Rosetta Maintains RocksDB's Point-Query Efficiency. We now show that unlike other filters, Rosetta not only brings benefits in range queries but also does not hurt point queries. We experiment with a uniform workload of 50M keys. We vary the bits per key from 10 to 20 and measure the change in FPR for both SuRF and Rosetta, compared to the Bloom filters on RocksDB. Figure 7 shows the results. SuRF-Hash and SuRF-real offer a drastically higher FPR. For example, the point-query FPR of SuRF-Hash is 10× worse compared to that of the default Bloom filter in RocksDB as it incurs more hash collisions with large number of keys. Similarly, if we were to use Prefix Bloom filters, FPR for point queries goes all the way up to 1 in this case. On the other hand, Rosetta only uses its last level Bloom filter which effectively indexes all bits of the keys and thus, brings great FPR. It keeps up the performance for larger memory budgets compared to the Bloom filters on RocksDB and even improves on it as more memory becomes available.

The most critical observation from this experiment is that the filters which are meant to be used for range queries, i.e., SuRF-Real and Prefix Bloom filters cannot give adequate point query performance. Thus, a key-value store would need to either maintain two filters per run (one for point queries and one for range queries), or lose out on performance for one of the query types. On the other hand, Rosetta can efficiently support both query types.

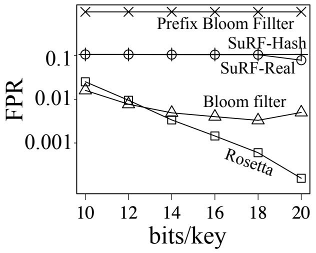


Figure 7: Rosetta maintains the point query performance on RocksDB.

Robust Benefits Across the Memory Hierarchy. For this experiment, we compare SuRF and Rosetta outside RocksDB to verify whether we see the same behavior as in the full system experiments. We use 10M keys each of size 64 bits. The experiment is performed using a uniform workload. We measure end-to-end latency to access data in memory, HDD, or SSD, as shown in Figure 9. For all storage media, Rosetta does spend more time in filter probe cost (represented by *p*), but the improved resulting FPR translates to end-to-end benefits even in memory. Thus, this result verifies the same end-to-end behavior we observe when integrating the filters in RocksDB.

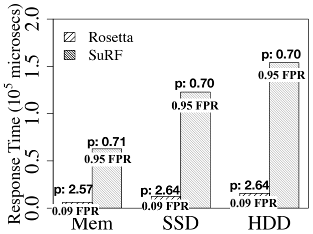


Figure 9: Rosetta improves performance across the memory hierarchy.

Rosetta Provides Competitive Performance with String Data. Given that SuRF is based on a trie design while Rosetta is based on hashing, the natural expectation with string data is that SuRF will be better. We now show that Rosetta achieves competitive performance to SuRF with string data. We use again a standalone experiment here, outside RocksDB to further isolate the properties of the filters.

We use a variable-length string data set, Wikipedia Extraction (WEX)[9] comprising of a processed dump (of size 6M) of English language in Wikipedia. The wiki markup for each article is transformed into machine-readable XML, and common relational features such as templates, infoboxes, categories, article sections, and redirects are extracted in tabular form. We generate a workload comprising of 1 million

