Oasis: An Optimal Disjoint Segmented Learned Range Filter

Guanduo Chen[1]
Fudan University
gdchen22@m.fudan.edu.cn

Zhenying He
Fudan University
zhenying@fudan.edu.cn

Meng Li[2]
Nanjing University
meng@nju.edu.cn

Siqiang Luo[†]
Nanyang Technological University
siqiang.luo@ntu.edu.sg

ABSTRACT

The learning-enhanced data structure has inspired the development of the range filter, bringing significantly better false positive rate (FPR) than traditional non-learned range filters. Its core idea is to employ piece-wise linear functions that uniformly map the entire key space into a bitmap sequentially. Nonetheless, such uniform mapping can be space-ineffective, impacting FPRs.

This paper introduces Oasis, a novel learned range filter that divides the key space into disjointed intervals by excluding large empty ranges explicitly and optimally maps those unpruned intervals into a compressed bitmap. The configuration optimality in Oasis is guaranteed by a careful theoretical analysis. To enhance the versatility of Oasis, we further propose Oasis+, which integrates the design space of both learned and non-learned filters, delivering robust performance across a wide range of workloads. We evaluate the performance of both Oasis and Oasis+ when integrated into the key-value system RocksDB, using a diverse set of real-world and synthetic datasets and workloads. In RocksDB, Oasis and Oasis+ improve the performance by up to 1.4× and 6.2× when compared to state-of-the-art learned and non-learned range filters.

PVLDB Reference Format:
Guanduo Chen, Zhenying He, Meng Li, Siqiang Luo. Oasis: An Optimal Disjoint Segmented Learned Range Filter. PVLDB, 17(8): 1911 - 1924, 2024.
doi:10.14778/3659437.3659447

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/Woooooow-Pro/Oasis-RangeFilter.

1 INTRODUCTION

Range Filters are essential for data-intensive applications. Range queries are fundamental database operations that retrieve all records within a specified query range $[l, r]$ from a key set $\mathbf{D}$. Range queries have diverse applications, such as similarity search [51] and anomaly detection [22] in time-series data, spatial network [39], block-chain database [52], web application [7], and distributed storage [43]. However, range queries are also I/O expensive operations, which involve scanning the whole dataset to identify all qualified records. To address this issue, range filters [1, 18, 21, 33, 37, 47, 50, 54] have been proposed. Range filters can quickly determine whether any records fall within a given query range, potentially returning false positives but guaranteeing no false negatives. Once a range filter confirms that no elements exist within the range, the overall range query can be safely terminated immediately without accessing the disk. This early termination avoids unnecessary data scanning, largely improving query efficiency.

Notably, the growing interest in Log-Structured Merge-trees (LSM-trees) [2, 4, 5, 9, 24, 30, 32, 36, 40–42, 44, 49, 55, 56, 58] by both the research and industrial communities has spurred architectural adaptations. LSM-trees employ Bloom filters [3] to enhance point query efficiency [8, 13, 14], which extracts the value for a given key. However, the demand for range queries in LSM-tree based applications has prompted either architectural redesigns to support range queries efficiently [59] or the replacement of these Bloom filters with general range filters capable of effectively supporting both point and range membership queries [13, 21, 33, 37, 47, 54].

State-of-the-art range filters. The state-of-the-art range filters can be classified into two types: prefix-based range filter (*i.e.*, encoding the key prefix) [10, 21, 33, 37, 50, 54] and learning-based range filter (*i.e.*, encoding empty ranges with a monotonical learned-model) [47]. The Succinct Range Filter (SuRF) [54] is a prefix range filter based on a compact trie structure. Rosetta [33] enhanced SuRF regarding short range queries by employing a series of stacked prefix Bloom filters, each with a unique prefix length. Following Rosetta, REncoder [50] and bloomRF [37] improve long range query performance by partially memorizing the key suffixes. Further, by assuming that the queries can be sampled, Proteus [21] augments SuRF with an appended prefix-based Bloom filter, whose optimal configuration is decided according to the sampled queries.

Rather than solely focusing on enhancing prefix-encoding efficiency, SNARF [47] first introduces the learned Range Filter (LRF), which cuts down the false positive rate (FPR)[1] by more than one order of magnitude when compared with state-of-the-art prefix-based filters. The general idea of SNARF is learning-based range encoding: (1) dividing the whole key space into multiple adjoined intervals, each containing the same number of keys; (2) allocating each interval a fixed-length bitmap segment; (3) training a linear function for each interval that maps each key within the interval to

[1] Work done when the author was working as a research assistant under the supervision of Siqiang Luo.
[2] Meng Li is affiliated with the State Key Laboratory for Novel Software Technology at Nanjing University.
[†] Siqiang Luo is the corresponding author.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 17, No. 8 ISSN 2150-8097.
doi:10.14778/3659437.3659447

[1]FPR is calculated as the ratio of false positive results to the total number of negatives.

a specific bit in its corresponding bitmap segment and sets that bit to 1. This approach maps the empty range[2] between two adjoined keys onto a series of consecutive zero bits in the bitmap segment. During the range query process, the left/right boundary is mapped onto a single bit in the bitmap segment, followed by verifying the presence of any non-zero bits between the two mapped bits.

Encoding large empty ranges is space-inefficient. Although the range encoding method in SNARF works well for small ranges, space cost becomes an issue when it encounters queries with large ranges. To better illustrate this issue, we provide a series of simple examples, all based on the same set of nine keys, as depicted in Figure 1. In each sub-figure, the right side exhibits one workload comprising four true negative range queries. Queries highlighted in red signify instances where the filter fails to correctly identify them as empty, while those in green indicate successful identification.

The example at the top of Figure 1 shows how SNARF encodes a key space, with the key space divided into two adjoined intervals having the same number of keys: interval $[0, 1100]$ and interval $[1100, 1500]$. The boundaries 0, 1100, 1500 are highlighted in blue. Notably, SNARF's intervals share boundaries with its left and right neighborhoods, overlapping with each other at boundaries. In this example, two intervals share the boundary of 1100. Also, SNARF uniformly allocates a 40-bit bitmap segment to each interval. Each interval is then assigned a model for mapping within-interval keys to the corresponding bitmap segment. In addition, the model of the first interval maps its interval boundary 0 (or 1100) to the 0th (or 40th) bits, while the model of the second interval maps its interval boundary 1100 (or 1500) to the 40th (or 79th) bits, respectively.

In this example, SNARF encodes the large empty range $(20, 1000)$ into a 35-bit bitmap segment, from the 1st bit to the 35th bit, consuming nearly half of the space budget. As a result, other relatively smaller empty ranges will suffer from bits starvation (*i.e.*, smaller space budget) due to the exploitation from such a large empty range. Then, once queries fall within these small ranges, they will suffer from a higher FPR. As shown in Figure 1, each query's bitmap positions overlap with those of existing keys, leading to all four queries returning false positive answers. This motivates us to consider: are there better ways to optimize the space usage for higher range filter quality? To answer this question, we have three proposals.

Proposal 1: recording rather than encoding. To avoid excessive memory consumption by large empty ranges, we propose to record the range boundaries directly rather than encoding the whole range into a long bitmap segment. For instance, in Figure 1, SNARF allocates excessive bits to encode an empty range from 20 to 1000. Our solution, the *space pruning* method, avoids encoding such large empty ranges by explicitly recording its boundaries (20 and 1000). As shown in Figure 1(A), the key space is partitioned into two disjoint intervals: $[0, 20]$ and $[1000, 1500]$. This allows us to project the range query $[4, 6]$ to the 4th-6th bits, which are all zero and thus enable correct answers. Similarly, query $[24, 26]$ immediately returns empty, as it falls in the pruned range (marked by strikethrough green). However, recording the boundaries may introduce extra space overhead, and hence, it is a challenge to balance the portions of encoding and recording for desired query performance.

[2]The empty range indicates the absence of any existing keys within it.

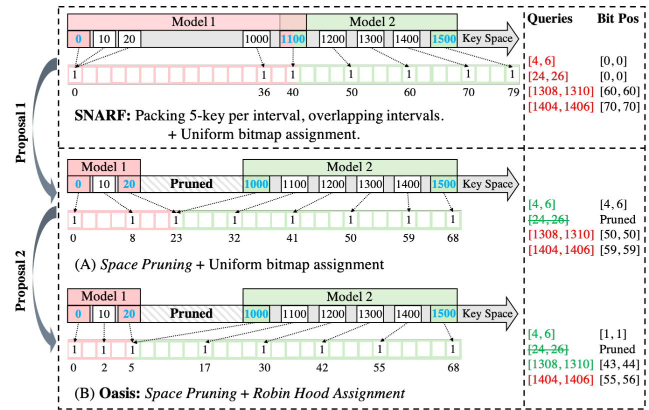

Figure 1: Learned range filter examples: SNARF with a bitmap of 80 bits; (A) LRF with range pruning method; (B) Oasis: LRF with interval bitmap allocated using Robin Hood assignment method. Blue Keys = stored interval boundary keys; Green Range Queries = true negatives; Red Range Queries = false positives for queries; Strikethrough Green Range Queries = true negatives produced by range pruning.

Proposal 2: sharing rather than exploitation. With those large ranges being pruned, the remaining key space is then divided into multiple disjoint intervals, which then raises another question: how to allocate a memory budget for the bitmap of each interval, as these intervals may contain different numbers of keys. One naive approach is to allocate bits to each interval proportionally to the number of keys its contains, similar to Rosetta and SNARF. However, such a strategy may fall short when the distance between adjoined keys varies significantly. For example, in Figure 1(A), the first small interval $[0, 20]$ contains only three keys, while the second, $25\times$ larger interval $[1000, 1500]$, has just six keys. Consequently, each bit in the second interval needs to cover a much larger space than a bit in the first interval, leading to an imbalance in bit utilization. To tackle this issue, we introduce a metric named *BitSpan Resolution* (BsR) and a method to optimize the overall BsR. BsR represents the average range length covered by one bit within an interval, calculated as the interval length divided by the number of bits in its bitmap. A lower BsR indicates better query performance (*i.e.*, lower FPR). To achieve a global optimal query performance, we propose a bitmap assignment method called the *Robin Hood assignment*, which robs bits from intervals with lower BsRs to those with higher BsRs. For example, in Figure 1(B), by redistributing bits from the first interval to the second, we can accurately answer the first three queries. To achieve a desired performance, the challenge lies in how to achieve optimal bitmap assignment across intervals via the Robin Hood assignment method.

Proposal 3: making the best of both worlds – combining learning-based methods and prefix-based methods. The inefficiency of learning-based range encoding for large ranges prompts us to consider the suitability of different encoding methods for different key distributions. We have noticed a lack of comprehensive comparisons between the performance of these methods across various key distributions. To bridge the gap, we develop an analysis framework capable of estimating the FPR of both encoding methods

for a given key distribution. Our preliminary findings suggest that the learning-based one is more suitable for dense key distributions, while the prefix-based one is better for sparse distributions. In particular, a sparse distribution implies a small number of keys within a fixed-length range, while a dense distribution indicates a large number of keys within the same range. As shown in Figure 1(B), the first interval $[0, 20]$ represents a dense distribution, while the second interval $[1000, 1500]$ exemplifies a sparse distribution. Building upon this insight, we propose that the prefix-based and learning-based range encodings can be strategically combined for a more robust query performance. The challenge here lies in choosing the desired encoding method for each interval.

