Optimizing Collections of Bloom Filters within a Space Budget

Gabriel Mersy
The University of Chicago
gmersy@uchicago.edu

Zhuo Wang
The University of Chicago
zhuo1@uchicago.edu

Stavros Sintos
University of Illinois Chicago
stavros@uic.edu

Sanjay Krishnan
The University of Chicago
skr@uchicago.edu

ABSTRACT

With a single Bloom filter, one can approximately answer set membership queries within a space budget. Practical systems often use collections of Bloom filters to facilitate applications such as data skipping, sideways information passing, and network filtering. While the optimal space-to-accuracy allocation is well-understood for a single filter, jointly optimizing how space is used across a collection of filters is yet to be studied. We pose this problem in the following way: (1) let's assume that each Bloom filter has some likelihood of being queried, and (2) given knowledge of this likelihood, how do we allocate space to minimize the expected false positive rate? In other words, "hot" filters are allocated more space, and "cold" filters are allocated less space. In this paper, we show how to solve this optimization problem. We first develop the concept of a "truncated" Bloom filter and theoretically analyze its false positive rate. We then formulate an optimization problem for a collection of truncated Bloom filters that minimizes the false positive rate across a utility distribution while meeting a strict space budget. Next, we show that the problem is convex and find a fast relaxation. Lastly, we apply our method to data skipping and full-text search, demonstrating its effectiveness across the range of possible space budgets when compared to the state of the art.

PVLDB Reference Format:
Gabriel Mersy, Zhuo Wang, Stavros Sintos, and Sanjay Krishnan. Optimizing Collections of Bloom Filters within a Space Budget. PVLDB, 17(11): 3551 - 3564, 2024.
doi:10.14778/3681954.3682020

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/gmersy/truncated-bloom-filter.

1 INTRODUCTION

A Bloom filter is a sketch that can approximately answer set membership queries within a space budget. Bloom filters have a crucial property that does not yield false negative results, which makes them highly valuable in database systems. The applications of Bloom filters span key-value stores [19, 44], data skipping [17, 18], query optimization [26, 60], and search engines [27]. Filters are often employed in these applications to reduce storage reads by serving as an intermediary index structure that can short-circuit data loading.

Let's consider a typical use case for a Bloom filter. A data volume might be partitioned across several different files. A user who is interested in identifying a particular record that matches an equality condition (e.g., an employee identified by a social security number) wants to query this data. On one hand, we can simply scan the entire data volume each time we want to answer this query. This approach requires no indexing but is slow. Another approach is to build an index over the search keys, which can require substantial additional space and may be hard to operationalize in some environments (e.g., a write-heavy database or a distributed system). A Bloom filter allows one to span these two extremes by annotating each partition with a space-limited filter that determines if it needs to be processed or not – the larger the filter, the more precise the skipping will be. This filter is straightforward to maintain under insertions.

While the design space of a single Bloom filter is well-understood [34], use cases like the one above show that we often consider collections of such filters (i.e., filters on each partition). The standard approach is to uniformly size each filter in a filter collection, which has a few key drawbacks. First, while a single Bloom filter has a fixed size regardless of the number of insertions, the total size of a collection of filters will grow as more filters are utilized (e.g., over new partitions of data). If we wish to cache the filter collection in memory, the constant growth can be problematic. Next, not all partitions are equally likely to be queried, and uniform sizing allocates the same precision to frequently queried partitions as to rare ones. This paper studies the following problem: *given an anticipated query distribution and an overall space budget, can we optimally size each filter to maximize the expected precision?*

Let's contextualize this problem statement in the example above, and make it much more concrete. Consider a telemetry system, such as AWS CloudWatch, which can generate a vast amount of unstructured log data. In systems like CloudWatch, log data is broken up into temporal partitions of roughly the same size. Over this log data, users can run keyword search queries. Bloom filters constructed on each partition can help skip partitions that could not possibly contain the desired keyword. While in general, one needs to process all partitions, we know that more recent data is more likely to be relevant to a user (the anticipated query distribution). Often, users set a time interval restricting how far back the search should go. The solution to the optimization problem above can construct an in-memory filter collection with a fixed size that should automatically reduce the precision of filters for the older partitions.

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. 11 ISSN 2150-8097.
doi:10.14778/3681954.3682020

Going beyond this example, there is a broader data retrieval architecture that this algorithmic problem enables. Data are stored in partitions on some form of slow storage, e.g., disk or a blob storage system. We wish to answer set membership queries over this data, e.g., keyword search. These partitions are indexed in memory on a query processing node with a guaranteed max size for the entire collection of Bloom filters. Since the total size of all Bloom filters is fixed via optimization, they can be predictably stored in memory. As more data are added, we show how this in-memory index can be re-optimized to maintain a fixed size (albeit with an increased error rate for some queries). The result is *a fixed-size index over growing data.* While this implies that overall filtering accuracy goes down over time, if there is a sufficient, known skew in the query distribution (e.g., a bias towards querying more recent filters), then perhaps the query-weighted accuracy does not degrade as much.

We contribute the algorithms to make this architecture a reality. Assuming a query distribution as a proxy for *utility* is not a new idea. For example, caching heavily leverages this principle, as it persists certain data items that are accessed more frequently than others [5]. In particular, we will show how to find an optimal index structure of Bloom filters that minimizes the false positive rate over a utility distribution, while adhering to a strict space budget. This can be thought of as a form of "partial" caching, where rather than evicting a low-utility filter completely from the cache, we simply reduce its precision. Interestingly enough, this is actually the optimal thing to do in the case of a Bloom filter.

The contributions of this paper are as follows.

- We propose the *truncated Bloom filter* as a simple extension in which the filter is shortened after construction, enabling the trade-off between space and false positive rate to be traversed in a fine-grained fashion (§3).
- We formulate a constrained optimization problem for a collection of truncated Bloom filters that assigns higher false positive rates to lower-utility filters by modulating filter lengths, subject to a strict bit-level space budget (§4).
- We prove, at odds with intuition, that the resulting optimization problem is convex. We further derive a relaxation of the original problem that is also provably convex and features an objective function that runs in linear time (§4).
- We demonstrate the effectiveness of our method when compared to the state of the art on two representative Bloom filter applications (§5).

2 BACKGROUND AND PROBLEM STATEMENT

We begin by covering some background and the problem statement.

2.1 Bloom Filters

A Bloom filter [6, 34, 35] is a probabilistic data structure for representing a set that can answer approximate membership queries. For convenience, we write $\mathcal{B}$ when we refer to a Bloom filter throughout the paper. An m-bit Bloom filter $\mathcal{B}$ contains n elements and is associated with k independent hash functions $h_1, h_2, \ldots, h_k$. We denote these parameters as the 3-tuple (m, k, n). We will use the notation $|\cdot|$ to denote the size of a sketch. For a Bloom filter, this is initially given by the parameter m.

The filter is initially an array of all zeros. When adding an element b to $\mathcal{B}$ a total of k hashed positions in the filter $\mathcal{B}[h_1(b)]$, $\mathcal{B}[h_2(b)], \ldots, \mathcal{B}[h_k(b)]$ are set to 1. For a query key q each of the k hashed positions in the filter is probed. If a position is not set to 1, then a negative (−) result is returned. If all of the positions are set to 1, then a positive (+) result is returned. If the key is in the filter, then it returns the correct answer. If the key is not in the filter then it returns the correct answer with probability $1 - \varepsilon$ for a small parameter $\varepsilon \in (0, 1)$. In other words, with probability ε the filter returns a positive result for a key that is not in the filter (i.e., a false positive).

Denote q^+ as a positive key that is in the filter and q^- as a negative key that is not in the filter. A Bloom filter by design has a false positive rate of $\mathbb{P}(+ \mid q^-) = \varepsilon$ and a zero false negative rate $\mathbb{P}(- \mid q^+) = 0$. The false positive rate ε is given as $\varepsilon \approx \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k$. It has been shown that this characterization is a very close approximation but is not quite correct due to a flawed bit independence assumption [7]. Since the alternative is an impractical combinatorial function, we exclusively make use of the closed-form approximation, as is standard [19].

Filter Collections. A filter collection is a set of N Bloom filters $\{\mathcal{B}_1, \mathcal{B}_2, \ldots, \mathcal{B}_N\}$ where each filter $\mathcal{B}_i$ represents a data object $\mathcal{D}_i$. We further assume that there is a utility function $u : \mathcal{D} \to \mathbb{R}_{\geq 0}$ that maps each data object to a non-negative real number. Throughout the paper, we write u to refer to a utility value that has already been pre-computed by the function in question. A utility function is simply a proxy for query frequency. In many environments, exact frequency statistics can be maintained by observing the access distribution of queries during a historical time window. In environments where the overhead of these exact statistics is undesirable, such as main-memory OLTP databases, access frequencies can be efficiently estimated using a combination of log sampling and exponential smoothing algorithms [31].

Utility-Weighted False Positive Rate. In the same way that we can calculate the false positive rate ε for a single Bloom filter, we can calculate the expected false positive rate for a filter collection using the individual false positive rates $\varepsilon_1, ..., \varepsilon_N$ and utility values $u_1, u_2, \ldots, u_N$. We arrive at a final figure of merit which is the utility-weighted false positive rate:

$$\mathcal{E} = \sum_{i=1}^{N} u_i \cdot \varepsilon_i \tag{1}$$

If one interprets the utility values as probabilities, this function measures the overall precision of the filter collection, e.g., how accurately the collection skips data given known partition query probabilities.

2.2 Problem Statement

We would like to produce a final index structure that allocates space to Bloom filters within a space budget B by minimizing the utility-weighted false positive rate $\mathcal{E}$.

Definition 2.1 (Problem Statement). We are given a space budget B, data objects $\mathcal{D}_1, \mathcal{D}_2, \ldots \mathcal{D}_N$ and corresponding utility values

$u_1, u_2, \ldots, u_N$. The goal is to produce a filter collection such that:

$$\sum_{i=1}^{N} |\mathcal{B}_i| \leq B$$

by minimizing

$$\min \sum_{i=1}^{N} u_i \cdot \varepsilon_i$$

where $u_i = u(\mathcal{D}_i)$ and $\varepsilon_i = \phi(\mathcal{B}_i)$ with ϕ denoting the Bloom filter false positive rate formula.

This problem statement is very similar to a caching problem statement at first glance; namely, identify the data items with the highest utility. The key difference when compared to standard caching techniques is that items are not simply evicted or cached according to a space budget. We now have the additional option of degrading the accuracy of an item. It may very well be that the optimal solution in some cases is to allocate 0 bits to an item, but that should only be a consequence of the optimization algorithm.