[9] https://aws.amazon.com/de/datasets/wikipedia-extraction-wex/

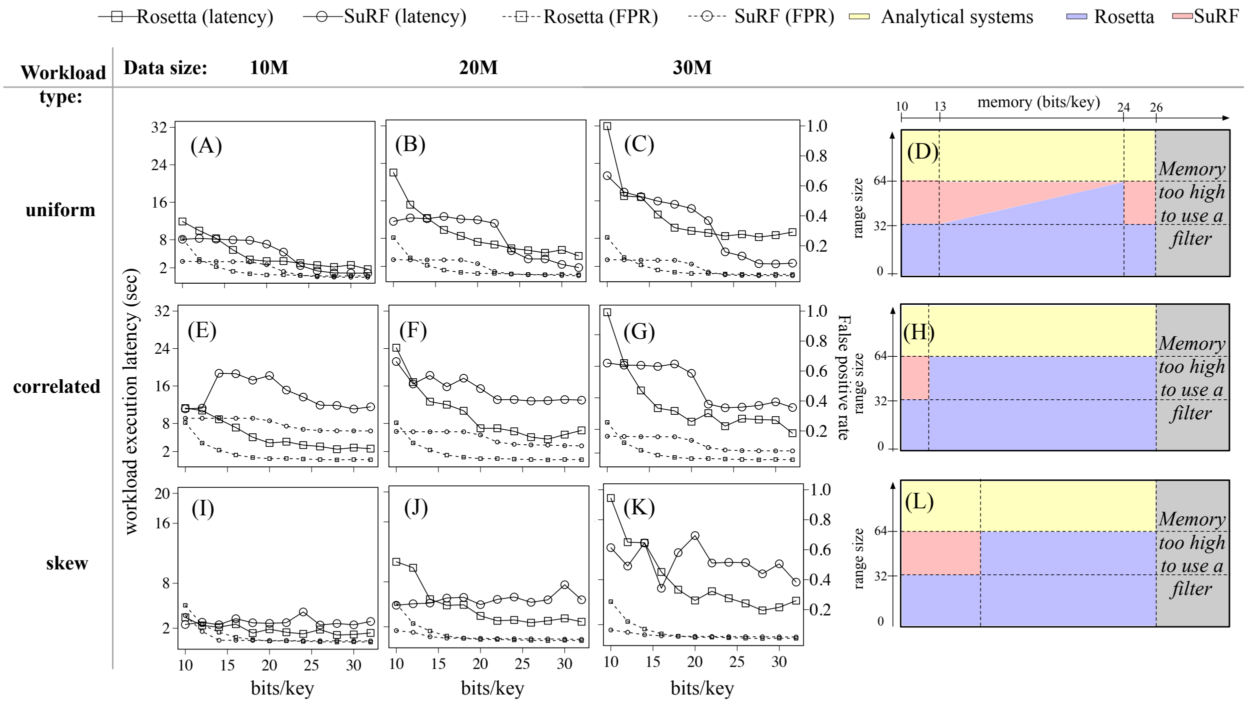


Figure 8: Rosetta exhibits a superior FPR-Memory tradeoff, compared to SuRF, across multiple workloads. Rosetta serves a larger spectrum of key-value workloads under diverse range sizes and memory budget.

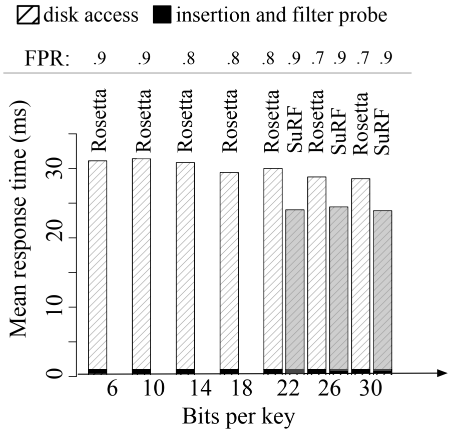


Figure 10: Rosetta achieves good performance with strings across all memory budgets.

queries drawn uniformly from the data set. We set the range size to be 128 and we vary the memory budget from 6 bits per key to 30 bits per key. For each setting of memory budget, we record the FPR, the mean execution time of each query, and the filter probe cost. Figure 10 shows that overall both Rosetta and SuRF offer similar FPR with SuRF outperforming Rosetta marginally. However, we also observe that there is a wide spectrum of memory budgets where SuRF cannot be applied as it demands a minimum memory of 20 bits per key to store the prefixes. Rosetta can support workloads with low memory budget and still keep up a low FPR leading to robust end-to-end performance.

Rosetta Exhibits a Superior FPR-Memory Tradeoff Across Diverse Workloads and Memory Budgets. We now return to the full RocksDB experiments and proceed to give a more high level positioning on when to use which filter. We do that by fixing the range size to the longest range size of 64 (i.e., the worst case for Rosetta) varying the bits per key allocated to each filter from 10 to 32. We also vary the data set to contain 10M to 30M keys. We measure both the FPR and the end-to-end workload execution latency for uniform, correlated, and skewed workloads.