Oasis. Based on our previous insights, we propose Oasis (**O**ptim**a**l Di**s**jo**i**nt **S**egmented Learned Range Filter), a novel learned range filter incorporating space pruning and Robin Hood assignment methods. Further, we formulate the problem of optimizing Oasis' configuration and solve it by integrating a segmentation algorithm that establishes the optimal configuration while maintaining consistent performance across different query distributions. Oasis outperforms state-of-the-art filters in terms of FPR across a range of workloads, covering point and range queries, with improvements of up to 100×. Additionally, it exhibits relatively low operation latency and moderate construction time.

Oasis+. Oasis+ offers a fresh perspective by combining prefix-based and learning-based encoding methods to boost query performance. Oasis+ employs a two-layer nested loop construction method. This method effectively balances FPR performance, even when subjected to different space budget constraints, at the cost of a slightly increased construction time.

Contributions. Our contributions are as follows:

- We introduce a novel space-pruning method that strikes a balance between query performance and filter sizes. This method optimizes filter performance and accelerates query processing.
- We propose a bitmap assignment (Robin Hood assignment) approach that enhances the robustness of different dataset distributions and reduces the FPR.
- We present Oasis, a novel learned range filter incorporating space pruning and Robin Hood assignment.
- We provide a detailed analysis of Oasis. Leveraging this analysis, we introduce a segmentation algorithm that can determine the optimal configuration without query sampling.
- We introduce Oasis+, a range filter that integrates the design space of both learning-based and prefix-based filters.
- We demonstrate that Oasis and Oasis+ achieve significantly lower FPRs compared to state-of-the-art baselines across diverse workloads, consisting of both point and range queries.
- We illustrate the integration of Oasis and Oasis+ into RocksDB and showcase up to 6.2× latency improvements compared to state-of-the-art baselines.

2 OASIS

This section first presents the framework of Oasis and how it integrates the space pruning and Robin Hood assignment methods into its framework in Figure 2. Section 2.1 details the implementation of the Oasis framework, and Section 2.2 illustrates its query process.

2.1 Oasis Framework

The fundamental framework of Oasis comprises a model array and compressed bitmap, as shown in Figure 2. The model array plays a pivotal role by recording the corresponding model for each interval, contributing to both space pruning and the Robin Hood assignment strategies. Each model in the array represents a linear function, as defined in Equation 1, mapping disjoint key intervals to unique segments on the uncompressed bitmap. In parallel, the compressed bitmap efficiently utilizes compression techniques to store the positions of bits set to 1.

2.1.1 Model Array. In Figure 2, Oasis first partitions the key space into disjoint intervals, each of which has a unique linear model projecting keys to a corresponding bitmap segment. Notably, segments mapped by neighboring models are adjoining and ordered identically to the models themselves. As shown in Equation 1, each $\texttt{model}_i$ maps a given key u to a bit of the corresponding bitmap segment. The offset of this bit from the starting point of the bitmap, $\texttt{model}_i(u)$, is represented by three parameters: a scale size parameter, α_i, and interval boundaries, $\texttt{beg}_i$, and $\texttt{end}_i$.

$$\texttt{model}_i(u) = \alpha_i \cdot \frac{u - \texttt{beg}_i}{\texttt{end}_i - \texttt{beg}_i}, \quad \text{where } u \in (\texttt{beg}_i, \texttt{end}_i) \tag{1}$$

The scale size records the outcome of the Robin Hood assignment strategy. Interval boundaries, derived from the key set, define the bounds of the associated interval and support space pruning. On the other hand, the model array organizes all models in ascending order according to their interval's left boundaries, facilitating the positioning of models during querying, as depicted in the following:

$$\texttt{models} = \{\texttt{model}_1, \ldots, \texttt{model}_m\}.$$

Scale Size. The scale size is the parameter required by the Robin Hood Assignment process. It helps the model to map each interval to a unique and consecutive bitmap segment on the bitmap. Also, the scale size implicitly stores the offset of each segment. Specifically, the scale size of model i, denoted as α_i, represents the number of bits in the bitmap segment assigned to the ith model, as illustrated in Figure 2. This bitmap segment starts at position $\sum_{j=1}^{i-1} \alpha_j$ and ends at $\sum_{j=1}^{i} \alpha_j$.

Interval Boundaries. The ith interval boundaries, denoted as $\texttt{beg}_i$ and $\texttt{end}_i$, signify the left and right boundaries of the related interval, defining the domain of the corresponding model. Additionally, these parameters facilitate Oasis in implementing the space pruning method. To elaborate further, the structure of Oasis' intervals ensures that no key exists between the upper bound of the ith interval and the lower bound of the $(i+1)^{\text{th}}$ interval. Consequently, any queries falling between these two keys can be immediately pruned, allowing the compressed bitmap to omit these pruning spaces. This results in space conservation and enhanced query processing efficiency.

In summary, the linear function of the ith interval passes through points $(\texttt{beg}_i, 0)$ and $(\texttt{end}_i, \alpha_i)$, and any data u falling within this interval is mapped to a position between 0 and α_i. Therefore, the final bitmap position of the data is calculated by adding the interval's offset in the bitmap and the position from the local linear model, as presented in Equation 2,

$$\texttt{pos} = \texttt{model}_i(u) + \sum_{j=1}^{i-1} \alpha_j = \alpha_i \cdot \frac{u - \texttt{beg}_i}{\texttt{end}_i - \texttt{beg}_i} + \sum_{j=1}^{i-1} \alpha_j. \tag{2}$$

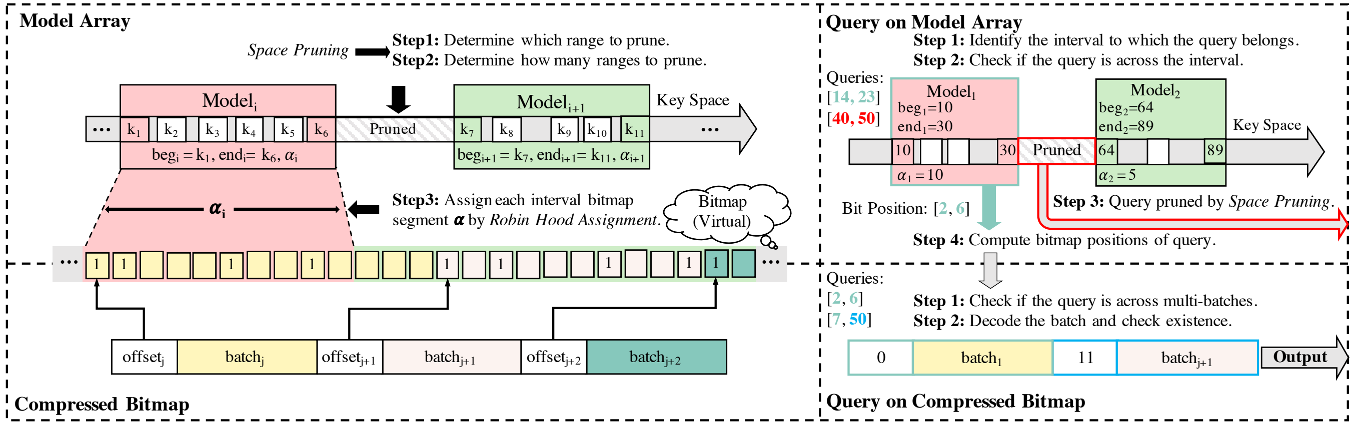

Figure 2: Oasis framework and its query process.

2.1.2 Compressed Bitmap. Inspired by SNARF, Oasis meticulously organizes β consecutive bit positions in the bitmap as batches, compressing each batch using the Elias-Fano Encoding [38, 46] technique. To store these compressed batches, Oasis employs an array named `batch_list`. Simultaneously, it utilizes another array, `batch_offset`, to record the offset of each batch from the beginning of the bitmap. The ith element in the batch_offset precisely represents the position of the first bit in the ith batch. Furthermore, since the number of keys in each batch is fixed, and their range information can be obtained from the `batch_offset`, there is no need to record the metadata of each batch. Hence, the memory overhead of this chunking compressed method is acceptable.

2.2 Query on Oasis

In this section, we delve into the range query process of Oasis. A point query can be considered a special case of a range query where the left and right boundaries are equal. Therefore, we will not separately discuss point queries. We begin by illustrating the operations for each component of Oasis separately in Section 2.2.1 and Section 2.2.2. Subsequently, we demonstrate the overall range query process of Oasis in Section 2.2.3.

2.2.1 Quering the Model Array. The querying process of the model array involves four steps, as shown in the upper right of Figure 2. Initially, the model array searches for the specific interval to which the query belongs among interval boundary parameters. Then, the process checks whether the query crosses the interval or falls within the pruned spaces to facilitate the early pruning process. Finally, the process uses Equation 2 to calculate the respective bitmap positions corresponding to the query boundaries.

As shown in Algorithm 1, the query process starts by invoking the `FindInterval` function to determine the index of the interval corresponding to the query (line 2). Subsequently, the algorithm evaluates the value of *status* to determine if further processing is necessary (line 3). A `status` value of 0 signifies that the query falls within the bounds of the interval indexed by *idx*, requiring additional processing. Utilizing the interval index *idx* and the query boundaries, the model computes the bitmap positions (line 5).

The `FindInterval` begins by initializing the `status` to 0. It then performs a binary search on each model's beg parameter to

Algorithm 1 Range Query on Model Array

Input: l, r - the left and right bound of the range query.
Output: `pos_l, pos_r` - the bitmap position of the query's left and right bound.
Output: `status` - result code, where 1 indicates the query across models, -1 indicates the query is empty, and 0 stands for an unknown result.
Function `upBound(arr, u)` - returns the first index that satisfies $u \geq \texttt{arr}_i$.

1: **procedure** QUERY(l, r)
2: `status`, $idx \leftarrow$ `FindInterval`(l, r)
3: **if** status $\neq 0$ **then**
4: **return** status, $-1, -1$
5: `pos_l, pos_r` $\leftarrow$ `getPos`(l, idx), `getPos`(r, idx)
6: **return** `status, pos_l, pos_r`
7: **procedure** FINDINTERVAL(l, r)
8: status $\leftarrow 0$
9: $i \leftarrow$ upBound$(\{\text{beg}_1, \ldots, \text{beg}_m\}, l)$
10: **if** $l >$ end$_i$ **and** $r <$ beg$_{i+1}$ **then**
11: status $\leftarrow -1$ ▷ *query falls within the empty range*
12: **if not** ($l >$ beg$_i$ **and** $r <$ end$_i$) **then**
13: status $\leftarrow 1$ ▷ *query across multiple intervals*
14: **return** status, i
15: **procedure** GETPOS(u, i)
16: **return** $\alpha_i \cdot \frac{u - \text{beg}_i}{\text{end}_i - \text{beg}_i} + \sum_{j=1}^{i-1} \alpha_j$

determine the interval in which the lower bound of the query falls. Once the interval index i is identified, the procedure evaluates whether the query falls within the space between the ith and $(i+1)^{\text{th}}$ intervals (line 10). If the conditions in line 10 are met, the procedure sets the status to -1, indicating that the current query is guaranteed to be empty. Conversely, if the query does not meet the conditions, the procedure further examines whether the query spans multiple intervals (line 12). If this condition is satisfied, the procedure immediately assigns the status to 1, indicating that the query range covers keys in the key set. The primary contributor to the query cost of the model array is the `FindInterval` function. Therefore, querying on the model array takes $O(\log m)$, where m denotes the size of the model array.