This formulation is further an instance of a broader problem that we call a *utility-compressed sketch*. In the general case, the space for a particular data object's sketch $|\mathcal{X}_i|$ is allocated by minimizing a weighted sum across error functions of the form $\phi(|\mathcal{X}_i|)$ subject to a budget constraint $\sum_{i=1}^{N} |\mathcal{X}_i| \leq B$. In typical sketches, the error function ϕ is monotonic in space. We can think of a utility-compressed sketch as an optimization over lossy data structures that represent data items at several different "resolutions", which is related to similar problems in data compression [3]. There are many possible definitions of $\phi(|\mathcal{X}_i|)$. In the case of a Bloom filter, $\phi(|\mathcal{X}_i|)$ is the false positive rate as a function of its size – we could apply the same reasoning to other sketches such as the count-min sketch [16].

3 TRUNCATED BLOOM FILTERS

To solve this problem, we will introduce a new sketch called a truncated Bloom filter. It is based on the simple idea that previously-allocated space in a Bloom filter can later on be revoked. This concept is useful for a few key reasons. First, it allows us to formulate the optimization problem as a revocation of previously-allocated space; that is, we can start with a uniform allocation of space and remove bits based on the optimization objective. Second, it allows us to operate in an online setting where new data are added but we wish to keep a fixed storage size. Finally, it allows for consistent querying across the entire filter collection with a fixed set of hash functions.

The standard Bloom filter is inflexible when there are changing requirements for false positive rates. Both the target false positive rate and the expected number of elements in the filter are often specified prior to construction, which means that the Bloom filter usually must be reconstructed from scratch if less space is desired in the future. While there has been work on resizing fingerprint filters (i.e., expansion or contraction) as the number of elements in the filter changes [20], we study the problem of reducing the space occupied by a Bloom filter, since the standard Bloom filter does not support fine-grained space reduction operations.

3.1 Key Differences

We begin by highlighting the key differences between a truncated Bloom filter and a standard Bloom filter.

Definition 3.1 (Truncated Bloom Filter). Let $\mathcal{B}$ be a Bloom filter with m bits, k hash functions, and n added elements. A *truncated Bloom filter* is a new Bloom filter $\mathcal{B}'$ formed by removing the rightmost $m - m'$ bits from $\mathcal{B}$. The truncated filter $\mathcal{B}'$ satisfies $\mathcal{B}'[1, 2, \ldots, m'] = \mathcal{B}[1, 2, \ldots, m']$.

The only notational difference is that a new parameter m' is introduced to specify the truncated filter length. To operationalize this definition, we show that simply considering the subset of hash functions that fall before m' is sufficient to query the filter.

Definition 3.2 (Valid Hash Function). We say that a hash function $h_s : \mathbb{U} \to [m]$ taking an arbitrary object $o \in \mathbb{U}$ from universe $\mathbb{U}$ as input is a *valid hash function* if $h_s(o) \leq m'$. We say that a hash function is an *invalid hash function* if $h_s(o) > m'$, i.e., a hash function is valid if it maps to the remaining region and is invalid if it maps to the truncated region.

These concepts are also highlighted visually in Figure 1. As we move to the right in the figure, the filters exhibit increasing levels of truncation (i.e., smaller values of m'). Given a negative query key q^- the number of valid hash functions (solid arrows) decreases, while the number of invalid hash functions (dotted arrows) increases.

Membership queries are issued to a truncated filter using the procedure described in Algorithm 1. The key algorithmic difference in querying a truncated Bloom filter is that there is an extra condition to check whether each hash function is valid or not.

Input: Query q, truncated Bloom filter $\mathcal{B}'$ with $m' \leq m$ bits,
k hash functions $h_1, h_2, \ldots, h_k$ where $h_s : \mathbb{U} \to [m]$
for $s \in [k]$ **do**
  if $h_s(q) \leq m' \wedge \mathcal{B}'[h_s(q)] = 0$ **then**
    return –
  end
end
return +

Algorithm 1: Querying a truncated Bloom filter.

There are also a few degenerate cases that are worth noting. First, if $m' = 0$ then the query always returns a positive result and has a 100% false positive rate. Second, if there are no valid hash functions, then the query always returns a positive result and also has a 100% false positive rate. Third, if $m' = m$ then the false positive rate is unchanged and is thus identical to the original filter.

3.2 False Positive Rate Analysis

Aside from the degenerate cases, the false positive rate of a truncated Bloom filter strictly falls between 100% and the original false positive rate. To build intuition concerning the false positive rate of a truncated Bloom filter, consider the first filter in Figure 1. Only 3 out of the 4 hash functions are valid, and the false positive rate in this example is $(1 - (1 - \frac{1}{m_1})^{4n_1})^3$ which is strictly higher than the false positive rate $(1 - (1 - \frac{1}{m_1})^{4n_1})^4$ of the original filter. In fact, the false positive rate is governed by the number of valid hash

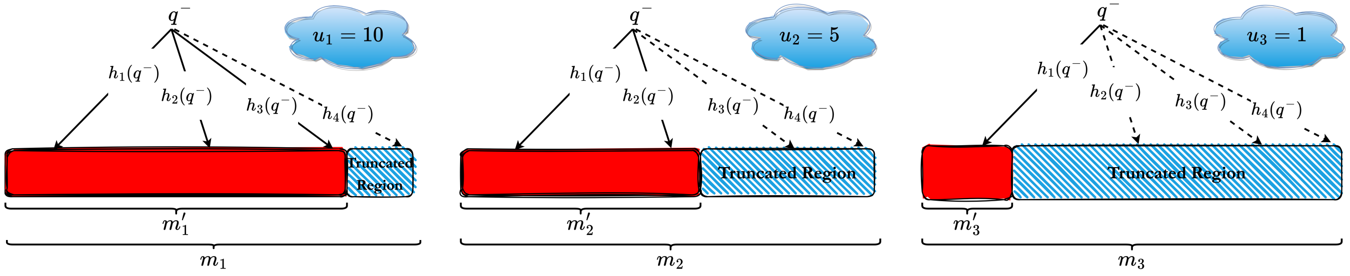

Figure 1: An overview of our method.

functions. Before a particular query, we do not know precisely how many hash functions will be valid. Instead, we define a random function (called the *false positive function*) $\phi(\cdot)$, that returns a random variable for the false positive rate, as shown in Proposition 3.4.

Definition 3.3 (False Positive Function). The false positive function $\phi(V)$ of a truncated Bloom filter is:

$$\phi(V) = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^V \tag{2}$$

where V is a random variable for the number of valid hash functions with range $R_V = \{0\} \cup [k]$.

Proposition 3.4 (False Positive Rate). *Given a collection of k independent and uniform hash functions $h_1, \ldots, h_k$ and the fraction of remaining bits $p = \frac{m'}{m}$ after truncation, the false positive rate ε of a truncated Bloom filter assuming bit independence is:*

$$\varepsilon = \mathbb{E}[\phi(V)] = \sum_{v=0}^{k} \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^v \cdot \binom{k}{v} \cdot p^v (1-p)^{k-v} \tag{3}$$

with $V \sim \text{Bin}(k, p)$ represented as a binomial random variable.

Proof. Let $\mathcal{B}[1, 2, \ldots, m]$ be a Bloom filter and let $\mathcal{B}' = \mathcal{B}[1, 2, \ldots, m']$ be its truncated filter. Let $H = \{h_1, \ldots, h_k\}$ be the set of k hash functions used in filter $\mathcal{B}$. Let $\mathcal{B}'(q) \in \{-, +\}$ be the result of Algorithm 1. The false positive rate is defined as $\varepsilon = \Pr[\mathcal{B}'(q) = + \mid q \notin \mathcal{B}]$, where q is a key that is not stored in Bloom filter $\mathcal{B}$. From the query algorithm we have that $\mathcal{B}'(q^-) = +$ if for every $h \in H$ with $h(q) \leq m'$ it holds that $\mathcal{B}'[h(q)] = \mathcal{B}[h(q)] = 1$. Let $\mathcal{I} \subseteq H$ be a subset of the hash functions and let $q \notin \mathcal{B}$ be a key that is not stored in the filter. We define the random variables:

- $A_{\mathcal{I}}$: Given that $q \notin \mathcal{B}$, if $\bigwedge_{h \in \mathcal{I}} \mathcal{B}[h(q) = 1]$ then $A_{\mathcal{I}} = 1$, otherwise $A_{\mathcal{I}} = 0$.
- $C_{\mathcal{I}}$: if $h(q) \leq m'$ for every $h \in \mathcal{I}$ and $h(q) > m'$ for every $h \in H \setminus \mathcal{I}$ then $C_{\mathcal{I}} = 1$, otherwise $C_{\mathcal{I}} = 0$.

We have,

$$\varepsilon = \Pr[\mathcal{B}'(q) = + \mid q \notin \mathcal{B}] = \Pr\left[\bigvee_{\mathcal{I} \subseteq H} \left(A_{\mathcal{I}} \bigwedge C_{\mathcal{I}}\right)\right]$$

$$= \sum_{\mathcal{I} \subseteq H} \Pr\left[A_{\mathcal{I}} \bigwedge C_{\mathcal{I}}\right] = \sum_{v=0}^{k} \sum_{\mathcal{I} \subseteq H, |\mathcal{I}|=v} \Pr\left[A_{\mathcal{I}} \bigwedge C_{\mathcal{I}}\right]$$

$$= \sum_{v=0}^{k} \sum_{\mathcal{I} \subseteq H, |\mathcal{I}|=v} \Pr[C_{\mathcal{I}}] \cdot \Pr[A_{\mathcal{I}} \mid C_{\mathcal{I}}]$$

$$= \sum_{v=0}^{k} \sum_{\mathcal{I} \subseteq H, |\mathcal{I}|=v} p^v (1-p)^{k-v} \cdot \Pr\left[\bigwedge_{h \in \mathcal{I}} \mathcal{B}[h(q) = 1] \mid q \notin \mathcal{B}\right]$$

$$= \sum_{v=0}^{k} \sum_{\mathcal{I} \subseteq H, |\mathcal{I}|=v} p^v (1-p)^{k-v} \cdot \prod_{h \in \mathcal{I}} \Pr\left[\mathcal{B}[h(q) = 1] \mid q \notin \mathcal{B}\right]$$

$$= \sum_{v=0}^{k} \sum_{\mathcal{I} \subseteq H, |\mathcal{I}|=v} p^v (1-p)^{k-v} \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^v$$

$$= \sum_{v=0}^{k} \binom{k}{v} p^v (1-p)^{k-v} \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^v$$

$$= \mathbb{E}[\phi(V)]$$

□

4 OPTIMIZING FILTER COLLECTIONS

In this section, we formalize a constrained optimization problem that produces a utility-compressed sketch of truncated Bloom filters. We first prove that the optimization problem is convex and can thus be solved using a standard algorithm for convex optimization. We then derive a relaxation that is also shown to be convex. Crucially, the relaxation can be solved faster than the original problem both in theory and in practice.