Figures 8(A)-(C) depict results for uniform workloads. Figures 8(E)-(G) depict results for correlated workloads. And finally, Figures 8(I)-(K) depict result for skewed workloads. Across all workloads, we observe consistently that Rosetta achieves an overall superior performance and that it can make better use of additional memory to lower the FPR. Intuitively, this is because its core underlying structure is Bloom filters where every additional bit may be utilized more, compared to SuRF where where the prefixes are culled beyond a definite length. Although the additional suffixes for SuRF-real do help in reducing the FPR, it still falls behind the FPR achieved by Rosetta.

When memory budget is very tight, SuRF does gain over Rosetta. However, the latency achieved at these constrained

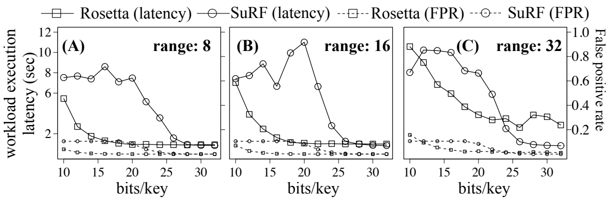


Figure 11: Rosetta always improves performance for smaller range sizes.

memory levels is substantially worse than with a few additional bits per key. Therefore, for most practical applications, especially on the cloud, it makes sense to devote the additional bits. This decision also depends on other factors such as the cloud cost policy, e.g., increasing memory budget may lead to upgrading the hardware to high-end virtual machines, which also improves the cap on the maximum allowed throughput and hence, may significantly reduce latency. Figure 11 repeats the same experiment for smaller ranges and shows that Rosetta is nearly always better.

Holistic Positioning of SuRF and Rosetta. Overall, our results indicate that both SuRF and Rosetta can bring significant advantages complementing each other across different workloads. Figures 8(D), (H), and (L) summarize when each filter is most suitable with respect to data size, range size and memory budget. All three figures denote the spectrum of scenarios of executing workloads on key-value stores. Within each figure, we partition horizontally to indicate different range sizes and vertically to indicate different memory budgets. The area beyond a memory budget of 26 bits per key is too high for filters to be used and the area beyond range size of 64 is meant to be served using analytical systems or column stores. Our observation is that, Rosetta works better for short and medium range queries across all workload-memory combinations. For long range queries, although the point of separation slightly varies with the workload type, overall SuRF performs better with low memory budget and Rosetta covers the area with high memory budget.

6 RELATED WORK

We now discuss further related work on range filtering.

Range Filter with Optimal Space. Goswami et al. [44] analyzes the space lower bound for a range filter. The primary technique of the proposed data structure is based on a pairwise-independent and locality-hash function. The keys are hashed by the function, and then a weak prefix search data structure is built based on the hashed keys for answering range queries. Their algorithm does not use Bloom filters, dyadic ranges, nor memory allocation strategies, which are the core design components in Rosetta.

Dyadic Intervals for Other Applications. The idea of decomposing large ranges of data to dyadic intervals has been applied widely for streaming applications [19, 20, 23] to enable computation of sophisticated aggregates – quantiles [41], wavelets [40], and histograms [39]. The canonical dyadic decomposition has also been exploited to determine and track the most frequently used data within a data store [22] as well as to support range query investigations for privacy-preserving data analysis, i.e., processing noisy variants of confidential user data [21]. Other efforts in this direction include designing persistent sketches so that the sketches can be queried about a prior state of data [59, 70] and facilitating range queries using multi-keyword search with security-performance trade-offs [31].