Running Example. Take the range query $[14, 23]$ as an example. As shown in Figure 2, the model array (1) first identifies that the query falls within the $\texttt{interval}_1$ by searching on the interval boundary parameters of the internal models within the array. (2) Then, it invokes the corresponding $\texttt{model}_1$, utilizing Equation 2 to calculate the bitmap positions of the query's left boundary,

$$\texttt{pos_l} = \alpha_1 \frac{14 - \text{beg}_1}{\text{end}_1 - \text{beg}_1} + \sum_{i=1}^{0} \alpha_i = 10 \cdot \frac{14 - 10}{30 - 10} = 2,$$

Algorithm 2 Range Query on Compressed Bitmap

```
Input: pos_l, pos_r - left and right bitmap positions of the query.
Output: exist - the presence of data within the range query.
Function batchQuery(batch, l, r) - returns whether there is element between l
and r for the given batch.
1: procedure Query(pos_l, pos_r)
2:     if batch_offset.front > pos_r or batch_offset.back < pos_l then
3:         return false                          ▷ query out of range
4:     idx ← upBound(batch_offset, pos_l)
5:     if batch_offset[idx + 1] <= pos_r then
6:         return true          ▷ query across multiple compacted batches
7:     offset_l ← pos_l − batch_offset[idx]
8:     offset_r ← pos_r − batch_offset[idx]
9:     return batchQuery(batch_lists[idx], offset_l, offset_r)
```

and right boundary, $\texttt{pos_r} = 10 \cdot (23 - 10)/(30 - 10) = 6$.

2.2.2 Querying the Compressed Bitmap. The query process of the compressed bitmap is detailed in Algorithm 2. The procedure begins by checking whether the query falls outside the bitmap's range (lines 2-3). It immediately returns a negative result if the query is not within the bitmap. Otherwise, the procedure employs a binary search on the batch_offset to identify the batch to which the left boundary of the query belongs (line 4). Once the procedure determines the batch index of the query's left boundary, it checks whether the right boundary of the query falls within the current batch (line 5). A positive outcome of the condition in line 5 signifies that the query spans multiple batches, confirming the presence of data within the queried range. Consequently, the procedure directly returns a positive answer. Conversely, if the condition in line 5 is unmet, the procedure will calculate the offset for both boundaries (lines 7-8) and then decompress the batch corresponding to the query and check for a set bit within the specified boundaries. Notably, the decoding time of a batch takes around $O(\beta)$, where β denotes the number of bit positions in a batch of the compressed bitmap, as mentioned in Section 2.1.2. Hence, the overall time complexity is $O(\beta + \log N/\beta)$. Here, N is the size of the key set, and N/β denotes the number of compressed batches.

2.2.3 Querying the Oasis. The execution process for a range query in Oasis is detailed in Algorithm 3. Initially, Oasis utilizes the model array component to determine bit positions corresponding to the query's left and right boundaries (line 2). The procedure checks the return status of the model array to determine whether further processing is required. If the return status is not equal to 0, the algorithm immediately provides the result based on the status value (lines 3-4). Recalling that a value of 1 denotes a non-empty query, while −1 indicates an empty query. In cases where the model array cannot perform early pruning, Oasis uses the compressed bitmap to determine whether data exists within the range defined by the model array, which finally determines the query's membership status (line 5). Clearly, Oasis' query time is the sum of the time complexity of Algorithm 1 and Algorithm 2, which is $O(\beta + \log N/\beta + \log m)$. Further, the pruning strategies in these processes can reduce the practical query time.

3 FPR OPTIMIZATION FOR OASIS

In Section 3.1, we first analyze the FPR of Oasis and frame the problem of minimizing FPR as an optimization problem. Then, we provide a general solution for this optimization but with a high time complexity. To address this, in Section 3.2, we introduce a

Algorithm 3 Range Query on Oasis

```
Input: l, r - the left and right bound of the range query.
Input: models - the model array of Oasis.
Input: comp_bitmap - the compressed bitmap of Oasis.
1: procedure RangeQuery(l, r, models)
2:     status, pos_l, pos_r ← models.Query(l, r)
3:     if status ≠ 0 then
4:         return status = 1
5:     return comp_bitmap.Query(pos_l, pos_r)
```

segmentation algorithm that leverages the characteristics of the optimization function to search for a near-optimal configuration for Oasis.

3.1 FPR Analysis

We begin by examining a sorted key set $\mathbf{D} = \{k_1, \ldots, k_N\}$. The following terms and notations will be used in this subsection:

- p_i - Probability of an empty query falling within the ith interval.
- x - Denotes the correlation distance of an empty query. Formally, for an empty query $[l, r]$, the adjacent keys to its left and right boundaries are $k_i,\ k_{i+1}$, where $k_i < l < r < k_{i+1}$. The correlation distance of this query is defined as $\min(l - k_i,\ k_{i+1} - r)$.
- $F(x \le X)$ - The cumulative percentage of empty queries with a correlation distance $x \le X$.
- m - Denotes the number of intervals in Oasis.
- ρ_i - The BitSpan Resolution of the ith interval.
- $\mathbf{O}_E = \{o_1, \ldots, o_m\}$ is used to indicate the subscripts of each interval's right boundary. For example, an interval i in the Oasis can be represented as $[k_{o_{i-1}+1},\ k_{o_i}]$.
- $\mathbf{A} = \{\alpha_1, \ldots, \alpha_m\}$, where α_i denotes the range size of the corresponding bitmap segment for the ith interval.

Analyze the FPR of Oasis. Recalling the query process of Oasis, it is evident that false positive occurrences are confined within the interval and do not extend across multiple intervals, as actual keys bound the intervals, and hence, a query range that goes across intervals will only lead to a true positive. Owing to the monotonicity of linear models, false positive answers can exclusively arise at the mapping positions of the query boundaries. This is the case when at least one of these mapping positions is set to 1 by the filter. Hence, unlike the hash-based prefix filter, Oasis' FPR remains unaffected by the length of the query. Put all these together, we can formulate the FPR of Oasis in terms of workload distribution and the BsR, as demonstrated in Equation 3.

$$FPR = F\left(x \le \sum_{i=1}^{m} p_i \rho_i\right). \tag{3}$$

However, in many cases, we cannot obtain the workload distribution; BsR becomes the only factor that affects the FPR. Since Oasis uses a linear model for each interval, the BsR can be determined by dividing the interval's range by its corresponding bitmap mapping range as we did in Section 1. Specifically, for an interval i in an Oasis instance assigned with a bitmap segment length of α_i, the BsR of the ith interval is defined by Equation 4.

$$\rho_i = \frac{k_{o_i} - k_{o_{i-1}+1}}{\alpha_i}. \tag{4}$$

Therefore, the FPR of the Oasis can be formalized as follows:

$$FPR = F\left(x \le \sum_{i=1}^{m} p_i \cdot \frac{k_{o_i} - k_{o_{i-1}+1}}{\alpha_i}\right). \tag{5}$$

The optimization problem. Drawing on our prior Oasis analysis, optimizing its FPR is equivalent to minimizing the expected value of the overall BsR. This equivalence is due to the monotonicity of the cumulative distribution function $F(x \le X)$. Therefore, within a given space requirement, the optimal filter configuration can be formulated as an optimization problem:

$$\begin{aligned} \arg\min_{m,\mathbf{O}_E,\mathbf{A}} \quad & \sum_{i=1}^{m} p_i \cdot \frac{k_{o_i} - k_{o_{i-1}+1}}{\alpha_i} \\ \text{s.t.} \quad & m \cdot \texttt{kCost} + \tau \cdot \sum_{i=1}^{m} \alpha_i \le \texttt{kMemBudget}. \end{aligned}$$

In this context, kCost represents the memory overhead associated with the stored interval metadata. Specifically, in Oasis, we employ two 64-bit integers to represent the boundaries of an interval, along with an additional 64-bit integer to record the length of the bitmap segment corresponding to the interval. Here, τ denotes the compression ratio associated with the bitmap's compression algorithm, specifically using Elias-Fano encoding as described in [38]. On the other hand, kMemBudget signifies the allocated memory for the entire filter, which equals the product of the key numbers and the bits-per-key assigned to the filter.

Optimize FPR. When the workload distribution and p_i are known, we consider a scenario where the optimized key set segmentation has been established, but the bitmap's mapping assignment remains unresolved. In this case, we are left with the task of determining the optimal $\mathbf{A}$ to minimize the FPR, which can be restated as follows.

$$\begin{aligned} \arg\min_{\mathbf{A}} \quad & \sum_{i=1}^{m} p_i \cdot \frac{k_{o_i} - k_{o_{i-1}+1}}{\alpha_i} \\ \text{s.t.} \quad & \sum_{i=1}^{m} \alpha_i \le \texttt{kMemBudget}', \end{aligned}$$

where $\texttt{kMemBudget}' = (\texttt{kMemBudget} - m \cdot \texttt{kCost})/\tau$ represents a constant value.

By applying the Cauchy–Schwarz inequality [6][3], the lower bound of the objective function is established as follows:

$$\begin{aligned} & \left[\sum_{i=1}^{m}\left(\sqrt{p_i \frac{k_{o_i} - k_{o_{i-1}+1}}{\alpha_i}}\right)^2\right]\left[\sum_{i=1}^{m}\left(\sqrt{\alpha_i}\right)^2\right] \ge \left(\sum_{i=1}^{m}\sqrt{p_i \frac{k_{o_i} - k_{o_{i-1}+1}}{\alpha_i}} \cdot \sqrt{\alpha_i}\right)^2 \\ \Longrightarrow & \sum_{i=1}^{m} p_i \cdot \frac{k_{o_i} - k_{o_{i-1}+1}}{\alpha_i} \ge \frac{\left(\sum_{i=1}^{m}\sqrt{p_i\left(k_{o_i} - k_{o_{i-1}+1}\right)}\right)^2}{\texttt{kMemBudget}'}. \end{aligned} \tag{6}$$

When equality holds, we attain the minimum value of the objective function. Applying Cauchy–Schwarz inequality, the optimal configuration $\mathbf{A}^{\mathsf{OPT}} = \{\alpha_1^{\mathsf{OPT}}, \ldots, \alpha_m^{\mathsf{OPT}}\}$ can be computed in $O(m)$ time using the following equation:

$$\frac{\sqrt{p_i\left(k_{o_i} - k_{o_{i-1}+1}\right)}}{\alpha_i^{\mathsf{OPT}}} = \frac{\sum_{l=1}^{m}\sqrt{p_l\left(k_{o_l} - k_{o_{l-1}+1}\right)}}{\texttt{kMemBudget}'}. \tag{7}$$

Then, plug $\mathbf{A}^{\mathsf{OPT}}$ into the initial optimization problem mentioned earlier simplifies the problem to:

$$\begin{aligned} \arg\min_{m,\mathbf{O}_E} \quad & \frac{\tau \cdot \left(\sum_{i=1}^{m}\sqrt{p_i \cdot \left(k_{o_i} - k_{o_{i-1}+1}\right)}\right)^2}{\texttt{kMemBudget} - m \cdot kCost} \\ \text{s.t.} \quad & \texttt{kMemBudget} - m \cdot \texttt{kCost} \ge 0. \end{aligned} \tag{8}$$

[3] $\left(\sum_{i=1}^{n} x_i y_i\right)^2 \le \left(\sum_{i=1}^{n} x_i^2\right)\left(\sum_{i=1}^{n} y_i^2\right)$, For $x_i, y_i > 0$, the equality holds when $x_1/y_1 = x_2/y_2 = \cdots = x_n/y_n$.