4.1 An Exact Optimization Problem

We are given a space budget B, utility values $u_1, u_2, \ldots, u_N$ and Bloom filters $\mathcal{B}_1, \mathcal{B}_2, \ldots, \mathcal{B}_N$ constructed with parameter tuples of the form (m_i, k_i, n_i).

Formulation. We begin with a simple description of the optimization problem. The basic intuition is that a higher false positive rate is assigned to a filter that has a lower utility value, and a lower false positive rate is assigned to a filter with a higher utility value. To achieve this desired property, we write $u \cdot \mathbb{E}[\phi(V)]$. The false positive rate of a truncated Bloom filter is modulated by setting the number of remaining bits m' prior to the truncation operation. Therefore, our variables are simply the truncated filter lengths $m'_1, m'_2, \ldots, m'_N$ across our collection of Bloom filters. Referring back to Figure 1, we can see how the utility values affect the truncated filter lengths in our optimization routine.

Following this intuition, the objective is to minimize the dot product between the utility values and the false positive rate terms by finding an optimal assignment of truncated filter lengths, subject to a few constraints. First, the sum of the m_i' must not exceed the budget B. Second, each m_i' must fall between 0 and the original filter length m_i. Using the false positive rate from Equation (3), we define our constrained optimization problem as follows:

$$\begin{aligned} \min_{m_1', \ldots, m_N'} \quad & \sum_{i=1}^{N} u_i \cdot \mathbb{E}[\phi(V_i)] \\ \text{s.t.} \quad & \sum_{i=1}^{N} m_i' \leq B \\ & 0 \leq m_i' \leq m_i, \ \forall i \in [N] \end{aligned}$$

where B is the target budget, and each variable m_i' corresponds to a truncated length. Note that each truncated filter length $m_i' \in \mathbb{R}_{\geq 0}$ is currently treated as a continuous optimization variable. Of course, a fractional truncated filter length is not possible, so we simply take the optimal solution and for each $i \in [N]$ round down to the nearest integer; $m_i' \leftarrow \lfloor m_i' \rfloor$. This ensures that the budget constraint is still satisfied. We can also substitute a budget equality constraint $\sum_{i=1}^{N} m_i' = B$ for the budget inequality constraint $\sum_{i=1}^{N} m_i' \leq B$.

Convexity. In order to establish convexity, we need to verify that the objective and constraint functions are all convex [8]. By inspection, the constraint functions in the above formulation are linear and therefore convex. Hence, it suffices to establish the convexity of the objective function. In Lemma 4.1, we show that the false positive function $\phi(\cdot)$ as written in Equation (2) is convex on $\mathbb{R}_{\geq 0}$.

Lemma 4.1 (Convexity of ϕ). *$\phi(x) = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^x$ is convex on $x \geq 0$ for $m, k, n > 0$.*

Proof. By the definition of convexity, we want to show that the second derivative $\frac{d^2\phi(x)}{dx^2}$ is non-negative. Let $x \geq 0$ and $m, k, n > 0$. Define $w = 1 - \left(1 - \frac{1}{m}\right)^{kn}$ which is a strictly positive value by construction. We differentiate twice to obtain:

$$\frac{d^2\phi(x)}{dx^2} = w^x \log^2(w) \geq 0.$$

□

It has now been shown that the truncated Bloom filter false positive function is convex under mild conditions. We now prove a more surprising result concerning the convexity of the objective function. Intuition derived from a plot of the binomial probability mass function may suggest that the objective function is unlikely to be convex in the optimization variable. On the contrary, we find that the function is indeed convex, as presented in Proposition 4.2. The structure of the proof relies the observation that the binomial distribution is an *exponential family*, which allows us to employ properties of such families to infer that our objective function is convex.

Proposition 4.2 (Objective Convexity). *The objective function $f(\mathbf{m}') = \sum_{i=1}^{N} u_i \cdot \mathbb{E}[\phi(V_i)]$ is convex in $\mathbf{m}'$ on the hyper-rectangle $[0, m_1] \times \ldots \times [0, m_N]$.*

Proof. Recall that $\mathbf{m}' = [m_1', \ldots, m_N']$. We first note the symmetry of the objective function. The core idea is to show that the base function $\mathbb{E}[\phi(V_i)]$ is convex in a single variable m_i' and then deduce that the objective function is composed of a convexity-preserving operation on multiple instances of the base function with distinct variables.

We borrow from the statistics literature the known fact that the binomial distribution is an exponential family for a fixed number of trials (the number of hash functions in our case). In particular, this means that the binomial probability mass function can be rewritten in a canonical form. Let X be an exponential family random variable with probability mass function $\mathbb{P}_X(x; \theta)$ that depends on the parameter θ. Assuming that $\phi(\cdot)$ is convex on the (real interval) range of X and the expectation $\mathbb{E}[X]$ is linear in the parameter θ, then it is known that $\mathbb{E}[\phi(X)]$ is convex in the parameter θ [47, 49].

Let $m_i, k_i, n_i > 0$ and $0 \leq m_i' \leq m_i$ for all $i \in [N]$. Each V_i is a binomial random variable from an exponential family with mass $\mathbb{P}_{V_i}(v_i; p_i)$ where $p_i = \frac{m_i'}{m_i}$. The range of a binomial random variable is non-negative. By Lemma 4.1, it then follows that the function $\phi(\cdot)$ is convex on the range of V_i. The expectation $\mathbb{E}[V_i] = k_i p_i$ is linear in the parameter p_i [10]. Therefore, we have that the function $\mathbb{E}[\phi(V_i)]$ is convex in the success probability parameter p_i. This implies that the second derivative with respect to p_i is non-negative $\frac{d^2\mathbb{E}[\phi(V_i)]}{dp_i^2} \geq 0$ on $[0, 1]$. We differentiate Equation (3) twice with respect to the success probability p_i and obtain:

$$\begin{aligned} \frac{d^2\mathbb{E}[\phi(V_i)]}{dp_i^2} = \sum_{v_i=0}^{k_i} \phi(v_i) \cdot \binom{k_i}{v_i} \cdot \Big[& v_i(v_i - 1){p_i}^{v_i - 2}(1 - p_i)^{k_i - v_i} \\ & -2v_i(k_i - v_i){p_i}^{v_i - 1}(1 - p_i)^{k_i - v_i - 1} \\ & +(k_i - v_i)(k_i - v_i - 1){p_i}^{v_i}(1 - p_i)^{k_i - v_i - 2} \Big] \end{aligned}$$

It remains to show that $\mathbb{E}[\phi(V_i)]$ is also convex in the truncated filter length variable m_i'. Notice that $\frac{dp_i}{dm_i'} = \frac{1}{m_i} > 0$. We now differentiate $\mathbb{E}[\phi(V_i)]$ twice with respect to m_i' and find that the following holds on $[0, m_i]$:

$$\frac{d^2\mathbb{E}[\phi(V_i)]}{d{m_i'}^2} = \frac{1}{{m_i}^2} \cdot \frac{d^2\mathbb{E}[\phi(V_i)]}{dp_i^2} \geq 0$$

Specifically, the second derivative with respect to our desired variable m_i' can be rewritten as a positive constant multiplied by the second derivative with respect to p_i evaluated at m_i'/m_i. Hence, $\mathbb{E}[\phi(V_i)]$ is convex in the variable m_i' since $\frac{d^2\mathbb{E}[\phi(V_i)]}{d{m_i'}^2} \geq 0$. Lastly, a weighted sum of N convex functions with non-negative coefficients $u_i \geq 0$ is known to preserve convexity, so the objective function f is convex in $\mathbf{m}'$ on the hyper-rectangle $[0, m_1] \times \ldots \times [0, m_N]$. □

Complexity Analysis. The time complexity of computing the objective is $O(N \max_{i \in [N]} k_i)$. When objective functions are computationally expensive, the evaluation of gradient vectors and Hessian matrices at high-dimensional points in space can become a critical bottleneck for many convex optimization solvers.

In our case, linearity of differentiation suggests that the run time of the underlying solver will be highly dependent on the time complexity of the objective function. For example, each entry of

the gradient would contain a summation that mirrors the inner summation over k_i in the objective function. While this running time is likely acceptable for many applications, in the next section we explore how to find a close approximation of the optimal solution with an objective function that can be computed in only linear time $O(N)$. In doing so, we remove the dependency on the number of hash functions and eliminate the $O(k_i)$ factor.

4.2 A Faster Relaxation

In this section, we describe a relaxation of the optimization problem that features an approximate objective function. In particular, the objective function is a lower bound of the original objective function from the previous section. When minimizing a lower bound, we are able to simplify and speed up the optimization problem by paying the price of an exact optimal solution. Such lower bound methods are often applied when the original problem is intractable. We find that, while our original problem is certainly tractable, minimizing a lower bound is a solution that benefits both theory and practice.

The Lower Bound. We now derive a lower bound for the objective function. Since Lemma 4.1 states that ϕ is a convex function, we invoke Jensen's inequality $\mathbb{E}[\phi(V)] \geq \phi(\mathbb{E}[V])$ and write the following lower bound for the false positive rate term using the fact that $\mathbb{E}[V] = kp$.

$$\phi(\mathbb{E}[V]) = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^{kp} \tag{4}$$

Hence, we can reformulate the objective function to yield a relaxation by substituting the lower bound in Equation (4) for the original false positive rate term from Equation (3). We can verify that the new problem is a relaxation of original problem, since $\sum_{i=1}^{N} u_i \cdot \phi(\mathbb{E}[V_i]) \leq \sum_{i=1}^{N} u_i \cdot \mathbb{E}[\phi(V_i)]$ and the feasible region remains the same.

Convexity. In a similar fashion to the previous section, we now confirm that the approximate objective function is convex in our optimization variable. This brings us to Proposition 4.3.