Among these, the closest to our work are the 1) Count-Min sketch [20, 23] and 2) Persistent Bloom Filters (PBF) [59]. The Count-Min sketch integrates bloom filters within dyadic partitions or Segment Trees. The work demonstrates the efficiency of approximate query answering using the integrated framework. However, Rosetta is different from the Count-Min sketch framework in the following ways: (i) Rosetta handles diverse workloads composed of different query types and interleaved access patterns, (ii) Rosetta specifically consider the CPU-memory contention affecting workload performance, and (iii) Rosetta considers resource-constrained situations where it is crucial to design an optimal allocation scheme for storing the filters and dyadic data partitions.

PBF [59] solves a different problem, i.e., answering a point query over a time range (e.g. does this IP address appear between 9pm and 10pm?). Our solutions use similar design elements, e.g., Bloom filters and dyadic range decomposition, but ours differs significantly in several ways. Firstly, we achieve a new balance, trading more CPU time for a better FPR. Furthermore, Rosetta achieves memory-optimality with respect to the maximum range size and FPR, due to being able to effectively assign different memory budgets to different Bloom filters.

7 CONCLUSION

We introduce Rosetta, a probabilistic range filter designed specifically for LSM-tree based key-value stores. The core idea is to sacrifice filter probe time which is not visible in end-to-end key-value store performance in order to significantly reduce FPR. Rosetta brings benefits for short and medium range queries across various workloads (uniform, skewed, correlated) without hurting point queries.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their valuable feedback. This work is partially funded by the USA Department of Energy project DE-SC0020200.

REFERENCES

[1] Abadi, D. J., Boncz, P. A., Harizopoulos, S., Idreos, S., and Madden, S. The Design and Implementation of Modern Column-Oriented Database Systems. *Foundations and Trends in Databases 5*, 3 (2013), 197–280.

[2] Alsubaiee, S., Altowim, Y., Altwaijry, H., Behm, A., Borkar, V. R., Bu, Y., Carey, M. J., Cetindil, I., Cheelangi, M., Faraaz, K., Gabrielova, E., Grover, R., Heilbron, Z., Kim, Y.-S., Li, C., Li, G., Ok, J. M., Onose, N., Pirzadeh, P., Tsotras, V. J., Vernica, R., Wen, J., and Westmann, T. AsterixDB: A Scalable, Open Source BDMS. *Proceedings of the VLDB Endowment 7*, 14 (2014), 1905–1916.

[3] Alsubaiee, S., Carey, M. J., and Li, C. Lsm-based storage and indexing: An old idea with timely benefits. In *Second International ACM Workshop on Managing and Mining Enriched Geo-Spatial Data* (2015), pp. 1–6.

[4] Apache. Accumulo. *https://accumulo.apache.org/*.

[5] Apache. HBase. *http://hbase.apache.org/*.

[6] Armstrong, T. G., Ponnekanti, V., Borthakur, D., and Callaghan, M. Linkbench: a database benchmark based on the facebook social graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (2013), pp. 1185–1196.

[7] Athanassoulis, M., Kester, M. S., Maas, L. M., Stoica, R., Idreos, S., Ailamaki, A., and Callaghan, M. Designing Access Methods: The RUM Conjecture. In *Proceedings of the International Conference on Extending Database Technology (EDBT)* (2016), pp. 461–466.

[8] Balmau, O., Didona, D., Guerraoui, R., Zwaenepoel, W., Yuan, H., Arora, A., Gupta, K., and Konka, P. TRIAD: Creating Synergies Between Memory, Disk and Log in Log Structured Key-Value Stores. In *Proceedings of the USENIX Annual Technical Conference (ATC)* (2017), pp. 363–375.

[9] Bender, M. A., Farach-Colton, M., Johnson, R., Kraner, R., Kuszmaul, B. C., Medjedovic, D., Montes, P., Shetty, P., Spillane, R. P., and Zadok, E. Don't Thrash: How to Cache Your Hash on Flash. *Proceedings of the VLDB Endowment 5*, 11 (2012), 1627–1637.

[10] Bloom, B. H. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM 13*, 7 (1970), 422–426.

[11] Bortnikov, E., Braginsky, A., Hillel, E., Keidar, I., and Sheffi, G. Accordion: Better Memory Organization for LSM Key-Value Stores. *Proceedings of the VLDB Endowment 11*, 12 (2018), 1863–1875.