In order to find the optimal segmentation for a fixed number of intervals m, we can iterate through all $\binom{N-1}{m}$ possible combinations. This gives us the final optimal solution for $\mathbf{O}_E$. To address the optimization problem, we proceed as follows: for each value of m, we identify the optimal segmentation method and compute the corresponding upper bound of the FPR using Equation 6. We then select the value of m that yields the smallest upper bound for the FPR, along with its corresponding segmentation method, as the final optimal solution for $\mathbf{O}_E$. Finally, we calculate the optimal bitmap segment setting for each interval using Equation 7, obtaining the optimal solution for $\mathbf{A}$.

3.2 Segmentation Algorithm

In Section 3.1, a general solution is presented for obtaining the optimal configuration of the filter under any scenario in a time complexity of $O(\sum_{m=1}^{M}\binom{N-1}{m})$, where $M = \texttt{kMemBudget}/\texttt{kCost}$ represents the maximum number of intervals. However, this time complexity is too high for practical use. Next, we develop an algorithm capable of providing a near-optimal solution within a reasonable time complexity is essential.

Let us consider a scenario where the workload follows a uniform distribution, meaning that the left boundary of the query occurs with equal probability at any position within the key space. Thus, the probability of a query falling within an interval is proportional to the interval's range size (i.e., $p_i \propto (k_{o_i} - k_{o_{i-1}+1})$). In this uniform workload scenario, we can adapt the optimization problem defined in Equation 8 to:

$$\begin{aligned} \arg\min_{m,\mathbf{O}_E} \quad & \frac{\tau \cdot \left[\sum_{i=1}^{m}\sqrt{\left(k_{o_i} - k_{o_{i-1}+1}\right)^2}\right]^2}{\texttt{kMemBudget} - m \cdot \texttt{kCost}} \\ = \; & \frac{\tau \cdot \left[\sum_{i=1}^{m}\left(k_{o_i} - k_{o_{i-1}+1}\right)\right]^2}{\texttt{kMemBudget} - m \cdot \texttt{kCost}} \\ \text{s.t.} \quad & \texttt{kMemBudget} - m \cdot \texttt{kCost} \ge 0. \end{aligned} \tag{9}$$

From Equation 9, the objective function is inversely related to the cumulative sum of lengths of empty ranges between adjacent intervals. In other words, with a uniform workload assumption, a larger cumulative sum corresponds to a lower FPR. This aligns with Oasis' original intent to eliminate large empty ranges between adjacent keys. Consequently, under a uniform workload and fixed interval count (m is fixed), optimal segmentation is achieved by selecting the top-m distances between neighboring keys. These m key pairs then define interval boundaries, allowing estimation of the lower bound for the optimization problem's objective function as follows:

$$\frac{\tau \cdot \left(k_N - k_1 - \sum^{\text{top-}m}\delta\right)^2}{\texttt{kMemBudget} - m \cdot \texttt{kCost}}, \tag{10}$$

where δ represents the distance between adjoined keys within $\mathbf{D}$.

To find the optimal value of m, we iterate through all possible values (from M to 0) and select the one that minimizes Equation 10. Having identified the optimal m, we set m^{th} largest adjacent key distances as the threshold. Neighboring keys with distances exceeding this threshold are designated as boundaries between adjacent intervals and directly recorded, yielding the optimal segmentation. After finding the optimal segmentation, the optimal bitmap segment assignment of the uniform workload is as follows:

$$\alpha_i = \frac{k_{o_i} - k_{o_{i-1}+1}}{k_N - k_1 - \sum^{\text{top-}m}\delta} \cdot \left(\texttt{kMemBudget} - m \cdot \texttt{kCost}\right)/\tau. \tag{11}$$

Algorithm 4 Segmentation

Input: $\mathbf{D}$ - the sorted key set, where $\mathbf{D} = \{k_1, \ldots, k_N\}$.
Input: `kMemBudget` - the memory budget for the entire filter.
Input: `kCost` - the memory overhead associated with storing interval metadata.
Function: `TopDst`(m, $\mathbf{D}$) - returns top-m adjacent key distances in acending order.
Output: `models` - the model array of Oasis.

```
1: procedure SEGMENTATION(D,kMemBudget, kCost)
2:     M ← kMemBudget / kCost
3:     min_heap ← TopDst(M, D)
4:     delta_sum ← k_N − k_1 − ∑ min_heap
5:     mem_budget ← kMemBudget − kCost · M
6:     threshold, best_e ← min_heap.top, delta_sum^2/mem_budget
7:     while not min_heap.Empty() do                          ▷ solve m
8:         delta_sum ← delta_sum + min_heap.top
9:         mem_budget ← mem_budget + kCost
10:        cur_e ← delta_sum^2 / mem_budget
11:        if cur_e < best_e then
12:            threshold, best_e ← min_heap.top, cur_e
13:        min_heap.Pop()
14:    delta_sum ← k_N − k_1
15:    models.begins.Append(k_1)                              ▷ solve O_E
16:    for i ∈ {1, . . . , N − 1} do
17:        if k_{i+1} − k_i ≥ threshold then
18:            models.begins.Append(k_{i+1})
19:            models.ends.Append(k_i)
20:            delta_sum ← delta_sum − (k_{i+1} − k_i)
21:    models.ends.Append(k_N)
22:    b_array_range ← (1/τ) (kMemBudget − kCost · models.ends.size)
23:    for each interval [begin, end] do                      ▷ solve A
24:        α ← (end−begin)/delta_sum · b_array_range
25:        models.size_array.Append(α)
26:    return models
```

Algorithm 4 outlines our method. It employs a min-heap to maintain the top-M neighboring key distances as threshold candidates (lines 2-3), where M denotes the maximum number of intervals the model can store. The algorithm then identifies the optimal threshold through the following steps: it iterates through the candidate set in descending order (lines 4-13), calculating the corresponding lower bound for the FPR under each candidate threshold using Equation 10. The candidate that yields the lowest estimated FPR is selected as the optimal threshold. Once the optimal threshold is determined, the algorithm segments the keys into multiple intervals (lines 15-21). Neighboring keys whose distance exceeds the threshold are designated as boundaries between adjacent intervals. Finally, the algorithm computes the optimal bitmap range assignment (lines 22-25). It iterates through all intervals while utilizing Equation 11. The overall time complexity of our algorithm is $O(N \log M)$.

4 OASIS+

As mentioned earlier, learning-based filters show efficiency in encoding dense key distributions, while prefix-based filters excel with sparse ones. Motivated by this, we introduce Oasis+, a novel range filter that integrates both the learning- and prefix-based methods, offering a solution that can adapt to a broader range of scenarios.

4.1 Oasis+ Framework

The key insight of Oasis+ is replacing learning-based encoding methods for sparse intervals with prefix-based ones. Specifically, Oasis+ utilizes Oasis to encode dense intervals and Proteus to encode sparse intervals. As shown in Figure 3, the Oasis+ framework closely resembles Oasis, comprising both a bitmap and a model array, but it differs in several aspects, described as follows.

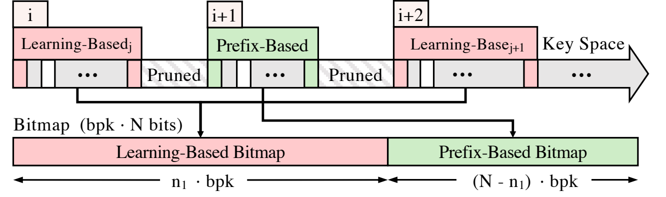

Figure 3: Framework of Oasis+.

4.1.1 Bitmap. The bitmap structure in Oasis+ accommodates distinct representations for learning- and prefix-based filters. It achieves this by dividing the bitmap into two adjoining segments, allocating space budgets based on the number of keys each filter type stores. For example, if the learning-based filter encodes n_1 keys, and the prefix-based filter encodes n_2 keys, Oasis+ allocates segments based on the ratio n_1/n_2, with one segment assigned to the learning-based filter and the other to the prefix-based filter.

4.1.2 Model Array. Like Oasis, Oasis+ divides the key space into disjoint intervals, storing their left and right boundaries. To identify the filter type of each interval, Oasis+ introduces an extra filter indicator. Further, the scale size parameter is exclusive to intervals encoded by the learning-based filter.

Filter Indicator. The filter indicator is a bitmap that records the filter type for each interval. The ith bit corresponds to the ith interval, with a set bit indicating the use of the learning-based filter and an unset bit indicating the prefix-based filter.

In the encoding process of the Oasis model (as per Equation 2), calculating the bitmap segment offset for a specific interval requires accumulating the scale size of previous intervals in the model array. Given the different filter types in Oasis+, Equation 2 needs slight modification. The bit position u within the ith interval, representing the jth interval encoded by Oasis, can be expressed as follows:

$$\text{pos} = \alpha_j \cdot \frac{u - \text{beg}[i]}{\text{end}[i] - \text{beg}[i]} + \sum_{l=1}^{j-1} \alpha_l. \tag{12}$$

4.2 Range Query on Oasis+

Algorithm 5 outlines the operation of Oasis+. It first uses the `FindInterval` function to determine the query's interval index (line 2). Then it evaluates the need for further query steps based on the value of `status` (lines 3-4). If further processing is required, it checks the filter indicator to decide the interval's filter type (line 5). When the current interval i is encoded by Proteus, the query is directly sent to Proteus, and its outcome is returned (line 6). However, if Oasis encodes the current interval, Oasis+ calculates the bitmap positions of the query's boundaries using Equation 12

Algorithm 5 Oasis+ Range Query

Input: l, r - the left and right bound of the range query.
Input: `models` - the model array of Oasis+.
Input: `bitmap` - the bitmap of Oasis+.

```
1: procedure RANGEQUERY(l, r, models, bitArray)
2:     status, i ← models.FindInterval(l, r)
3:     if status ≠ 0 then
4:         return status = 1
5:     if models.filter_indicator[i] = 0 then          ▷ encoded by Proteus
6:         return bitmap.Proteus.Query(l, r)
7:     else                                             ▷ encoded by Oasis
8:         pos_l, pos_r ← models.getPos(l, i), models.getPos(r, i)
9:         return bitmap.Ranger.comp_bitmap.Query(pos_l, pos_r)
```

(line 8). Finally, the query is addressed by applying these bitmap positions to Oasis' compressed bitmap (line 9).

4.3 Constructing Oasis+

Like Oasis, the construction strategy decides the upper bound of Oasis+'s performance. However, there is a crucial difference from Oasis' segmentation strategy outlined in Section 3.2. In Oasis, the goal is to find the optimal interval settings, whereas, in the construction of Oasis+, we go a step further by searching the optimal filter type for each interval. Here, we follow the uniform workload assumption mentioned in Section 3.2 and further employ the Oasis' segmentation strategy to build up the intervals of Oasis+.

Here, we first present methods for estimating the FPR of both filters (Oasis and Proteus) without query sampling. Subsequently, we use these methods to establish our construction algorithm. Finally, we prove that an Oasis+ constructed by our algorithm will be at least as good as the best-performing filter between Oasis and Proteus when used individually.

4.3.1 FPR Estimation without Sampling. Consider a scenario where Oasis+ has established its intervals. The remaining space budget for each filter is given by $\texttt{bpk}' = \texttt{bpk} - m \cdot \texttt{kCost}'/N$. Here, $\texttt{kCost}'$ denotes the memory overhead of interval metadata in Oasis+, which consists of two 64-bit unsigned integers for interval boundaries and a 1-bit filter type indicator denoted as $\mathbb{I}_t(i)$ for brevity.