Proposition 4.3 (Relaxation Convexity). *The objective function $g(\mathbf{m}') = \sum_{i=1}^{N} u_i \cdot \phi(\mathbb{E}[V_i])$ is convex in $\mathbf{m}'$ on the hyper-rectangle $[0, m_1] \times \ldots \times [0, m_N]$.*

Proof. We show that the desired function can be constructed from a base function $\phi(\cdot)$ that is known to be convex and a convexity-preserving operation. Let $m_i, k_i, n_i > 0$ and $0 \leq m'_i \leq m_i$ for all $i \in [N]$. Define $x_i = k_i p_i$ and compute $\frac{dx_i}{dm'_i} = \frac{k_i}{m_i} > 0$. By Lemma 4.1, the base function $\phi(x_i)$ is convex in x_i since $x_i \geq 0$. This means that its second derivative is non-negative. Then, we have:

$$\frac{d^2\phi(x_i)}{d{m'_i}^2} = \frac{k_i^2}{m_i^2} \cdot \frac{d^2\phi(x_i)}{dx_i^2} \geq 0$$

Therefore, the function $\phi(x_i)$ is convex in m'_i on $[0, m_i]$ since the second derivative with respect to m'_i is non-negative. Finally, the weighted sum of the base functions with non-negative coefficients $u_i \geq 0$ preserves convexity, so we conclude that g is convex in $\mathbf{m}'$ on the hyper-rectangle $[0, m_1] \times \ldots \times [0, m_N]$. □

Complexity Analysis. The key difference in complexity when comparing the lower bound to the original objective function is that the $O(k_i)$ factor in the running time is replaced with $O(1)$ time to compute the binomial expected value $\mathbb{E}[V_i] = k_i p_i$. The overall time complexity for computing the objective function is reduced to linear $O(N)$ with respect to the number of Bloom filters. The run time of many solvers would also be reduced, as was discussed previously.

Further Relaxations. We can further simplify the lower bound in Equation (4) observing that $1 - \left(1 - \frac{1}{m}\right)^{kn} \approx 1 - e^{-kn/m}$ and using the optimal value of k in the Bloom filter, $k = \frac{m}{n} \ln 2$. The objective function becomes $\min_{\mathbf{m}'} \sum_{i=1}^{N} u_i \cdot 0.618^{m'_i/n_i}$, which is also a convex function. We can further simplify the function observing that usually $m'_i \leq 4n_i$, so the function $0.618^{m'_i/n_i}$ can be approximated with the line $-0.213 \frac{m'_i}{n_i} + 1$. Hence, the objective function becomes $\min_{\mathbf{m}'} \sum_{i=1}^{N} -0.213 \cdot \frac{u_i}{n_i} m'_i$. Equivalently, the objective can be written as $\max_{\mathbf{m}'} \sum_{i=1}^{N} \frac{u_i}{n_i} m'_i$. Notice that it is trivial to optimize this function while satisfying the size constraints of the truncated filters. We sort and traverse all Bloom filters in descending order of $\frac{u_i}{n_i}$. For each Bloom filter $\mathcal{B}_i$, we assign the maximum value of $m'_i \leq m_i$ that does not violate the total size constraint. When the total size becomes B, then for all the remaining filters we set $m'_i = 0$. The overall running time of the algorithm is $O(N \log N)$.

5 EVALUATION

In this section, we evaluate our method on two applications and several microbenchmarks. When there is some skew in the utility distributions, the results show that we outperform the state of the art on key metrics across the range of possible memory budgets. Additionally, we find that our method displays minimal performance degradation under very tight budgets.

5.1 Implementation Details

We first give some details about the hardware and software characteristics that form the basis of the evaluation. Experiments were run on a single machine that has an 8 core Apple M3 processor, 8 GB of RAM, and a 512 GB SSD.

The research prototype for our method was written in Python 3.10. The truncated Bloom filters were built using the `bitarray` package [46]. Filters were constructed with the standard procedure that is commonly used in practice. Specifically, the construction algorithm takes in a target false positive rate $0 < \varepsilon < 1$ (in our case $\varepsilon = 1e-4$) and the number of elements $n > 0$ that will be added to the filter. It then calculates an estimate for the optimal filter length m and number of hash functions k given these parameters. We employed the efficient `murmurhash3` hash function family [1].

The optimization code was implemented in `CVXPY` [23], and the optimal solution was found by an embedded conic solver (`ECOS`) [24]. `ECOS` is an interior-point method for second-order cone programs. The relaxation formulation was implemented, since the original objective function contains the non-convex binomial probability mass function, which violates disciplined convex programming (DCP). The implementation additionally featured a budget equality constraint, as well as floor function rounding to map the optimal solution onto the natural numbers.

5.2 Data Skipping

The first application that we evaluate our method on is data skipping [28, 52, 53, 56]. Data skipping is a technique employed by several query engines [2, 13, 17, 18, 42] to reduce read operations, often when tables are stored in a horizontally-partitioned columnar file format [51] such as Apache Parquet. In the conventional case, a partition (i.e., row group) is associated with attribute metadata that enables a query to skip certain partitions when the metadata index reveals that there are no matching tuples. By filtering out irrelevant data with a lightweight index, data skipping can significantly accelerate queries that have highly-selective predicates [54].

A Bloom filter is often the core data structure underlying point queries over categorical attributes. This is because we can make a determination whether at least one tuple that satisfies the predicates is either certainly not in the partition or is likely in the partition with a small probability of being wrong (i.e., false positive). In our evaluation, we associate a Bloom filter with each of the p categorical attributes in a row group. This yields a total of $N \cdot p$ Bloom filters in the full index. Each attribute filter collection is given a space budget that is a fraction of the original collection size (e.g., 20% of the original). The filters are then truncated according to the mean utility of the tuples contained in a row group.

The metric of success for in-memory data skipping with Bloom filters is minimizing the time that is wasted due to false positives. Therefore, the goal in this setting is to produce a truncated Bloom filter index that has only a negligible impact on wasted time and falls within a space budget. The setup for the data skipping experiments is now described in detail.

Datasets. We evaluate our approach on 3 datasets. First, Connecticut real estate sales (RE) contains 997,162 tuples and 2 categorical attributes [14]. Each tuple corresponds to a real estate sale in the State of Connecticut over a 20-year period. Second, Washington electric vehicle registrations (EV) contains 162,637 tuples and 6 categorical attributes [57]. Each tuple is a current electric vehicle registration. Third, NASA HTTP web server logs (NASA) contains 3,452,337 tuples and 1 categorical attribute [30]. Tuples consist of log entries from a web server that hosted the NASA website. Row group size is systematically determined according to the total number of tuples in a table.

Query Model. Every query on a table $\mathcal{T}$ is associated with a set of p equality predicates formed over categorical attributes $A_1, \ldots A_p$ that have the structure $A_i = a_i$. We further restrict our evaluation to the class of conjunctive queries that limit their result set to the cardinality k. The form of the queries is as follows:

```
SELECT * FROM 𝒯
WHERE A_1 = a_1 AND ... AND A_p = a_p
LIMIT k;
```

We assume that during the execution of a query, we visit partitions in decreasing order of utility until k tuples have been added to the result set, or we reach the lowest utility partition.

Query Generation. We produce a workload of 2,500 queries for each dataset that conforms to the structure given above by choosing the most common predicate combinations. For a given limit value of k, we use the workload to compute the utility value of each tuple as its access frequency after a full table scan is run for each query.

5.2.1 *Metrics.* Three key evaluation metrics are defined.

- **Skip rate:** The ratio $\frac{\text{\# skipped}}{\text{\# visited}}$ between the number of partitions that are skipped by the indexes and the total number of partitions that are visited by a query.
- **Wasted time:** The total amount of time that is wasted due to index false positives. Wasted time captures when the index returns a positive result for a partition, and the query result set is then found to be empty. This metric is a proxy for false positive rate but is also correlated with predicate selectivity.
- **Query latency:** The execution time of a query. Query latency includes the time from reading the indexes from disk (if applicable), checking the query against the indexes, reading partitions from disk, and running queries on partitions.

5.2.2 *Baselines.* Each baseline as defined below is implemented or reimplemented in Python to promote a fair comparison.

- **Alphabetical Range (R):** In each partition, the values for an attribute are first sorted in lexicographical order $y_1 \leq y_2 \leq \ldots \leq y_l$. The query attribute value y_q is tested in the predicate $y_1 \leq y_q \leq y_l$. If the predicate evaluates to true, the partition is read from disk and is skipped otherwise.
- **On-disk Filters (D):** Each filter is stored at full resolution (i.e., no truncation) on disk [18]. When a partition is visited, the filter is read into main memory, and the query is checked against the index.
- **Proportional Truncation (PT):** Each filter is truncated according to the policy $m_i' = \lfloor B/F \cdot m_i \rfloor$ where F is the full-resolution (original) size of the index. This approach is an analog of a modular Bloom filter [36], since we are sizing the first module in proportion to the memory budget.
- **Top Utility (TU):** A policy is applied that closely resembles those commonly used for cache eviction (e.g., least recently used or least frequently used) [12]. Full-resolution filters are incrementally added to the cache C in decreasing order of utility until the budget is reached, and then the remaining filters are allocated 0 bits. Formally, this greedy algorithm finds the subset of filters $C \subseteq \{\mathcal{B}_1, \ldots, \mathcal{B}_N\}$ that maximizes $\sum_{c \in C} u_c$ subject to $\sum_{c \in C} m_c \leq B$.
- **Elastic Bloom Filter (EBF):** 7 caches are created. Each cache C_j for $0 \leq j \leq 6$ contains filters that model a different number of filter units via the truncation policy $m_{ij}' = \lfloor j/6 \cdot m_{ij} \rfloor$ [32]. To ensure that the budget B is fully utilized at its larger values, only the $h = \lceil (1 - B/F) \cdot 6 \rceil$ highest-level caches are enabled. Each cache is allocated an equal fraction of the budget $B' = \lfloor B/h \rfloor$. Filters are added from high-level to low-level caches in decreasing order of utility until each cache budget is reached, and the remaining filters are placed in C_0.

5.2.3 *Results.* We measure the performance of each method across the three metrics. For each dataset, we evaluate several budgets that fall in regular increments between 10% and 90% of the original index size. The baselines that guarantee the desired budgets are plotted as curves, while the baselines that have a fixed size are represented as points. We report the median metric for each budget across 10 independent trials.