[12] Bu, Y., Borkar, V. R., Jia, J., Carey, M. J., and Condie, T. Pregelix: Big(ger) Graph Analytics on a Dataflow Engine. *Proceedings of the VLDB Endowment 8*, 2 (2014), 161–172.

[13] Callaghan, M. CPU overheads for RocksDB queries. http://smalldatum.blogspot.com/2018/07/query-cpu-overheads-in-rocksdb.html, July 2018.

[14] Cao, Z., Chen, S., Li, F., Wang, M., and Wang, X. S. LogKV: Exploiting Key-Value Stores for Log Processing. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)* (2013).

[15] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. Bigtable: A distributed storage system for structured data. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2006), pp. 205–218.

[16] Chen, G. J., Wiener, J. L., Iyer, S., Jaiswal, A., Lei, R., Simha, N., Wang, W., Wilfong, K., Williamson, T., and Yilmaz, S. Realtime data processing at facebook. In *Proceedings of the 2016 International Conference on Management of Data* (2016), pp. 1087–1098.

[17] CockroachLabs. CockroachDB. *https://github.com/cockroachdb/cockroach*.

[18] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. Benchmarking cloud serving systems with YCSB. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)* (2010), pp. 143–154.

[19] Cormode, G. Sketch techniques for approximate query processing. In *Foundations and Trends in Databases* (2011).

[20] Cormode, G., Garofalakis, M. N., Haas, P. J., and Jermaine, C. Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches. *Foundations and Trends in Databases 4*, 1-3 (2012), 1–294.

[21] Cormode, G., Kulkarni, T., and Srivastava, D. Answering Range Queries Under Local Differential Privacy. In *arXiv:1812.10942* (2018).

[22] Cormode, G., and Muthukrishnan, S. What's Hot and What's Not: Tracking Most Frequent Items Dynamically. In *PODS* (2003).

[23] Cormode, G., and Muthukrishnan, S. An Improved Data Stream Summary: The Count-Min Sketch and Its Applications. In *LATIN 2004: Theoretical Informatics, 6th Latin American Symposium, Buenos Aires, Argentina, April 5-8, 2004, Proceedings* (2004), vol. 2976, pp. 29–38.

[24] Dayan, N., Athanassoulis, M., and Idreos, S. Monkey: Optimal Navigable Key-Value Store. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (2017), pp. 79–94.

[25] Dayan, N., Athanassoulis, M., and Idreos, S. Optimal Bloom Filters and Adaptive Merging for LSM-Trees. *ACM Transactions on Database Systems (TODS) 43*, 4 (2018), 16:1–16:48.

[26] Dayan, N., Bonnet, P., and Idreos, S. GeckoFTL: Scalable Flash Translation Techniques For Very Large Flash Devices. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (2016), pp. 327–342.

[27] Dayan, N., and Idreos, S. Dostoevsky: Better space-time trade-offs for lsm-tree based key-value stores via adaptive removal of superfluous merging. In *Proceedings of the 2018 International Conference on Management of Data* (2018), pp. 505–520.

[28] Dayan, N., and Idreos, S. The log-structured merge-bush & the wacky continuum. In *Proceedings of the 2019 International Conference on Management of Data* (2019), pp. 449–466.

[29] de Berg, M., van Kreveld, M., Overmars, M., and Schwarzkopf, O. C. More geometric data structures. In *Computational Geometry*. 2000, pp. 211–233.

[30] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. Dynamo: Amazon's Highly Available Key-value Store. *ACM SIGOPS Operating Systems Review 41*, 6 (2007), 205–220.

[31] Demertzis, I., Papadopoulos, S., Papapetrou, O., Deligiannakis, A., and Garofalakis, M. Practical Private Range Search Revisited. In *ACM SIGMOD* (2016).

[32] Dgraph. Badger Key-value DB in Go. *https://github.com/dgraph-io/badger*.

[33] Dharmapurikar, S., Krishnamurthy, P., and Taylor, D. E. Longest prefix matching using bloom filters. In *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (2003), pp. 201–212.