Simulate Query Correlation Distance Distribution. The distribution of query correlation distances can be simulated using key space distribution information. Since empty queries are confined to the intervals between adjacent sorted keys, the correlation distance x within a range of δ must satisfy $2x \leq \delta$. Assuming a uniform workload distribution, the expected correlation distance for a query within a range of length δ is $\delta/4$. Hence, we can simulate the distribution of query correlation distances using the cumulative distribution function (CDF) of adjoined key distances as follows:

$$F'(x \leq X) = \frac{\sum_{i=1}^{N-1} \mathbb{I}_{\delta_i \leq 4X}}{N-1}, \tag{13}$$

where $F'(\cdot)$ denotes the estimated CDF of query correlation distances; $\mathbb{I}_{\text{cond}}$ is the indicator function that equals 1 when its cond is met and 0 otherwise.

Estimate the FPR of Oasis. Since Oasis+ assumes a uniform workload during construction, we can use the analysis in Section 3.2 to estimate the FPR of Oasis portion using the following equation:

$$\begin{aligned} FPR_{LRF} &= F'\left(x \leq \frac{\sum_{i=1}^{m} \mathbb{I}_t(i) \cdot (k_{o_i} - k_{o_{i-1}+1})}{\texttt{BitmapSegmentLen}}\right) \\ &= F'\left(x \leq \tau \cdot \sum_{i=1}^{m} \mathbb{I}_t(i) \cdot \frac{k_{o_i} - k_{o_{i-1}+1}}{\texttt{bpk}' \cdot (o_i - o_{i-1}) - \texttt{kMeta}}\right), \end{aligned} \tag{14}$$

where *kMeta* denotes the memory overhead for the parameter of each Oasis' model, specifically, one 64-bit unsigned integer to record the scale size of the model.

Estimating the FPR of Proteus. The Contextual Prefix FPR (CPFPR) model uses sampled queries to estimate FPR and determine the optimal Proteus configuration. In this FPR estimation process, CPFPR considers two key workload attributes: query correlation distance and query length. However, Oasis+'s construction lacks sampled queries. To address this, we adapt CPFPR FPR estimation process.

Algorithm 6 Oasis+ Construction Algorithm

```
    Input: min_heap - top-M adjacent key distances
    Input: max_q_len - the maximum length of the query.
    Input: D = {k_1, ..., k_N} - sorted key set.
    Function: BuildArr(t, D) - returns model array and sum of their ranges.
    Function: WorstIdx(indicator, models) - returns sparsest Oasis interval idx.
    Function: LRFEst(bpk, models, indicator) - returns the FPR of Oasis.
 1: procedure CONSTRUCT(min_heap, max_q_len, D)
 2:   min_fpr ← Max
 3:   best_configs ← ∅
 4:   while not min_heap.Empty() do
 5:     models, delta_sum ← BuildArr(min_heap.top, D)
 6:     bpk' ← bpk − models.size × kCost'/N
 7:     indicator ← all 1 bit map of size model.size
 8:     p_config, p_fpr ← CPFPR_w/o(bpk', max_q_len, F_cor)
 9:     cur_fpr ← LRFEst(bpk', models, indicator)
10:     for i ← WorstIdx(indicator, models) do
11:       indicator[i] ← 0
12:       lrf_fpr ← LRFEst(bpk', mdoels, indicator)
13:       delta_sum ← delta_sum − (models.ends_i − index.begins_i)
14:       fpr ← lrf_fpr · delta_sum + p_fpr · (sum − delta_sum)
15:       if fpr < cur_fpr then
16:         cur_fpr = fpr
17:       else
18:         indicator[i] ← 1
19:         break
20:     if cur_fpr < best_fpr then
21:       best_fpr = cur_fpr
22:       best_configs = {models, indicator, p_config}
23:     min_heap.Pop()
24:   Oasis ← BuildOasis(best_configs, D)
25:   Proteus ← BuildProteus(best_configs, p_config, D)
```

Firstly, we assume a uniform query length distribution with a user-definable maximum length. Subsequently, we integrate it with the aforementioned distribution function F' to form the modified CPFPR model and then estimate Proteus' FPR as follows:

$$FPR_{Pro} = \texttt{CPFPR}_{w/o}(\texttt{bpk}, \texttt{max_q_len}, F'), \tag{15}$$

where $\texttt{CPFPR}_{w/o}$ represents the no-sampling versioned of CPFPR.

Estimate overall FPR. Under the uniform query distribution assumption, the overall FPR is calculated as the sum of the estimated FPR for each filter, weighted by the probability of empty queries falling within their respective intervals, given by:

$$\sum_{i=1}^{m} \frac{k_{o_i} - k_{o_{i-1}+1}}{k_N - k_1} \left[\mathbb{I}_t(i) \cdot FPR_{LRF} + (1 - \mathbb{I}_t(i)) \cdot FPR_{Pro}\right]. \tag{16}$$

4.3.2 Oasis+ Construction Algorithm. Algorithm 6 shows the construction process of Oasis+, which involves a two-layer nested loop. The outer loop follows the Oasis segmentation algorithm, which traverses all possible optimal interval settings of the Oasis. The inner loop, however, requires a strategy to capture the best interval-type assignment for a given segmentation.

As discussed in Section 1, the learning-based range encoding is more suitable for dense key distribution intervals, while the prefix-based one is better for sparse intervals. Following this, we propose a greedy algorithm leveraging the above estimation methods to identify the best interval-type assignment for Oasis+. Specifically, the greedy algorithm first assumes that all intervals are encoded by the learning-based method (line 7). It then iterates through all intervals from the sparsest to the densest (line 10). In each iteration, the algorithm tries to replace the encoding method for the current interval if it can reduce the overall FPR (lines 11-15). Otherwise, it reverts to the previous interval configuration (line 18). By iteratively attempting such replacements, the algorithm aims to reach the lowest overall FPR through localized optimizations.

In conclusion, the construction algorithm initially iterates over the top-M adjoining key distances in the outer to establish intervals (line 5). For each iteration, the algorithm employs the aforementioned greedy strategy to determine the best interval-type assignment and corresponding FPR (lines 5-22). If the current FPR is lower than all previous ones, the algorithm updates the optimal configuration (lines 20-22). Finally, after exiting the nested loop, the construction algorithm proceeds to build the entire filter with the optimal configuration (lines 24-25). Apparently, the time complexity of Algorithm 6 in the worst case is $O(M^2)$.

4.3.3 Proof of Robustness. Algorithm 6 ensures that Oasis+'s performance matches or exceeds that of the better-performing filter between Oasis and Proteus (without sampling). Firstly, the algorithm's outer loop is based on Oasis ' segmentation strategy and converges to Oasis' optimal settings. While the last iteration treats the entire key set as a single interval, aligning the space budget input with the overall filter's budget. Hence, it also optimizes Proteus' configuration. The algorithm consistently selects the best-estimated configuration, ensuring that Oasis+'s performance always matches or exceeds that of the included filter types.

5 SUPPORT IN-PLACE UPDATE

Given the disjoint nature of our filters' intervals, we approach updates within and between intervals distinctly.

Updates within Intervals. For updates within the interval, Oasis applies [47] solution, involving (1) locating the bitmap position of the key to insert/delete and (2) subsequently adding/removing it from the batch. Meanwhile, Oasis+ employs the corresponding update mechanism based on the associated encoding method.

Deletion of Boundaries. When deletions occur at interval boundaries, both Oasis and Oasis+ designate the nearest key to the deletion key as the new boundary. If no key remains within the current interval after deletion, the entire interval will be removed.

Insertions between Intervals. For the insertion between intervals, our filters try to merge it into the closest interval by setting the insertion key as the new boundary of that interval and inserting the old boundary into the bitmap. When the insertion occurs outside the range of the current filter index, we update the filter's boundary with the new key and insert the old boundary into the bitmap.

6 STANDALONE EVALUATION

In this section, we evaluate Oasis and Oasis+ as standalone filters, comparing them to state-of-the-art range filters. The results show that Oasis consistently outperforms existing filters across various scenarios, while Oasis+ exhibits robust and competitive performance, often matching or surpassing the best-performing filter in all tested situations. For all our experiments in both Section 6 and Section 7, we use 64-bit unsigned integers as the data type.

6.1 Datasets and Workloads

Datasets. We selected real-world datasets from the Search on Sorted Data (SOSD) benchmark [34] and followed the experimental settings of Proteus [21] for synthetic datasets. Each dataset comprises 10M 64-bit unsigned integer data points as filter keys.

- **Unif:** Keys generated uniformly from the range $[0, 2^{64} - 1]$.
- **Norm:** Keys generated from normal distribution with a mean of 2^{63} and a standard deviation of 0.01×2^{64}.
- **amzn:** 800M Amazon book popularity data ranging in $[0, 2^{63}]$.
- **face:** 200M unsampled Facebook user IDs ranging in $[0, 2^{64} - 1]$.

Workloads. In our experiments, queries are formulated in the following format: $[\texttt{left}, \texttt{left} + \texttt{offset})$. The `offset` is randomly selected from a uniform distribution within the range $(0, \texttt{max_range}]$. The `offset` is set to 0 for the point query. We generate 1M queries for each workload. The workloads used include:

- **Unif:** `left` is chosen uniformly from $[0, 2^{64} - \texttt{max_range}]$.
- **Real:** For real-world datasets, we randomly select 10M data points as keys, with an additional 1M data points chosen as the left boundaries for the queries.
- **Correlated:** We randomly pick 1M keys from the key set and generate the `left` by uniformly choosing from $[\texttt{key} + 1, \texttt{key} + \texttt{cor}]$, where `cor` is set to a fixed value of 2^{10}.

Baselines. We evaluate Oasis and Oasis+ against state-of-the-art prefix-based range filters. For REncoder[4], we use its enhanced version, REncoderSE, with default settings as the baseline. SNARF[5] and bloomRF[6] are configured with the default settings of their codebase. Proteus[7], Rosetta, and REncoder all employ a 0.02 query sample proportion. We iteratively adjust the real-suffix bit length in SuRF[8] to determine the appropriate setting.

Setup. All the experiments were conducted on a machine with Intel(R) Xeon(R) Gold 5215 CPU @ 2.50GHz, 62GB RAM.

6.2 Experiments Analysis

6.2.1 False Positive Rate. We analyze Oasis and Oasis+'s FPR compared to state-of-the-art range filters across diverse datasets and workloads (Figure 4). Each row in the figure corresponds to a dataset with different workloads, while each column represents workload categories: point queries, short range queries ($[2, 32]$), long range queries ($[2, 512]$), and a combination of very long range queries ($[2, 1024]$) with point queries.