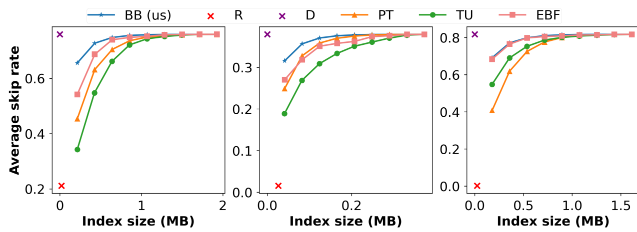

Figure 2: Average skip rate on RE, EV, and NASA at different size budgets (higher is better). We outperform the in-memory methods and are similar to the on-disk filter.

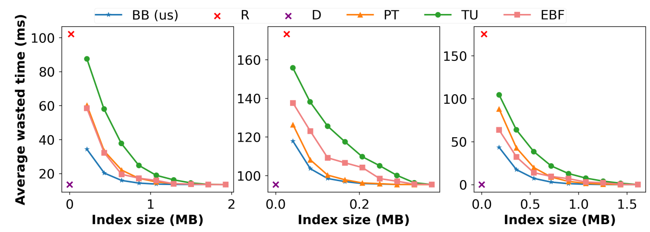

Figure 3: Average wasted time on RE, EV, and NASA at different index size budgets B (lower is better). We outperform the in-memory methods.

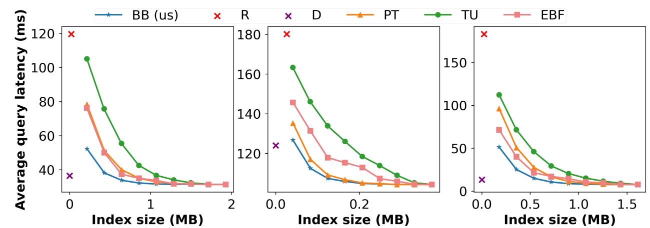

Figure 4: Average query latency on RE, EV, and NASA at different index size budgets B (lower is better). We are generally the lowest latency choice across all methods and budgets.

Proportional Truncation, Top Utility, and Elastic Bloom Filter have distinct drawbacks. First, Proportional Truncation is utility-oblivious in that it has a similar increase in false positive rate across all of the filters in the index. Second, Top Utility is utility-aware but struggles in the setting of long-tailed queries, since it exclusively produces false positives (i.e., cache misses) past a certain point. Third, Elastic Bloom Filter has access to more false positive rate options (i.e., filter lengths) and is utility-aware, but it lacks an optimization algorithm. The goal of our method is to mitigate these drawbacks and thereby maintain satisfactory performance across the range of possible budgets.

Skip Rate. In Figure 2, we see that our method achieves a similar skip rate to the full-resolution filters stored on disk, while improving upon the in-memory baselines. Alphabetical range has a near-zero skip rate, which shows that such a strategy is a poor choice for point queries. For this reason, we will omit further discussion about alphabetical range. Notably, the skip rates of Proportional Truncation and Top Utility fall sharply at budgets of less than 30%, while our method maintains only a modest reduction in skip rate. Elastic Bloom Filter usually has a higher skip rate than Top Utility and Proportional Truncation but is still lower than our method, although the difference is small on the NASA dataset.

Wasted Time. Most importantly, we find in Figure 3 that our method wastes less time on false positives than the other in-memory baselines. In particular, Proportional Truncation and Top Utility both have significant wasted time in scenarios with tight budgets. Elastic Bloom Filter wastes less time than Proportional Truncation and Top Utility on two of the datasets, as it benefits from finer-grained filter lengths. Consider the NASA dataset as a representative example. It is observed that at a 10% budget, our method wastes less time than both Proportional Truncation and Top Utility by a factor of ≈ 2 to 2.5, as well as Elastic Bloom Filter by a factor of ≈ 1.6. The performance advantage holds until Elastic Bloom Filter and Proportional Truncation catch up to our method at a budget of 80% of the original index size. Top Utility also eventually catches up at a budget of 90%.

Furthermore, our method performs better than the alternatives under varying attribute counts and predicate selectivity. Indeed, low predicate selectivity can lead to increased wasted time, a conclusion that can be reached by examining the adversarial EV dataset (middle). The reason for the increased wasted time is because there are many partitions that contain all of the attribute values, even though few tuples jointly satisfy the predicates.

Query Latency. We show in Figure 4 that our method is generally the fastest overall when compared to each of the baselines. The reason for this is two-fold. First, by minimizing the false positive rate in an optimal fashion, we reduce the impact that the failure modes of the in-memory methods have on wasted time, as aforementioned. Second, our index sits in memory, which accelerates our queries in most instances when compared to disk.

Sensitivity Analysis. In Figure 5, we measure the skip rate of our method as we vary the limit value and predicates for the first 100 queries on the RE dataset. Compression ratio (CR) is the space budget as a percentage of the original attribute filter collection size. On the left, we see that as we increase the limit value, the skip rate decreases for smaller budgets. This is due to the reduction in skewness as the utility distribution over the partitions becomes increasingly uniform. At larger budgets, the reduction in skip rate is negligible. On the right, we see that predicates can have different selectivities. As predicates are combined, the probability of a partition jointly satisfying the predicates decreases, leading to a higher skip rate.

Hybrid Memory & Disk Methods. Our method was designed to optimize an in-memory Bloom filter collection (which is usually the case in other systems [4, 19, 31]). However, we can evaluate a hybrid version of our method that has both an optimized in-memory index and access to filters on disk. In this instance, we store the bit positions from the invalid hash functions. Given a positive in-memory result, the truncated region of the filter is read from disk to probe those positions and reduce the false positive rate. In Figure 6, we compare the query latency across RE and NASA for our hybrid version (HBB) and a hybrid version of Top Utility (HTU) that stores the remaining filters on disk.

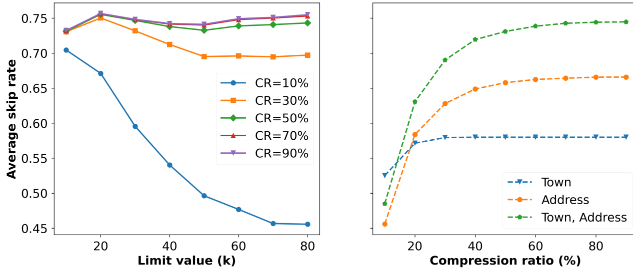

Figure 5: A sensitivity analysis for data skipping.

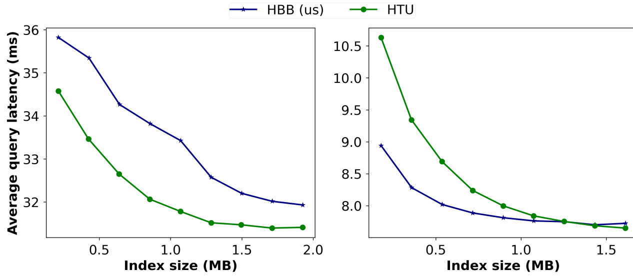

Figure 6: Hybrid memory & disk methods on RE (left) and NASA (right) at different index size budgets.

On the NASA dataset, the large fraction of negative queries (i.e, high predicate selectivity) enables our hybrid version to resolve most of the queries in memory with the optimized index while correcting the false positives in only a few additional filter reads. This leads HBB to outperform HTU at smaller space budgets. For the roughly equal fraction of positive and negative queries in RE, HBB does worse than HTU across the space budgets. On the one hand, as the percentage of negative queries grows, the performance of HBB improves. On the other hand, HTU is consistent across varying fractions of negative queries, since the number of filter reads is only dependent on the space budget. With knowledge of the fraction of negative keys for a particular filter, the problem of optimizing our method for disk accesses is left as future work.

5.3 Full-Text Search

Full-text search is a fundamental problem in information retrieval. In the context of search engines, the task is to find the documents that match the search terms (i.e., tokens), and then rank the matching documents according to some notion of relevance. If we represent each document as a set of tokens, we find that the matching problem is no different than issuing membership queries to the document. Following this intuition, a document can be represented as a Bloom filter (i.e., a signature file). Provided that the Bloom filters are constructed with the same hash functions and filter lengths, we can compute the set intersection between a query filter and a document filter in $O(m)$ time by simply applying a bit-wise AND.

The state-of-the-art approach for full-text search is the inverted index. In its most basic form, an inverted index maps a search term to a list of documents that contain the term. Since documents are represented as fixed-length integer identifiers, the total size of an inverted index grows quickly, even with documents that are rarely returned in a search result. The prevailing space-reduction techniques for inverted indexes are lossless compression [39] and caching [58].

If instead each document is represented as a Bloom filter, we can derive a lossy variable-length signature for each document by directly applying our method. Taking inspiration from BitFunnel [27], a commercial search engine that employs a Bloom filter index in place of an inverted index, we will demonstrate how our method can be used to effectively optimize a collection of Bloom filter signatures according to utility. Our setup for full-text search is different from data skipping, so we will again walk through each of the points individually.

Datasets. We use Amazon review datasets that are from two product categories [37, 38]. The first category is industrial and scientific (I&S) with 49,595 cleaned documents, and the second product category is musical instruments (MI) with 160,523 cleaned documents. We remove punctuation and stopwords before tokenizing the documents. We remove the minority of documents that do not contain between 5 and 100 tokens to produce a realistic document shard [27].

Query Model. For our experiments, document utility values are sampled from a right-skewed mixture of normal distributions. In this setting, we adopt a top-k query model. In particular, the probability that a given document appears in the top-k query results is proportional to its utility. The underlying skewness assumption is that only a small number of documents from the corpus end up appearing in the top-k search results. Similar to data skipping, our method enumerates documents in descending order of utility.

Query Generation. We consider the case of $k = 1$ and generate 2,500 queries for each dataset using the following procedure. Given the utility values $u_1, \ldots, u_N$ we first normalize to a probability distribution like so: $\mathbb{P}_i = u_i / \sum_{j \in [N]} u_j$. To generate a query, we sample a document from the probability distribution and choose n terms from the selected document. The specific terms are chosen so that they are only jointly present in a few documents on average.

5.3.1 Metrics. We begin the discussion of metrics with some notation. Let D be the document corpus. $\mathcal{T}_q$ is defined as the set of tokens in query q and $\mathcal{T}_d$ is the set of tokens in document d. The set R contains the documents retrieved by a particular query. The match set of a query $M = \{d : \mathcal{T}_d \cap \mathcal{T}_q = \mathcal{T}_q\}$ captures the documents that contain every token in the query [27]. Following this definition, the set $G \subseteq M$ represents the ground truth top-k matching documents in terms of utility.