[34] Dong, S., Callaghan, M., Galanis, L., Borthakur, D., Savor, T., and Strum, M. Optimizing Space Amplification in RocksDB. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)* (2017).

[35] Facebook. MyRocks. *http://myrocks.io/*.

[36] Facebook. RocksDB. *https://github.com/facebook/rocksdb*.

[37] Fan, B., Andersen, D. G., Kaminsky, M., and Mitzenmacher, M. Cuckoo Filter: Practically Better Than Bloom. In *Proceedings of the ACM International on Conference on emerging Networking Experiments and Technologies (CoNEXT)* (2014), pp. 75–88.

[38] Gembalczyk, D., Schuhknecht, F. M., and Dittrich, J. An Experimental Analysis of Different Key-Value Stores and Relational Databases. In *Datenbanksysteme fur Business, Technologie und Web (BTW'17)* (2017).

[39] Gilbert, A., Guha, S., Indyk, P., Kotidis, Y., Muthukrishnan, S., and Strauss, M. Fast, small-space algorithms for approximate histogram maintenance. In *Proceedings of the 34th ACM Symposium on Theory of*

Computing (2002).

[40] Gilbert, A., Kotidis, Y., Muthukrishnan, S., and Strauss, M. Surfing wavelets on streams: One-pass summaries for approximate aggregate queries. In *Proceedings of International Conference on Very Large Data Bases* (2003).

[41] Gilbert, A. C., Kotidis, Y., S.Muthukrishnan, and M.Strauss. How to summarize the universe: Dynamic maintenance of quantiles. In *Proceedings of International Conference on Very Large Data Bases* (2002).

[42] Golan-Gueta, G., Bortnikov, E., Hillel, E., and Keidar, I. Scaling Concurrent Log-Structured Data Stores. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)* (2015), pp. 32:1–32:14.

[43] Google. LevelDB. *https://github.com/google/leveldb/*.

[44] Goswami, M., Grønlund, A., Larsen, K. G., and Pagh, R. Approximate range emptiness in constant time and optimal space. In *Proceedings of the twenty-sixth annual ACM-SIAM symposium on Discrete algorithms* (2014), pp. 769–775.

[45] Idreos, S., Dayan, N., Qin, W., Akmanalp, M., Hilgard, S., Ross, A., Lennon, J., Jain, V., Gupta, H., Li, D., and Zhu, Z. Design continuums and the path toward self-designing key-value stores that know and learn. In *Biennial Conference on Innovative Data Systems Research (CIDR)* (2019).

[46] Jagadish, H. V., Narayan, P. P. S., Seshadri, S., Sudarshan, S., and Kanneganti, R. Incremental Organization for Data Recording and Warehousing. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)* (1997), pp. 16–25.

[47] Jannen, W., Yuan, J., Zhan, Y., Akshintala, A., Esmet, J., Jiao, Y., Mittal, A., Pandey, P., Reddy, P., Walsh, L., Bender, M. A., Farach-Colton, M., Johnson, R., Kuszmaul, B. C., and Porter, D. E. BetrFS: A Right-optimized Write-optimized File System. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)* (2015), pp. 301–315.

[48] Jermaine, C., Omiecinski, E., and Yee, W. G. The Partitioned Exponential File for Database Storage Management. *The VLDB Journal 16*, 4 (2007), 417–437.

[49] Kahveci, T., and Singh, A. Variable length queries for time series data. In *Proceedings 17th International Conference on Data Engineering* (2001), pp. 273–282.

[50] Kondylakis, H., Dayan, N., Zoumpatianos, K., and Palpanas, T. Coconut palm: Static and streaming data series exploration now in your palm. In *Proceedings of the 2019 International Conference on Management of Data* (2019), pp. 1941–1944.

[51] Kondylakis, H., Dayan, N., Zoumpatianos, K., and Palpanas, T. Coconut: sortable summarizations for scalable indexes over static and streaming data series. *The VLDB Journal* (09 2019).