Analyzing evaluation results of SOTA filters. Figure 4 shows SuRF's efficient handling of long range queries. However, suffix truncation in SuRF results in higher FPR for short range and point queries, dismissing crucial information for range identification. Rosetta addresses short query deficiencies by storing prefix lengths in separate Bloom filters, with an optimization strategy favoring longer prefixes, performing well in short range and point query workloads. Nevertheless, this strategy increases FPR in long range queries. REncoder and bloomRF address this limitation by incorporating suffix information into short-prefix Bloom filter bitmaps. However, in point query workloads, where a full-length Bloom filter is most efficient, REncoder and bloomRF exhibit higher FPR than Rosetta. Proteus, leveraging the CPFPR model, excels in identifying the optimal design space for prefix-based filters, surpassing or at least matching the performance of all such filters in every cases. SNARF efficiently encodes approximate uniform datasets, making it well-suited for uniform workloads (`face-real` and `Unif-Unif`). Its

[4]https://github.com/Range-Filter/REncoder
[5]https://github.com/kapilvaidya24/SNARF
[6]https://github.com/awitten1/bloomRF
[7]https://github.com/Eric-R-Knorr/Proteus
[8]https://github.com/efficient/SuRF

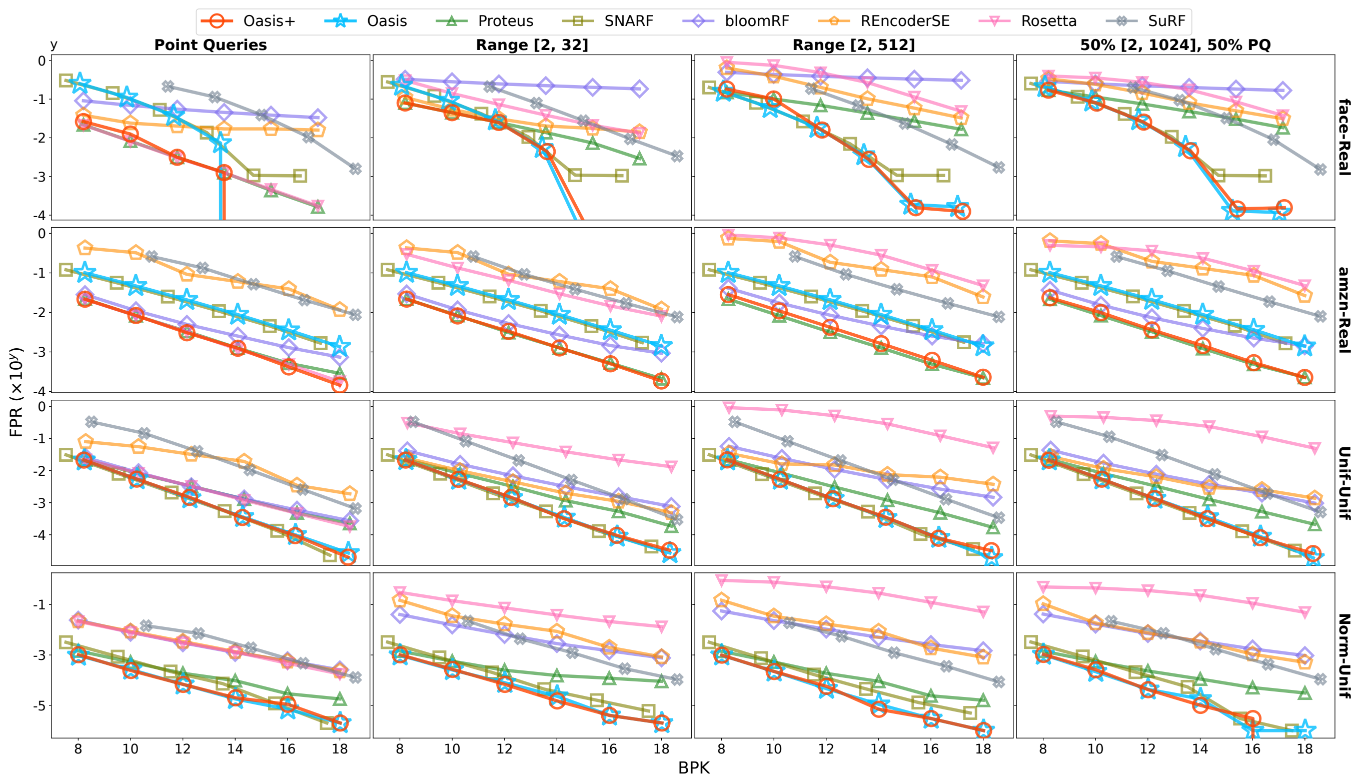

Figure 4: Filters FPR-space budgets (BPK) under different datasets and workloads.

uniform range encoding strategy performs exceptionally well with normalized distributed datasets, as seen in Figure 4, where SNARF demonstrates significantly lower FPR than Proteus in uniform and normal dataset-workload use cases.

Oasis' segmentation algorithm always generates the optimal setting for uniform workloads. For almost all uniform workloads, Oasis outperforms existing range filters. Figure 4, while SNARF excels in `Unif-Unif` and `Norm-Unif` cases, Oasis achieves a better FPR (improved by factors of 1.5× and 3.2× on average). This discrepancy for uniform workloads arises because a single linear model effectively captures the distribution of the uniformly distributed dataset. Additionally, in the normal workload's dense interval, one linear model suffices to distinguish true negative queries. The excessive number of models in SNARF leads to space wastage and increased overall BsR. Consequently, the fixed model number approach limits SNARF's performance. In contrast, Oasis, aided by its segmentation algorithm, reduces the number of models for these datasets, leveraging saved space budget from model metadata to enhance accuracy, contributing to superior FPR performance.

The `face` dataset, following a roughly uniform distribution, benefits SNARF with excellent performance and lower FPR when BPK exceeds 12. However, unlike `Unif` datasets, occasional bursts between adjoining sorted keys impact SNARF's FPR due to its adhesive intervals structure. In contrast, Oasis ' segmentation algorithm employs space pruning to efficiently exclude large empty ranges from the bitmap. Further, it selects an appropriate pruning number (e.g., 1063 intervals compared to SNARF's 1000000 intervals), reducing metadata overhead. Consequently, with an increased space budget, Oasis achieves a rapidly dropping overall BsR, reaching an FPR as low as 1.3×10^{-5} or even 0 when BPK exceeds 14.

While Proteus outperforms Oasis on the `face` dataset with a limited space budget under a short query workload, Oasis eventually surpasses Proteus as the query range increases under every space budget. This is because, although Proteus' inner SuRF can prune the query range to some extent, it still needs to query its additional prefix Bloom filter multiple times for a long range query. This multi-stage query process diminishes its performance. In contrast, Oasis operates only twice for any query range and remains unaffected by variations in the query range, demonstrating superior performance compared to Proteus and other filters as the query range expands.

In Figure 5(A), FPR improvement of Oasis over SNARF is illustrated for short range query workloads. Here, *improvement* is defined as the percentage reduction in Oasis' FPR relative to SNARF, calculated with respect to SNARF's FPR. As shown in Figure 5(A), Oasis consistently achieves at least a 15.7% improvement over SNARF, even on non-uniformly distributed workloads. Further, given that Oasis is theoretically optimal under uniformly distributed workloads, it outperforms SNARF with an average improvement of 50.5% for approximately uniform distributed workloads.

Oasis' Space Pruning Enhances Memory Efficiency. Although Oasis stores an additional key per model compared to SNARF, removing large empty ranges from the bitmap enables Oasis to encode the key space more efficiently with fewer models. Table 1 depicts the memory savings achieved through Oasis' space pruning strategy compared to SNARF, which is calculated by subtracting SNARF's compressed bitmap size from Oasis' under equivalent memory budgets and dataset constraints and then dividing the result by the

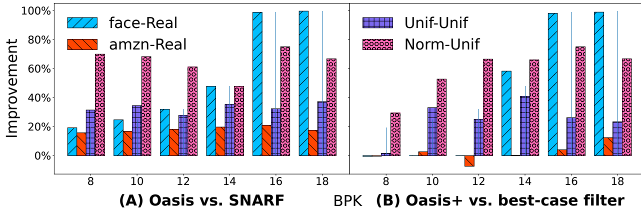

Figure 5: (A) FPR improvement of Oasis against SNARF. (B) Oasis+ compares to the best-case baseline on each workload case by case. Both (A), (B) run on short range queries.

number of keys in the dataset. The table shows that the space pruning method allows Oasis to achieve at least a 0.78 BPK improvement compared to SNARF's implementation in every case.

Oasis+ can effectively navigate the optimal design space between learning- and prefix-based filters. Figure 4 illustrates that Oasis+ consistently achieves a low FPR that either matches or exceeds the performance of all other filters, including Oasis. As shown in Figure 5(B), when comparing Oasis+'s FPR with the best-case baseline filter (the filter with the lowest FPR at a given dataset-workload pair and memory constraint) under various dataset-query distributions for short range workloads, Oasis+ exhibits a slight mismatch with the best-case baseline filter in one case (with a 7.3% decrease). However, in nearly every case, Oasis+ either performs comparably or significantly outperforms the best-case baseline filter (achieving up to a 98.9% improvement). This remarkable performance is attributed to Oasis+'s intelligent approach, which involves identifying intervals unsuitable for learning-based range encoding and substituting them with the prefix-based encoding method.

In short range and point query workloads on the `face-real` dataset, Proteus and REncoder excel with small space budgets (BPK < 12). However, as space budgets increase, Oasis outperforms all baselines. In contrast, Oasis+ adapts to the optimal structure for any situation. With limited space budgets, Oasis+ uses Proteus to encode the majority of keys but switches to Oasis as the dominant encoding method with expanded space budgets. This strategic adaptation enables Oasis+ to consistently maintain robust FPR performance, rivaling or surpassing the best filters in every case.

6.2.2 *Build Time.* Figure 6(A) presents the build time of each filter on the `face-real` of queries ranging within [2, 32]. Notably, Oasis' build time is significantly lower than that of hash-based filters because it processes each key only once during compressed bitmap construction, while hash-based filters require multiple invocations of hash functions for a single key. Also, the number of hash functions used by hash-based filters increases with the space budget, leading to a notable increase in build time with the space budget,

Table 1: Oasis space pruning method saved memory (in BPK) compared to SNARF under various datasets.

BPK	Workloads			
	`face`	`amzn`	`Unif`	`Norm`
8	0.80888	0.881837	0.926188	0.784174
10	0.815124	0.888882	0.932608	0.791904
12	0.807982	0.882253	0.926146	0.785024
14	0.800834	0.875694	0.919322	0.779205
16	0.807037	0.882454	0.926074	0.785694
18	0.81339	0.889322	0.932573	0.79328

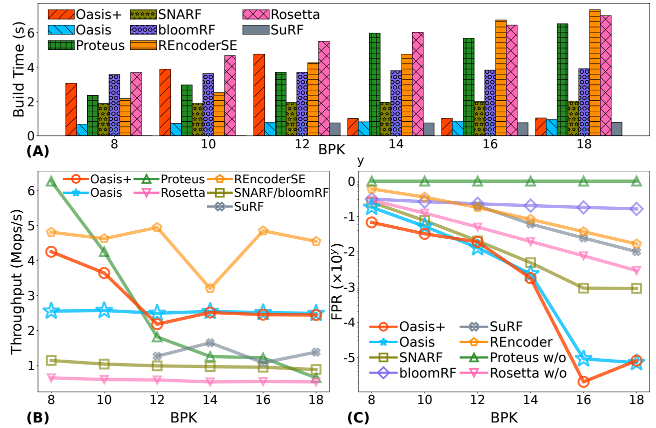

Figure 6: (A) Build time (in s). (B) Filter throughput. (C) Filter FPR on correlation workload. (A) and (B) on `face-real` short range; (C) on `face-correlated` short range.

unlike Oasis. The drop in Oasis+'s build time at BPK 14 is because, when the BPK is less than or equal to 12, Proteus dominants Oasis+'s encoding strategy, while when the BPK exceeds 12, Oasis+ prefers to utilize the Oasis to encode most of the keys. Since Oasis+ involves an additional configuration search process (recalling the nest loop in its construction algorithm) compared to Proteus and Oasis, its build time is slightly higher than that of both filters.