- **Precision@k:** $\frac{|R \cap G|}{k}$. For a Bloom filter index, precision is a salient metric for false positives. For other indexes, precision is higher when there are fewer false negatives.
- **Query latency:** The execution time of a query.

5.3.2 Baselines. The baselines are defined below.

- **Inverted Index (II):** The standard approach [9] where the index maps from a term t to the documents that contain the term such that $t \rightarrow [d_1, d_2, \ldots, d_r]$. Documents are represented as integer identifiers. The size of each document d

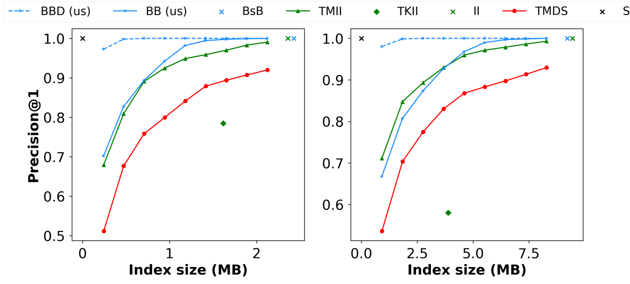

Figure 7: I&S (left) and MI (right). We have higher precision than the best-performing baseline (TMII) at larger budgets and rival its performance at smaller budgets.

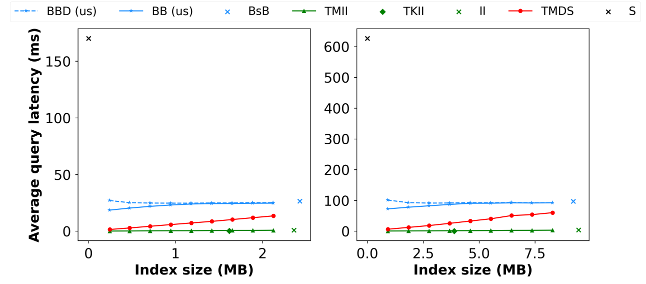

Figure 8: I&S (left) and MI (right). We are slower than the baselines besides scan but are impeded by implementation.

is $O(|\mathcal{T}_d|)$. The size of the index grows quickly, even with low utility documents.

- **Top-M Inverted Index (TMII):** Documents are added to the inverted index until the budget is reached using the same general approach as Top Utility. The greedy algorithm finds the highest-utility subset of documents with cardinality M. Note that a document either exists in the index with all of its terms, or does not exist at all.
- **Top-k Inverted Index (TKII):** The value k is first specified. A full inverted index is constructed using the procedure described above. Each term's document identifier list l is sorted in descending order of utility and then truncated to contain only the top-k highest-utility documents. The key difference from the Top-M Inverted Index is that token false negatives are possible, since only a (potentially empty) subset of the terms for each document is now stored.
- **Top-M Document Set (TMDS):** A forward index is maintained over the M highest-utility documents, where every document is stored as the set of terms $\mathcal{T}_d$. A query enumerates the documents from high to low utility and directly checks if $\mathcal{T}_q \cap \mathcal{T}_d = \mathcal{T}_q$.
- **Scan (S):** Each document is stored in a forward index on disk and is sequentially read at query time. Upon completion of the scan, the matching documents are sorted by utility, and then the top-k are returned.

5.3.3 Results. We select a range of space budgets by first calculating the smallest index size between the inverted index, forward index, and Bloom filter index. The budget is then varied from 10% to 90% of the smallest index size in regular increments, just like in the previous application. We find that the smallest index size is either the inverted index or the Bloom filter index. We evaluate the approaches on in-memory performance (solid lines), but we also include a version of our method that checks an on-disk forward index when the filter returns a positive result (dotted line). We also add a basic Bloom filter index (BsB) that is not truncated as an additional point of comparison. The results for precision and latency across the two datasets are highlighted in Figures 7 and 8 respectively.

In Figure 7, we observe that our method has higher precision than the other in-memory baselines at larger budgets and rivals the Top-M Inverted Index at smaller budgets. It is clear that the Top-M Document Set (forward) index is highly space-inefficient, which results in reduced performance. When comparing the inverted index approaches, it is evident that the Top-M Inverted Index is strictly better than the naive strategy undertaken by the Top-k Inverted Index where the document identifier lists are truncated. At the lone space budget of the Top-k Inverted Index, the Top-M variant is up to 60% higher in terms of precision.

Next, in Figure 8 our method is found to be slower than the other approaches, besides the full scan. The reason for this result is twofold. First, inverted indexes are able to reduce the search space rather significantly, which avoids the need for sequential iteration through the document corpus. This is not so much a surprising result, but it is worth noting that a combination of algorithmic optimizations and careful implementation can be undertaken to make Bloom signatures faster than conventional inverted indexes [27]. Second, the implementation style that we ultimately chose likely had a sizable impact on latency.

5.4 Microbenchmarks

Lastly, we evaluate the main characteristics of our method. In particular, we highlight the truncated Bloom filter and our optimization routine.

5.4.1 Truncated Bloom Filters. As a reminder, the parameter $p = \frac{m'}{m}$ is the ratio between the number of bits that are kept after truncation (m') and original filter size (m). In Figure 9, we show the effect of truncation on both the false positive rate and filter query latency. We generate an equal number of positive q^+ and negative q^- keys (1 million each). The keys are generated in such a way that the intersection between the positive and negative sets is empty.

After adding the positive keys, we query the filter with the negative keys at each truncation ratio, and the false positive rate is measured empirically. We compare the empirical false positive rate to the expected false positive rate from Equation (3) and the lower bound from Equation (4). We observe that given a sufficiently large number of negative queries, the empirical false positive rate (blue) is practically identical to the expected false positive rate (red), which is consistent with the law of large numbers. It is also evident that the gap between the expected false positive rate (red) and the lower bound (green) as given by the function $\mathbb{E}[\phi(V)] - \phi(\mathbb{E}[V])$ converges to 0 as $p \rightarrow 1$. This indicates that the lower bound approximation is better when fewer bits are truncated. Equivalently,

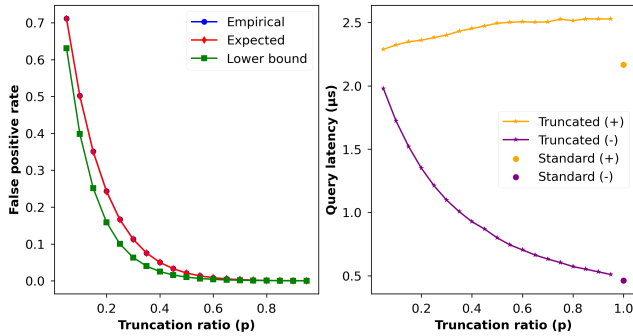

Figure 9: FPR (left) and query latency (right) as a function of the truncation ratio p. Latency is given as 95% confidence intervals, + and − are positive and negative keys respectively.

the optimality gap between the relaxation and original formulation would generally be smaller under larger budgets.

Next, the truncated filter is queried with both negative and positive keys to measure query latency. The result is compared to a standard filter. We see that there are two sources of overhead. First, the extra condition in Algorithm 1 that checks for valid hash functions introduces a small amount of latency overhead when compared to the standard filter, which is seen in the vertical distance between the rightmost truncated measurements and each standard measurement. Second, as the filter is increasingly truncated, the probability p of observing a valid hash function decreases, leading to additional latency overhead in the negative case only. We also see that the latency for the positive case decreases slightly, since fewer filter positions are probed in total.

These results suggest that our proposed data structure is particularly suitable in the high truncation regime $m' \ll m$ for applications where downstream costs far exceed the cost of querying the filter. For example, in one of the applications that we presented, the cost of checking a partition is often several orders of magnitude higher than querying the filter. However, given a collection of truncated Bloom filters with varying utility values, we would expect that the latency overhead is minimal in expectation. This is because a truncated filter that operates in the low truncation regime $m' \approx m$ exhibits only a slight increase in latency on negative queries when compared to the standard filter.

We can also confirm the latency result with a theoretical argument. Let $X \sim G(p)$ be a geometric random variable for the number of hash functions that are checked to observe a valid hash function with mean $\mathbb{E}[X] = \frac{1}{p}$. We find that, in expectation, the number of required probes increases for negatives and stays the same for positives, since only $\Omega(1)$ hash functions are required for a negative while $\Theta(k)$ hash functions are required for a positive.

However, if we treat a truncated Bloom filter as a set signature like in full-text search, then the intersection and union operations are computed in *reduced* time, provided that the filters were constructed with shared hash functions. For example, given two sets S_1 and S_2 represented as truncated Bloom filters, an approximation of $S_1 \cap S_2$ can be found in $O(\min(m_1', m_2'))$ time via bitwise operations.

Table 1: Solving the relaxation scales to large inputs.

N (thousands)	101	301	501	701	901
Optimization latency (s)	2.01	6.72	12.03	19.30	30.65

5.4.2 Optimization Latency. We now evaluate our lower bound optimization approach by measuring the solver latency at a specific budget of 50% and different quantities of Bloom filters. We generate N synthetic parameter tuples of the form (m_i, k_i, n_i) by sampling from a multivariate probability distribution. We then measure the solver latency as the number of Bloom filters N is varied. We report the solver latency as the median across 10 runs with the same input parameters.

As seen in Table 1, the optimization problem can be solved quickly even on a resource-constrained machine. For example, an (approximate) optimal solution can be found for a collection of half a million Bloom filters in around 12 seconds.

We reach the conclusion that the optimization overhead of our method is small, and we can quickly adapt the collection under changing utility values, or as new filters are added. In other words, this is a desirable property in the online case. Crucially, if we store a copy of the original filters on disk, we can re-optimize without having to reconstruct the Bloom filters from scratch.

6 RELATED WORK

Bloom filters in Key-value Stores. The most similar body of work to our method is the application of Bloom filters in key-value stores. Key-value stores are represented as log-structured merge (LSM) trees, where each level of the LSM tree in secondary storage is often associated with in-memory filters. Previous work such as Monkey [19] and ElasticBF [32] has shown that it is suboptimal to uniformly assign Bloom filter false positive rates.

Monkey [19] solves a different optimization problem that is exclusively restricted to LSM trees. For a given target lookup cost (sum of the filter false positive rates), Monkey minimizes the size of the filter collection. In our framework, we instead minimize the utility-weighted false positive rate for a target filter collection size. Our method can model this problem setting by swapping the objective and budget functions, where each utility value is set to 1. Additionally, the analytic expression for the optimal solution in Monkey requires the size ratio between adjacent LSM tree levels as an input parameter, which limits its generality to LSM trees.