[52] Kyrola, A., and Guestrin, C. Graphchi-db: Simple design for a scalable graph database system–on just a pc. *arXiv preprint arXiv:1403.0701* (2014).

[53] Lakshman, A., and Malik, P. Cassandra - A Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review 44*, 2 (2010), 35–40.

[54] Li, Y., He, B., Yang, J., Luo, Q., Yi, K., and Yang, R. J. Tree Indexing on Solid State Drives. *Proceedings of the VLDB Endowment 3*, 1-2 (2010), 1195–1206.

[55] LinkedIn. Voldemort. *http://www.project-voldemort.com*.

[56] Lu, L., Pillai, T. S., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. WiscKey: Separating Keys from Values in SSD-conscious Storage. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)* (2016), pp. 133–148.

[57] Luo, C., and Carey, M. J. LSM-based Storage Techniques: A Survey. *arXiv:1812.07527v3* (2019).

[58] O'Neil, P. E., Cheng, E., Gawlick, D., and O'Neil, E. J. The log-structured merge-tree (LSM-tree). *Acta Informatica 33*, 4 (1996), 351–385.

[59] Peng, Y., Guo, J., Li, F., Qian, W., and Zhou, A. Persistent bloom filter: Membership testing for the entire history. In *Proceedings of the 2018 International Conference on Management of Data* (2018), pp. 1037–1052.

[60] Pirza, P., Tatemura, J., Po, O., and Hacigumus, H. Performance Evaluation of Range Queries in Key Value Stores. *J Grid Computing* (2012).

[61] Raju, P., Kadekodi, R., Chidambaram, V., and Abraham, I. PebblesDB: Building Key-Value Stores using Fragmented Log-Structured Merge Trees. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)* (2017), pp. 497–514.

[62] Ren, K., Zheng, Q., Arulraj, J., and Gibson, G. SlimDB: A Space-Efficient Key-Value Storage Engine For Semi-Sorted Data. *Proceedings of the VLDB Endowment 10*, 13 (2017), 2037–2048.

[63] Sears, R., Callaghan, M., and Brewer, E. Rose: Compressed, log-structured replication. 526–537.

[64] Sears, R., and Ramakrishnan, R. bLSM: A General Purpose Log Structured Merge Tree. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (2012), pp. 217–228.

[65] Shetty, P., Spillane, R. P., Malpani, R., Andrews, B., Seyster, J., and Zadok, E. Building Workload-Independent Storage with VT-trees. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)* (2013), pp. 17–30.

[66] Stanley, R. P. *Catalan numbers*. Cambridge University Press, 2015.

[67] Thonangi, R., and Yang, J. On Log-Structured Merge for Solid-State Drives. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)* (2017), pp. 683–694.

[68] Van Kreveld, M., Schwarzkopf, O., de Berg, M., and Overmars, M. *Computational geometry algorithms and applications*. Springer, 2000.

[69] Vincon, T., Hardock, S., Riegger, C., Oppermann, J., Koch, A., and Petrov, I. Noftl-kv: Tackling write-amplification on kv-stores with native storage management.

[70] Wei, Z., Luo, G., Yi, K., Du, X., and Wen, J. R. Persistent Data Sketching. In *ACM SIGMOD* (2015).

[71] WiredTiger. Source Code. *https://github.com/wiredtiger/wiredtiger*.

[72] Wu, X., Xu, Y., Shao, Z., and Jiang, S. LSM-trie: An LSM-tree-based Ultra-Large Key-Value Store for Small Data Items. In *Proceedings of the USENIX Annual Technical Conference (ATC)* (2015), pp. 71–82.

[73] Yahoo. bLSM: Read-and latency-optimized log structured merge tree. *https://github.com/sears/bLSM* (2016).

[74] Zhang, Y., Li, Y., Guo, F., Li, C., and Xu, Y. ElasticBF: Fine-grained and Elastic Bloom Filter Towards Efficient Read for LSM-tree-based KV Stores. In *Proceedings of the USENIX Conference on Hot Topics in Storage and File Systems (HotStorage)* (2018).