6.2.3 *Throughput.* Oasis' robust throughput is consistently maintained even with variations in the space budget. Both Oasis and Oasis+ outperform most baselines in throughput. Hash-based filters exhibit lower throughput than Oasis due to the need to check the existence of all unique prefixes within the range during range queries. In contrast, Oasis checks only twice at query boundaries, and its block-based compressed bitmap pruning strategy further reduces query time. Despite Oasis' high throughput, REncoder can surpass it by applying SIMD instructions. Notably, Oasis achieves significantly higher throughput than SNARF, approximately 2 to 3 times greater, thanks to Oasis' space pruning strategy during the query process. The fluctuations in Oasis+'s throughput are attributed to Oasis+'s structure variations.

6.2.4 *Correlated Workload.* Here, we show that both Oasis and Oasis+ can handle correlated workloads. To maintain fairness, none of the filters in this experiment access workload information through sampling. The evaluation focuses on the `face-correlated` workload, including queries from 2 to 32. In Figure 6(C), without sampling, the FPR of both Oasis and Oasis+ is comparable to or even surpasses those hash-based filters. This is because (1) most state-of-the-art range filters heavily rely on sampling-based configuration settings to adapt to workload variations, and (2) the keys in the `face` dataset are densely distributed, resulting in Oasis constructing an instance with a relatively small overall BsR. For Proteus, however, without query sampling, it will use its default setup that only encodes half of the entire key length, which is insufficient to handle the correlation workload. While the performance gap between bloomRF and other baseline methods primarily arises from bloomRF's unsuitability for the `face` dataset, a trend is also evident in experiments on the `face-real` datasets.

7 SYSTEM EVALUATION

In this section, we integrate Oasis and Oasis+ into RocksDB v6.20.3, demonstrating their capacity to enhance RocksDB's end-to-end query performance. Compared to Proteus and SNARF, we achieve up to 1.8× and 1.4× performance improvements, respectively.

Integrating Range Filter with RocksDB. RocksDB is an LSM-tree storage engine based key-value database that stores data in immutable files known as Sorted String Tables (SST). These files are organized hierarchically, with each level containing multiple files and subsequent levels having several times more files than preceding levels. To integrate range filters into RocksDB seamlessly, we follow the approach outlined in [33], exposing essential filter APIs for populating and querying. Further, we extend support for range filters in `Seek` operation, create a range filter instance for each SST file, and maintain a dictionary to store each run's filters.

Experimental Setup. The experiments in this section use datasets and query workloads described in Section 6. Each key is assigned a 512-byte value, with half of its content set to zero, following the approach in [21, 33]. In each experimental, we populate RocksDB with 50M unsigned 8-byte keys. After the population phase, we initially force flush the MemTables and wait for all compaction processes to finish. Once these operations conclude, we conduct 5M empty range queries on the stable state of the RocksDB instance. For RocksDB configuration, we follow the settings outlined in [21].

End-to-End Performance. Here, we evaluate the end-to-end range query latency of Oasis, Oasis+, SNARF, Proteus, and SuRF across four dataset-workload pairs, as depicted in Figure 7. Rows in the figure correspond to the `Unif` and `Norm` datasets, while columns represent short range and long range query workloads. As Rosetta is specialized for very short range queries (2 to 16), we exclude it from experiments for fairness.

In read-only workloads with RocksDB, accessing SST files during queries is the primary source of latency. Therefore, achieving a lower FPR reduces overall system latency, as shown in Figure 7, where latency trends align closely with FPR trends in standalone experiments. Notably, in every scenario, Oasis and Oasis+ consistently exhibit lower latency than other baselines, surpassing Proteus and SNARF by up to 1.8× and 1.4×, respectively. However, as FPR decreases, the probability of loading disk blocks into memory diminishes, leading to a higher proportion of CPU cost in end-to-end latency. For instance, in `Norm-Unif`, when BPK is greater than or equal to 12, SuRF's FPR is higher than Proteus and SNARF. Nevertheless, SuRF's query throughput in this dataset surpasses both, resulting in lower latency compared to Proteus and SNARF.

8 RELATED WORK

Trie-based prefix range filter. The Adaptive Range Filter [1], part of Project Siberia [26] within Hekaton [11], utilizes a binary trie and additional information bits to cover the entire keyspace. It uses query sampling for training to decide which nodes to keep and encodes the entire tree into a bit sequence. However, ARF's encoding is space-consuming, and its training process can be time-intensive. In contrast, SuRF [54] does not rely on query sampling during construction. SuRF-Base uses [19] to encode common key prefixes. SuRF-Hash enhances this approach by adding fixed hash

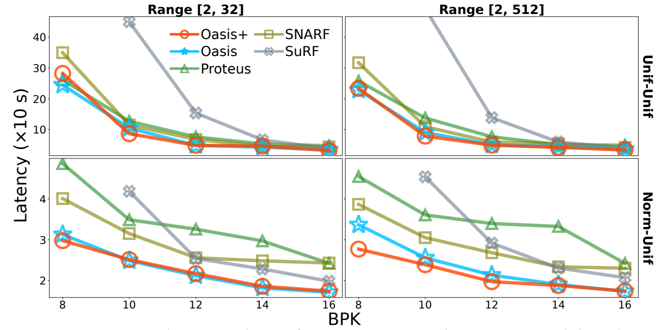

Figure 7: End-to-end performance on diverse workloads.

bits to reduce FPR in point query processing. SuRF-Real stores fixed bits of the following suffix for each key to support range queries.

Hash-based prefix range filter. [10] records predefined prefix lengths of each key in a series of Bloom filters. Rosetta [33] extends this concept by storing each key's prefix length in separate Bloom filters, proposing an optimization method using sampled queries for memory allocation. bloomRF [37], derived from Rosetta, extends Bloom filter to preserve keys' local order, accelerating queries and enhancing long range queries performance. REncoder [50], like bloomRF, maintains suffix information of keys with Bitmap Tree encoding and utilizes SIMD instruction to improve query speed.

Hybrid prefix range filter. Proteus [21] integrates trie-based and hash-based filter design space using the CPFPR model. It employs a two-layer structure with SuRF in the upper layer and a prefix Bloom filter in the lower layer. However, its reliance on sampled queries limits practical applicability in real-world scenarios.

Learning-based range filter. SNARF [47] uses linear spline models to represent the CDF of the dataset, employing it as filter's "hash function". It also utilizes compaction techniques like Elias-Fano [38] and Golomb [17] to compress its sparse bit array, conserving space.

Learning-enhanced data structure. Learning-enhanced techniques have been applied in indexing [12, 15, 16, 20, 23, 25, 29, 31, 45, 53, 57] and filters [23, 27, 28, 35, 48]. Like LRF, learned indexes aim to model the eCDF of datasets. However, their complex training process and excessive hyperparameters cause longer build time and space inefficiency. Learned filters, created by classification models, are not designed for range queries.

9 CONCLUSIONS

This paper presents Oasis, a novel learned range filter with excellent low FPR for both point and range queries. Oasis achieves this by (1) pruning large empty ranges, (2) adaptively assigning the space budget, and (3) utilizing a segmentation algorithm for optimal configuration. Additionally, we enhance Oasis with Oasis+, integrating the design space of learning- and prefix-based filters for robust performance across various workloads.

ACKNOWLEDGMENTS

Guoduo Chen and Zhenying He was supported by NSFC (Grant No. 62272106). Siqiang Luo was supported by NTU-NAP startup grant (022029-00001).

REFERENCES

[1] Karolina Alexiou, Donald Kossmann, and Per-Åke Larson. 2013. Adaptive Range Filters for Cold Data: Avoiding Trips to Siberia. *Proc. VLDB Endow.* 6, 14 (sep 2013), 1714–1725. https://doi.org/10.14778/2556549.2556556

[2] Sattam Alsubaiee, Alexander Behm, Vinayak Borkar, Zachary Heilbron, Young-Seok Kim, Michael J. Carey, Markus Dreseler, and Chen Li. 2014. Storage Management in AsterixDB. *Proc. VLDB Endow.* 7, 10 (jun 2014), 841–852. https://doi.org/10.14778/2732951.2732958

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

[4] 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 Trans. Comput. Syst.* 26, 2, Article 4 (jun 2008), 26 pages. https://doi.org/10.1145/1365815.1365816

[5] Lixiang Chen, Ruihao Chen, Chengcheng Yang, Yuxing Han, Rong Zhang, Xuan Zhou, Peiquan Jin, and Weining Qian. 2023. Workload-Aware Log-Structured Merge Key-Value Store for NVM-SSD Hybrid Storage. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. 2207–2219. https://doi.org/10.1109/ICDE55515.2023.00171

[6] Wikipedia contributors. 2023. Cauchy–Schwarz inequality. https://en.wikipedia.org/wiki/CauchyâĂŞSchwarz_inequality [Online; accessed December-2023].

[7] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana, USA) *(SoCC '10)*. Association for Computing Machinery, New York, NY, USA, 143–154. https://doi.org/10.1145/1807128.1807152

[8] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2018. Optimal Bloom Filters and Adaptive Merging for LSM-Trees. *ACM Trans. Database Syst.* 43, 4, Article 16 (dec 2018), 48 pages. https://doi.org/10.1145/3276980

[9] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's Highly Available Key-Value Store. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles* (Stevenson, Washington, USA) *(SOSP '07)*. Association for Computing Machinery, New York, NY, USA, 205–220. https://doi.org/10.1145/1294261.1294281

[10] Sarang Dharmapurikar, Praveen Krishnamurthy, and David E. Taylor. 2003. Longest Prefix Matching Using Bloom Filters. In *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (Karlsruhe, Germany) *(SIGCOMM '03)*. Association for Computing Machinery, New York, NY, USA, 201–212. https://doi.org/10.1145/863955.863979

[11] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL Server's Memory-Optimized OLTP Engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, New York, USA) *(SIGMOD '13)*. Association for Computing Machinery, New York, NY, USA, 1243–1254. https://doi.org/10.1145/2463676.2463710

[12] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, David Lomet, and Tim Kraska. 2020. ALEX: An Updatable Adaptive Learned Index. 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, 969–984. https://doi.org/10.1145/3318464.3389711

[13] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. 2017. Optimizing Space Amplification in RocksDB.. In *CIDR*, Vol. 3. 3.

[14] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. 2021. RocksDB: Evolution of Development Priorities in a Key-Value Store Serving Large-Scale Applications. *ACM Trans. Storage* 17, 4, Article 26 (oct 2021), 32 pages. https://doi.org/10.1145/3483840

[15] Paolo Ferragina and Giorgio Vinciguerra. 2020. The PGM-Index: A Fully-Dynamic Compressed Learned Index with Provable Worst-Case Bounds. *Proc. VLDB Endow.* 13, 8 (apr 2020), 1162–1175. https://doi.org/10.14778/3389133.3389135

[16] Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2019. FITing-Tree: A Data-Aware Index Structure. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1189–1206. https://doi.org/10.1145/3299869.3319860

[17] R. Gallager and D. van Voorhis. 1975. Optimal Source Codes for Geometrically Distributed Integer Alphabets (Corresp.). *IEEE Trans. Inf. Theor.* 21, 2 (sep 1975), 228–230. https://doi.org/10.1109/TIT.1975.1055357

[18] Mayank Goswami, Allan Grønlund, Kasper Green Larsen, and Rasmus Pagh. 2015. Approximate Range Emptiness in Constant Time and Optimal Space. In *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms* (San Diego, California) *(SODA '15)*. Society for Industrial and Applied Mathematics, USA, 769–775.