Elastic Bloom filter (ElasticBF) [32] extends Monkey by dividing each larger filter into smaller separable filter units (i.e., a filter group) so that more units can be loaded for frequently-accessed filter groups. ElasticBF maintains multiple in-memory LRU queues (a multi-queue [59]) where each queue corresponds to a particular number of filter units. An adjustment scheme is then employed with the goal of decreasing the number of expected false positives by moving filter groups between queues as their access frequencies change. Unlike our method, ElasticBF does not solve an optimization problem.

Modular Bloom Filters. The closest data structure to the truncated Bloom filter is the modular Bloom filter [36] which in turn is similar to the blocked Bloom filter [43] and a filter group [32]. Like

a modular Bloom filter, we employ a smaller in-memory filter at the expense of a higher false positive rate and do not require re-indexing. Unlike a modular Bloom filter, we are able to navigate the space versus accuracy trade-off in a fine-grained fashion, since truncation removes one bit at a time. A modular Bloom filter is limited to the product of false positive rates across several coarse-grained modules that are each associated with a hash function subset. In a sense, our method constructs a two module filter by optimally sizing the first module. However, the optimal first module introduces additional query latency overhead for the truncated variant when compared to the modular variant, since a hash function subset is not mapped to each module.

Fingerprint Filters. In a general fingerprint filter, each element is stored as a bit string in a hash table. Hence, elements can be deleted, in contrast to a Bloom filter where only insertions are allowed. A truncation operation can also be extended to a fingerprint filter.

For example, a cuckoo filter [25] uses a modified version of cuckoo hashing to store the fingerprint of each element. For a given number of elements n, fingerprint size s, and bucket size b, the false positive rate is bounded from above by $1 - \left(1 - \frac{1}{2^s}\right)^{2b}$ [25]. The truncation operation can be extended to a cuckoo filter by removing the $s - s' > 0$ most significant bits for each fingerprint in the filter, where s' is the number of bits we store.

Next, a quotient filter [4, 40, 41] partitions a p-bit fingerprint for an element into its quotient (q most significant bits) and remainder (r least significant bits) i.e., $p = q + r$. The quotient is a bucket in the hash table, and the remainder is stored in a bucket alongside metadata bits to resolve soft collisions from identical quotients. The false positive rate is $1 - \left(1 - \frac{1}{2^p}\right)^n = 1 - \left(1 - \frac{1}{2^{q+r}}\right)^n$ [4]. To truncate a quotient filter, the $r - r'$ most significant bits of the remainder can be removed, where r' is the number of bits we should store. Note that for a quotient filter, truncation is different than contraction. Prior work such as InfiniFilter [20] has shown how quotient filters can be resized. For example, the number of hash table buckets can be halved by moving a bit from the quotient to the remainder and copying over the remainders into a newly-allocated hash table. Truncation instead removes a bit from the remainder to shorten the bit string stored in a bucket.

For a fixed n, we claim that a higher fingerprint filter false positive rate can only be produced with $O(n)$ truncation operations. To truncate a fingerprint filter, we visit each of the n fingerprints stored in the hash table and remove the desired number of bits. In a Bloom filter, we only need $O(1)$ truncation operations to produce a higher false positive rate. Hence, truncation for a Bloom filter is a more efficient operation.

Key-value Filters. Certain types of filters support approximate set membership queries that return values when queried for a key. An early example is the Bloomier filter [11]. Fingerprint key-value filters such as multi-level cuckoo filters [45], Chucky [21], and quotient maplets [15] typically store values in each hash bucket with the fingerprint. While a collection of N filters can be replaced by a single key-value filter that stores the locations that may contain the key, such an approach is not suitable when optimizing for a query distribution across the locations. For example, suppose that a key is encoded as a fingerprint in a quotient filter, and a value is an array of $\log(N)$-bit integer codes that represents multiple locations where the key may exist. To reduce the space occupied by a rarely-queried location, an integer code would have to be removed from a value array, which may introduce a false negative.

Learned Filters. There are several filter variants that incorporate information from a query distribution to achieve a better trade-off between space and false positive rate than a standard filter. One variant constructs a stacked filter representation to encode both positive and frequent negative keys in alternating filter layers [22]. Another variant models a filter as a binary classification task that returns a positive result if the model output probability is above a certain threshold, which means that predictable positive elements are "stored" in the model [29]. If the output probability is below the threshold, a standard filter containing the less predictable elements is queried to prevent false negatives. Learned filters have been extended to unbounded data streams, where a non-zero false negative rate must be tolerated in order to enforce a false positive rate guarantee [33]. Learned filters optimize the trade-off between space and false positive rate for a single filter by using a query distribution over the keys. Our method instead optimizes this trade-off for a collection of multiple filters by using a query distribution over the collection.

Other Bloom Filter Applications. There are many applications of Bloom filters besides those that we presented in this paper. As one example, Bloom filters are often used to optimize joins [26, 48, 55, 60]. As another example, Bloom filters are a vital data structure for in-memory data stores such as Redis [44]. In computational biology, binary trees with compressed Bloom filter nodes are used to efficiently search for sets that contain certain subsequences [50]. This approach turns each Bloom filter into a succinct data structure, whereas we truncate the filter to reduce space.

7 CONCLUSION

We presented an approach that optimizes a collection of Bloom filters across a utility distribution to satisfy a space budget. It was shown that truncating a Bloom filter after construction can lead to a subtly different trade-off between accuracy and space that benefits optimization. Fittingly, we developed a convex optimization problem that admits a fast relaxation to navigate this trade-off. Moving from theory to practice, we demonstrated on two separate applications that our method effectively mitigates the limitations of existing approaches. A possible direction of future work is to extend our optimization problem to different types of filter and sketch collections. It may also hold promise to simultaneously optimize for query distributions present in a single such data structure.

ACKNOWLEDGMENTS

This research was performed in part with support from the University of Chicago, Center for Unstoppable Computing (CERES), and the Institute for Translational Medicine Sociome Project.

REFERENCES

[1] Austin Appleby. 2015. murmurhash3. Retrieved January 9, 2024 from https://github.com/aappleby/smhasher/blob/master/src/MurmurHash3.cpp

[2] Michael Armbrust, Tathagata Das, Liwen Sun, Burak Yavuz, Shixiong Zhu, Mukul Murthy, Joseph Torres, Herman van Hovell, Adrian Ionescu, Alicja Łuszczak, Michał Świtakowski, Michał Szafranski, Xiao Li, Takuya Ueshin, Mostafa Mokhtar, Peter Boncz, Ali Ghodsi, Sameer Paranjpye, Pieter Senster,

Reynold Xin, and Matei Zaharia. 2020. Delta lake: high-performance ACID table storage over cloud object stores. *Proc. VLDB Endow.* 13, 12 (aug 2020), 3411–3424. https://doi.org/10.14778/3415478.3415560

[3] Bruno Barbarioli, Gabriel Mersy, Stavros Sintos, and Sanjay Krishnan. 2023. Hierarchical Residual Encoding for Multiresolution Time Series Compression. *Proc. ACM Manag. Data* 1, 1, Article 99 (may 2023), 26 pages. https://doi.org/10.1145/3588953

[4] Michael A. Bender, Martin Farach-Colton, Rob Johnson, Russell Kraner, Bradley C. Kuszmaul, Dzejla Medjedovic, Pablo Montes, Pradeep Shetty, Richard P. Spillane, and Erez Zadok. 2012. Don't thrash: how to cache your hash on flash. *Proc. VLDB Endow.* 5, 11 (jul 2012), 1627–1637. https://doi.org/10.14778/2350229.2350275

[5] Benjamin Berg, Daniel S. Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, and Gregory R. Ganger. 2020. The CacheLib Caching Engine: Design and Experiences at Scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Berkeley, CA, USA, 753–768. https://www.usenix.org/conference/osdi20/presentation/berg

[6] 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

[7] Prosenjit Bose, Hua Guo, Evangelos Kranakis, Anil Maheshwari, Pat Morin, Jason Morrison, Michiel Smid, and Yihui Tang. 2008. On the false-positive rate of Bloom filters. *Inform. Process. Lett.* 108, 4 (2008), 210–213. https://doi.org/10.1016/j.ipl.2008.05.018

[8] Stephen P Boyd and Lieven Vandenberghe. 2004. *Convex optimization*. Cambridge university press, New York, NY, USA.

[9] Sergey Brin and Lawrence Page. 1998. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems* 30, 1 (1998), 107–117. https://doi.org/10.1016/S0169-7552(98)00110-X Proceedings of the Seventh International World Wide Web Conference.

[10] Lawrence D. Brown. 1986. Fundamentals of Statistical Exponential Families with Applications in Statistical Decision Theory. *Lecture Notes-Monograph Series* 9 (1986), i–279. http://www.jstor.org/stable/4355554

[11] Bernard Chazelle, Joe Kilian, Ronitt Rubinfeld, and Ayellet Tal. 2004. The Bloomier filter: an efficient data structure for static support lookup tables. In *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms* (New Orleans, Louisiana) *(SODA '04)*. Society for Industrial and Applied Mathematics, USA, 30–39.

[12] Hong-Tai Chou and David J. DeWitt. 1985. An Evaluation of Buffer Management Strategies for Relational Database Systems. In *Proceedings of the 11th International Conference on Very Large Data Bases - Volume 11 (VLDB '85)*. VLDB Endowment, Stockholm, Sweden, 127–141.

[13] Clickhouse. 2024. Understanding ClickHouse Data Skipping Indexes. Retrieved January 9, 2024 from https://clickhouse.com/docs/en/optimize/skipping-indexes

[14] State of Connecticut. 2023. Real Estate Sales 2001-2020 GL. Retrieved January 10, 2024 from https://catalog.data.gov/dataset/real-estate-sales-2001-2018

[15] Alex Conway, Martín Farach-Colton, and Rob Johnson. 2023. SplinterDB and Maplets: Improving the Tradeoffs in Key-Value Store Compaction Policy. *Proc. ACM Manag. Data* 1, 1, Article 46 (may 2023), 27 pages. https://doi.org/10.1145/3588726

[16] Graham Cormode and S. Muthukrishnan. 2005. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms* 55, 1 (2005), 58–75. https://doi.org/10.1016/j.jalgor.2003.12.001

[17] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. 2016. The Snowflake Elastic Data Warehouse. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 215–226. https://doi.org/10.1145/2882903.2903741

[18] Databricks. 2023. Bloom filter indexes. Retrieved January 9, 2024 from https://docs.databricks.com/en/optimizations/bloom-filters.html

[19] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2017. Monkey: Optimal Navigable Key-Value Store. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 79–94. https://doi.org/10.1145/3035918.3064054

[20] Niv Dayan, Ioana Bercea, Pedro Reviriego, and Rasmus Pagh. 2023. InfiniFilter: Expanding Filters to Infinity and Beyond. *Proc. ACM Manag. Data* 1, 2, Article 140 (jun 2023), 27 pages. https://doi.org/10.1145/3589285

[21] Niv Dayan and Moshe Twitto. 2021. Chucky: A Succinct Cuckoo Filter for LSM-Tree. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 365–378. https://doi.org/10.1145/3448016.3457273

[22] Kyle Deeds, Brian Hentschel, and Stratos Idreos. 2020. Stacked Filters: Learning to Filter by Structure. *Proc. VLDB Endow.* 14, 4 (dec 2020), 600–612. https://doi.org/10.14778/3436905.3436919

[23] Steven Diamond and Stephen Boyd. 2016. CVXPY: a python-embedded modeling language for convex optimization. *J. Mach. Learn. Res.* 17, 1 (jan 2016), 2909–2913. https://dl.acm.org/doi/10.5555/2946645.3007036

[24] Alexander Domahidi, Eric Chu, and Stephen Boyd. 2013. ECOS: An SOCP solver for embedded systems. In *2013 European Control Conference (ECC)*. Institute of Electrical and Electronics Engineers, New York, NY, USA, 3071–3076. https://doi.org/10.23919/ECC.2013.6669541

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

[26] Kevin P. Gaffney, Martin Prammer, Larry Brasfield, D. Richard Hipp, Dan Kennedy, and Jignesh M. Patel. 2022. SQLite: past, present, and future. *Proc. VLDB Endow.* 15, 12 (aug 2022), 3535–3547. https://doi.org/10.14778/3554821.3554842

[27] Bob Goodwin, Michael Hopcroft, Dan Luu, Alex Clemmer, Mihaela Curmei, Sameh Elnikety, and Yuxiong He. 2017. BitFunnel: Revisiting Signatures for Search. In *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval* (Shinjuku, Tokyo, Japan) *(SIGIR '17)*. Association for Computing Machinery, New York, NY, USA, 605–614. https://doi.org/10.1145/3077136.3080789

[28] Andreas Kipf, Damian Chromejko, Alexander Hall, Peter Boncz, and David G. Andersen. 2020. Cuckoo index: a lightweight secondary index structure. *Proc. VLDB Endow.* 13, 13 (sep 2020), 3559–3572. https://doi.org/10.14778/3424573.3424577

[29] 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

[30] Lawrence Berkeley National Laboratory. 1995. NASA-HTTP. Retrieved January 10, 2024 from https://ita.ee.lbl.gov/html/contrib/NASA-HTTP.html

[31] 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)*. IEEE Computer Society Offices, Washington, DC, USA, 26–37. https://doi.org/10.1109/ICDE.2013.6544811

[32] Yongkun Li, Chengjin Tian, Fan Guo, Cheng Li, and Yinlong Xu. 2019. ElasticBF: Elastic Bloom Filter with Hotness Awareness for Boosting Read Performance in Large Key-Value Stores. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 739–752. https://www.usenix.org/system/files/atc19-li-yongkun.pdf

[33] Qiyu Liu, Libin Zheng, Yanyan Shen, and Lei Chen. 2020. Stable learned bloom filters for data streams. *Proc. VLDB Endow.* 13, 12 (jul 2020), 2355–2367. https://doi.org/10.14778/3407790.3407830

[34] Michael Mitzenmacher. 2002. Compressed Bloom filters. *IEEE/ACM Transactions on Networking* 10, 5 (2002), 604–612. https://doi.org/10.1109/TNET.2002.803864

[35] Michael Mitzenmacher and Andrei Broder. 2004. Network Applications of Bloom Filters: A Survey. *Internet Mathematics* 1, 4 (1 1 2004), 485–509. https://doi.org/10.1080/15427951.2004.10129096

[36] Ju Hyoung Mun, Zichen Zhu, Aneesh Raman, and Manos Athanassoulis. 2022. LSM-Trees Under (Memory) Pressure. In *Thirteenth International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures (ADMS 2022)*. Sydney, Australia, 23–35. https://www.adms-conf.org/2022-camera-ready/ADMS22_mun.pdf

[37] Jianmo Ni. 2018. Amazon Review Data (2018). Retrieved January 10, 2024 from https://cseweb.ucsd.edu/~jmcauley/datasets/amazon_v2/

[38] Jianmo Ni, Jiacheng Li, and Julian McAuley. 2019. Justifying Recommendations using Distantly-Labeled Reviews and Fine-Grained Aspects. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, Kentaro Inui, Jing Jiang, Vincent Ng, and Xiaojun Wan (Eds.). Association for Computational Linguistics, Hong Kong, China, 188–197. https://doi.org/10.18653/v1/D19-1018

[39] 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

[40] Prashant Pandey, Michael A. Bender, Rob Johnson, and Rob Patro. 2017. A General-Purpose Counting Filter: Making Every Bit Count. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 775–787. https://doi.org/10.1145/3035918.3035963

[41] Prashant Pandey, Alex Conway, Joe Durie, Michael A. Bender, Martin Farach-Colton, and Rob Johnson. 2021. Vector Quotient Filters: Overcoming the Time/Space Trade-Off in Filter Design. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD*

'21). Association for Computing Machinery, New York, NY, USA, 1386–1399. https://doi.org/10.1145/3448016.3452841
[42] PostgreSQL. 2024. F.7. bloom — bloom filter index access method. Retrieved January 9, 2024 from https://www.postgresql.org/docs/current/bloom.html
[43] Felix Putze, Peter Sanders, and Johannes Singler. 2010. Cache-, hash-, and space-efficient bloom filters. *ACM J. Exp. Algorithmics* 14, Article 4 (jan 2010), 18 pages. https://doi.org/10.1145/1498698.1594230
[44] Redis. 2024. Bloom filter. Retrieved January 9, 2024 from https://redis.io/docs/data-types/probabilistic/bloom-filter/
[45] Kai Ren, Qing Zheng, Joy Arulraj, and Garth Gibson. 2017. SlimDB: a space-efficient key-value storage engine for semi-sorted data. *Proc. VLDB Endow.* 10, 13 (sep 2017), 2037–2048. https://doi.org/10.14778/3151106.3151108
[46] Ilan Schnell. 2023. bitarray. Retrieved January 9, 2024 from https://pypi.org/project/bitarray/
[47] Tore Schweder. 1982. On the Dispersion of Mixtures. *Scandinavian Journal of Statistics* 9, 3 (1982), 165–169. http://www.jstor.org/stable/4615873
[48] Amazon Web Services. 2020. Amazon Redshift now leverages Bloom filters to improve data lake query performance by up to 2x. Retrieved January 9, 2024 from https://aws.amazon.com/about-aws/whats-new/2020/05/amazon-redshift-now-leverages-bloom-filters-to-improve-data-lake-query-performance/
[49] Moshe Shaked. 1980. On Mixtures from Exponential Families. *Journal of the Royal Statistical Society. Series B (Methodological)* 42, 2 (1980), 192–198. http://www.jstor.org/stable/2984960
[50] Brad Solomon and Carl Kingsford. 2016. Fast search of thousands of short-read sequencing experiments. *Nature Biotechnology* 34, 3 (2016), 300.
[51] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. 2005. C-store: a column-oriented DBMS. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB '05)*. VLDB Endowment, Trondheim, Norway, 553–564. https://dl.acm.org/doi/10.5555/1083592.1083658
[52] Sivaprasad Sudhir, Wenbo Tao, Nikolay Laptev, Cyrille Habis, Michael Cafarella, and Samuel Madden. 2023. Pando: Enhanced Data Skipping with Logical Data Partitioning. *Proc. VLDB Endow.* 16, 9 (may 2023), 2316–2329. https://doi.org/10.14778/3598581.3598601
[53] Liwen Sun, Michael J. Franklin, Sanjay Krishnan, and Reynold S. Xin. 2014. Fine-Grained Partitioning for Aggressive Data Skipping. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (Snowbird, Utah, USA) *(SIGMOD '14)*. Association for Computing Machinery, New York, NY, USA, 1115–1126. https://doi.org/10.1145/2588555.2610515
[54] Paula Ta-Shma, Guy Khazma, Gal Lushi, and Oshrit Feder. 2020. Extensible Data Skipping. In *2020 IEEE International Conference on Big Data (Big Data)*. IEEE Computer Society, Los Alamitos, CA, USA, 372–382. https://doi.org/10.1109/BigData50022.2020.9377740
[55] Daniel Ting and Rick Cole. 2021. Conditional Cuckoo Filters. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 1838–1850. https://doi.org/10.1145/3448016.3452811
[56] Yulai Tong, Jiazhen Liu, Hua Wang, Ke Zhou, Rongfeng He, Qin Zhang, and Cheng Wang. 2023. Sieve: A Learned Data-Skipping Index for Data Analytics. *Proc. VLDB Endow.* 16, 11 (jul 2023), 3214–3226. https://doi.org/10.14778/3611479.3611520
[57] State of Washington. 2023. Electric Vehicle Population Data. Retrieved January 10, 2024 from https://catalog.data.gov/dataset/electric-vehicle-population-data
[58] Jiangong Zhang, Xiaohui Long, and Torsten Suel. 2008. Performance of Compressed Inverted List Caching in Search Engines. In *Proceedings of the 17th International Conference on World Wide Web* (Beijing, China) *(WWW '08)*. Association for Computing Machinery, New York, NY, USA, 387–396. https://doi.org/10.1145/1367497.1367550
[59] Yuanyuan Zhou, James Philbin, and Kai Li. 2001. The Multi-Queue Replacement Algorithm for Second Level Buffer Caches. In *Proceedings of the General Track: 2001 USENIX Annual Technical Conference*. USENIX Association, USA, 91–104.
[60] Jianqiao Zhu, Navneet Potti, Saket Saurabh, and Jignesh M. Patel. 2017. Looking ahead makes query plans robust: making the initial case with in-memory star schema data warehouse workloads. *Proc. VLDB Endow.* 10, 8 (apr 2017), 889–900. https://doi.org/10.14778/3090163.3090167