[19] G. Jacobson. 1989. Space-Efficient Static Trees and Graphs. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science (SFCS '89)*. IEEE Computer Society, USA, 549–554. https://doi.org/10.1109/SFCS.1989.63533

[20] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. 2020. RadixSpline: A Single-Pass Learned Index. In *Proceedings of the Third International Workshop on Exploiting Artificial Intelligence Techniques for Data Management* (Portland, Oregon) *(aiDM '20)*. Association for Computing Machinery, New York, NY, USA, Article 5, 5 pages. https://doi.org/10.1145/3401071.3401659

[21] Eric R. Knorr, Baptiste Lemaire, Andrew Lim, Siqiang Luo, Huanchen Zhang, Stratos Idreos, and Michael Mitzenmacher. 2022. Proteus: A Self-Designing Range Filter. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 1670–1684. https://doi.org/10.1145/3514221.3526167

[22] Haridimos Kondylakis, Niv Dayan, Kostas Zoumpatianos, and Themis Palpanas. 2019. Coconut Palm: Static and Streaming Data Series Exploration Now in Your Palm. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1941–1944. https://doi.org/10.1145/3299869.3320233

[23] 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

[24] Avinash Lakshman and Prashant Malik. 2010. Cassandra: A Decentralized Structured Storage System. 44, 2 (apr 2010), 35–40. https://doi.org/10.1145/1773912.1773922

[25] Hai Lan, Zhifeng Bao, J. Shane Culpepper, and Renata Borovica-Gajic. 2023. Updatable Learned Indexes Meet Disk-Resident DBMS - From Evaluations to Design Choices. *Proc. ACM Manag. Data* 1, 2, Article 139 (jun 2023), 22 pages. https://doi.org/10.1145/3589284

[26] Justin J. Levandoski, Per-Åke Larson, and Radu Stoica. 2013. Identifying hot and cold data in main-memory databases. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. 26–37. https://doi.org/10.1109/ICDE.2013.6544811

[27] Meng Li, Deyi Chen, Haipeng Dai, Rongbiao Xie, Siqiang Luo, Rong Gu, Tong Yang, and Guihai Chen. 2022. Seesaw Counting Filter: An Efficient Guardian for Vulnerable Negative Keys During Dynamic Filtering. In *Proceedings of the ACM Web Conference 2022 (WWW '22)*. Association for Computing Machinery, New York, NY, USA, 2759–2767. https://doi.org/10.1145/3485447.3511996

[28] Meng Li, Wenqi Luo, Haipeng Dai, Huayi Chai, Rong Gu, Xiaoyu Wang, and Guihai Chen. 2024. The Reinforcement Cuckoo Filter. In *IEEE INFOCOM 2024 - IEEE Conference on Computer Communications*.

[29] Pengfei Li, Hua Lu, Rong Zhu, Bolin Ding, Long Yang, and Gang Pan. 2023. DILI: A Distribution-Driven Learned Index. *Proc. VLDB Endow.* 16, 9 (may 2023), 2212–2224. https://doi.org/10.14778/3598581.3598593

[30] Junfeng Liu, Fan Wang, Dingheng Mo, and Siqiang Luo. 2024. Structural Designs Meet Optimality: Exploring Optimized LSM-tree Structures in A Colossal Configuration Space. *SIGMOD '24* (2024). https://doi.org/10.1145/3654978

[31] Baotong Lu, Jialin Ding, Eric Lo, Umar Farooq Minhas, and Tianzheng Wang. 2021. APEX: A High-Performance Learned Index on Persistent Memory. *Proc. VLDB Endow.* 15, 3 (nov 2021), 597–610. https://doi.org/10.14778/3494124.3494141

[32] Chen Luo and Michael J. Carey. 2019. LSM-Based Storage Techniques: A Survey. *The VLDB Journal* 29, 1 (jul 2019), 393–418. https://doi.org/10.1007/s00778-019-00555-y

[33] 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

[34] 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 (sep 2020), 1–13. https://doi.org/10.14778/3421424.3421425

[35] Michael Mitzenmacher. 2018. A Model for Learned Bloom Filters, and Optimizing by Sandwiching. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems* (Montréal, Canada) *(NIPS'18)*. Curran Associates Inc., Red Hook, NY, USA, 462–471.

[36] Dingheng Mo, Fanchao Chen, Siqiang Luo, and Caihua Shan. 2023. Learning to Optimize LSM-trees: Towards A Reinforcement Learning based Key-Value Store for Dynamic Workloads. *Proc. ACM Manag. Data* 1, 3, Article 213 (nov 2023), 25 pages. https://doi.org/10.1145/3617333

[37] Bernhard Mößner, Christian Riegger, Arthur Bernhardt, and Ilia Petrov. 2022. bloomRF: On performing range-queries in Bloom-Filters with piecewise-monotone hash functions and prefix hashing. *arXiv preprint arXiv:2207.04789* (2022).

[38] Giuseppe Ottaviano and Rossano Venturini. 2014. Partitioned Elias-Fano Indexes. In *Proceedings of the 37th International ACM SIGIR Conference on Research &

Development in Information Retrieval (Gold Coast, Queensland, Australia) *(SIGIR '14)*. Association for Computing Machinery, New York, NY, USA, 273–282. https://doi.org/10.1145/2600428.2609615

[39] Dimitris Papadias, Jun Zhang, Nikos Mamoulis, and Yufei Tao. 2003. - Query Processing in Spatial Network Databases. In *Proceedings 2003 VLDB Conference*, Johann-Christoph Freytag, Peter Lockemann, Serge Abiteboul, Michael Carey, Patricia Selinger, and Andreas Heuer (Eds.). Morgan Kaufmann, San Francisco, 802–813. https://doi.org/10.1016/B978-012722442-8/50076-8

[40] Ivan Luiz Picoli, Philippe Bonnet, and Pinar Tözün. 2019. LSM Management on Computational Storage. In *Proceedings of the 15th International Workshop on Data Management on New Hardware* (Amsterdam, Netherlands) *(DaMoN'19)*. Association for Computing Machinery, New York, NY, USA, Article 17, 3 pages. https://doi.org/10.1145/3329785.3329927

[41] Xuecheng Qi, Huiqi Hu, Jinwei Guo, Chenchen Huang, Xuan Zhou, Ning Xu, Yu Fu, and Aoying Zhou. 2023. High-availability in-memory key-value store using RDMA and Optane DCPMM. *Frontiers Comput. Sci.* 17, 1 (2023), 171603. https://doi.org/10.1007/S11704-022-1123-8

[42] Meta 2012. *RocksDB*. Meta. https://rocksdb.org/

[43] Russell Sears, Mark Callaghan, and Eric Brewer. 2008. Rose: Compressed, Log-Structured Replication. *Proc. VLDB Endow.* 1, 1 (aug 2008), 526–537. https://doi.org/10.14778/1453856.1453914

[44] Russell Sears and Raghu Ramakrishnan. 2012. BLSM: A General Purpose Log Structured Merge Tree. 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, 217–228. https://doi.org/10.1145/2213836.2213862

[45] Zhaoyan Sun, Xuanhe Zhou, and Guoliang Li. 2023. Learned Index: A Comprehensive Experimental Evaluation. *Proc. VLDB Endow.* 16, 8 (jun 2023), 1992–2004. https://doi.org/10.14778/3594512.3594528

[46] Joy A. Thomas Thomas M. Cover. 2006. *Elements of Information Theory* (2 ed.). Wiley-Interscience, 127—-128.

[47] Kapil Vaidya, Subarna Chatterjee, Eric Knorr, Michael Mitzenmacher, Stratos Idreos, and Tim Kraska. 2022. SNARF: A Learning-Enhanced Range Filter. *Proc. VLDB Endow.* 15, 8 (apr 2022), 1632–1644. https://doi.org/10.14778/3529337.3529347

[48] Kapil Vaidya, Eric Knorr, Tim Kraska, and Michael Mitzenmacher. 2020. Partitioned learned bloom filter. *arXiv preprint arXiv:2006.03176* (2020).

[49] Ruihong Wang, Jianguo Wang, Prishita Kadam, M. Tamer Özsu, and Walid G. Aref. 2023. dLSM: An LSM-Based Index for Memory Disaggregation. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. 2835–2849. https://doi.org/10.1109/ICDE55515.2023.00217

[50] Ziwei Wang, Zheng Zhong, Jiarui Guo, Yuhan Wu, Haoyu Li, Tong Yang, Yaofeng Tu, Huanchen Zhang, and Bin Cui. 2023. REncoder: A Space-Time Efficient Range Filter with Local Encoder. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. 2036–2049. https://doi.org/10.1109/ICDE55515.2023.00158

[51] Qingsong Wen, Liang Sun, Fan Yang, Xiaomin Song, Jingkun Gao, Xue Wang, and Huan Xu. 2020. Time series data augmentation for deep learning: A survey. *arXiv preprint arXiv:2002.12478* (2020).

[52] 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

[53] Geoffrey X. Yu, Markos Markakis, Andreas Kipf, Per-Åke Larson, Umar Farooq Minhas, and Tim Kraska. 2022. TreeLine: An Update-in-Place Key-Value Store for Modern Storage. *Proc. VLDB Endow.* 16, 1 (sep 2022), 99–112. https://doi.org/10.14778/3561261.3561270

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

[55] Teng Zhang, Jianying Wang, Xuntao Cheng, Hao Xu, Nanlong Yu, Gui Huang, Tieying Zhang, Dengcheng He, Feifei Li, Wei Cao, Zhongdong Huang, and Jianling Sun. 2020. FPGA-Accelerated Compactions for LSM-based Key-Value Store. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. USENIX Association, Santa Clara, CA, 225–237. https://www.usenix.org/conference/fast20/presentation/zhang-teng

[56] Xin Zhang, Qizhong Mao, Ahmed Eldawy, Vagelis Hristidis, and Yihan Sun. 2022. Bi-Directional Log-Structured Merge Tree. In *Proceedings of the 34th International Conference on Scientific and Statistical Database Management* (Copenhagen, Denmark) *(SSDBM '22)*. Association for Computing Machinery, New York, NY, USA, Article 19, 4 pages. https://doi.org/10.1145/3538712.3538730

[57] Zhou Zhang, Zhaole Chu, Peiquan Jin, Yongping Luo, Xike Xie, Shouhong Wan, Yun Luo, Xufei Wu, Peng Zou, Chunyang Zheng, Guoan Wu, and Andy Rudoff. 2022. PLIN: A Persistent Learned Index for Non-Volatile Memory with High Performance and Instant Recovery. *Proc. VLDB Endow.* 16, 2 (oct 2022), 243–255. https://doi.org/10.14778/3565816.3565826

[58] Fuheng Zhao, Leron Reznikov, Divyakant Agrawal, and Amr El Abbadi. 2023. Autumn: A Scalable Read Optimized LSM-tree based Key-Value Stores with Fast Point and Range Read Speed. *arXiv preprint arXiv:2305.05074* (2023).

[59] Wenshao Zhong, Chen Chen, Xingbo Wu, and Song Jiang. 2021. REMIX: Efficient Range Query for LSM-trees. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, 51–64. https://www.usenix.org/conference/fast21/presentation/zhong