Practical Dynamic Extension for Sampling Indexes

DOUGLAS B. RUMBAUGH, The Pennsylvania State University, USA
DONG XIE, The Pennsylvania State University, USA

The execution of analytical queries on massive datasets presents challenges due to long response times and high computational costs. As a result, the analysis of representative samples of data has emerged as an attractive alternative; this avoids the cost of processing queries against the entire dataset, while still producing statistically valid results. Unfortunately, the sampling techniques in common use sacrifice either sample quality or performance, and so are poorly suited for this task. However, it is possible to build high quality sample sets efficiently with the assistance of indexes. This introduces a new challenge: real-world data is subject to continuous update, and so the indexes must be kept up to date. This is difficult, because existing sampling indexes present a dichotomy; efficient sampling indexes are difficult to update, while easily updatable indexes have poor sampling performance. This paper seeks to address this gap by proposing a general and practical framework for extending most sampling indexes with efficient update support, based on splitting indexes into smaller shards, combined with a systematic approach to the periodic reconstruction. The framework's design space is examined, with an eye towards exploring trade-offs between update performance, sampling performance, and memory usage. Three existing static sampling indexes are extended using this framework to support updates, and the generalization of the framework to concurrent operations and larger-than-memory data is discussed. Through a comprehensive suite of benchmarks, the extended indexes are shown to match or exceed the update throughput of state-of-the-art dynamic baselines, while presenting significant improvements in sampling latency.

CCS Concepts: • **Information systems → Data structures**.

Additional Key Words and Phrases: Sampling Indexes, Dynamic Extension, Independent Range Sampling

ACM Reference Format:
Douglas B. Rumbaugh and Dong Xie. 2023. Practical Dynamic Extension for Sampling Indexes. 1, 1 (October 2023), 26 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The collection and storage of vast amounts of data has grown progressively easier, while critical decision making processes have become increasingly data-driven in their approach. At the convergence of these trends, an interest in the execution of analytical queries against massive datasets has emerged. Unfortunately, such queries require vast computational resources and have lengthy response times. An attractive technique for resolving this difficulty is to analyze a smaller *representative sample* of the data, thereby avoiding the overhead of processing the dataset in full. An extensive body of work has developed around this technique, including approximate query processing (AQP) [7, 15, 30, 43], interactive data exploration [22, 50], financial audit sampling [36], and feature selection for machine learning [29]. However, sampling for analysis is non-trivial; samples must be independent and identically distributed to ensure statistical validity [14], representativeness,

Authors' addresses: Douglas B. Rumbaugh, The Pennsylvania State University, University Park, Pennsylvania, USA, drumbaugh@psu.edu; Dong Xie, The Pennsylvania State University, University Park, Pennsylvania, USA, dongx@psu.edu.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2023 Association for Computing Machinery.
XXXX-XXXX/2023/10-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

and fairness [46]. When sampling from the result set of a database query (often called independent query sampling) inter-query independence is also important [27]. The sampling techniques used in practice either sacrifice these properties, or pay a sufficiently large cost in their maintenance to render them of limited utility in analytical contexts. For example, SQL's `TABLESAMPLE` operator is often implemented using Bernoulli sampling. This technique requires the processing of the query in full, which negates most of the performance benefits of sampling [25]. More efficient techniques, such as block or systematic sampling (used in Postgres [3]), fail to preserve independence [25]. Reservoir sampling, which produces pre-collected offline samples [7, 22, 43, 47], is not robust against changes in data distribution.

Ideally, an independent query sampling technique would both maintain statistical validity *and* avoid the overhead of query execution against the entire dataset. The above techniques fail to satisfy both of these requirements, but both can be achieved with the help of indexes. There exist two main approaches for index-assisted sampling; namely, the use of data structures designed specifically for efficient sampling [46] and the use of randomized traversals of search trees to sample records [36]. Both of these techniques maintain statistical validity while avoiding processing the full database query, but they each are associated with a significant limitation.

The former approach is appealing because the cost of drawing sample sets using these specialized indexes is often near-constant with respect to the dataset size. After paying a one-time cost per query, samples can be drawn in constant time. These indexes are predominately based upon Walker's alias structure [48, 49], which is a data structure that supports drawing independent samples from a set of weighted items in constant time. Unfortunately, adding or removing a single item from an alias structure requires a complete reconstruction. Because real-world datasets are constantly changing and timely analytical results require that samples reflect the current state of the data, these sampling indexes are not practical for most systems. Tree-traversal based sampling provides an alternative approach. Randomized tree-traversals are combined with rejection sampling to select records for inclusion in the sample set [36]. This allows for updates, but requires a full tree-traversal for each record sampled.

There exist a wide range of complex indexes that facilitate independent sampling [5, 6, 27, 50], but a recent survey [46] has shown that nearly all of them rely on these two techniques. Thus, the selection of a sampling index presents an apparent dichotomy: indexes based on alias structures achieve superior sampling performance, but face difficulty handling updates; tree-traversal based approaches enable support for efficient updates, but exhibit worse sampling performance.

This paper seeks to overcome the dichotomous relationship between the update support and sampling performance of index-assisted sampling techniques by proposing a general and practical framework that is capable of extending most static sampling indexes with efficient support for updates. This framework is based upon the principle of dividing a static index into smaller pieces, which can be systematically reconstructed to support data updates. It draws inspiration from the Bentley-Saxe method [42, 44] and the well-studied design space of the LSM tree [11, 17–19, 21], with the aim of achieving efficient update support while maintaining a sampling performance advantage over tree-traversal based techniques. The key contributions of this paper are,

- The creation of a general framework for extending static sampling indexes with support for updates, while maintaining efficient sampling performance. In particular, three concrete examples of the application of the framework to existing sampling indexes are proposed.
- An exploration of the design space of the framework, discussing trade-offs between update performance, sampling performance, and memory usage.
- A discussion of the generalization of the framework to support larger-than-memory data and concurrent updates.

- A comprehensive evaluation of the trade-offs within the framework's design space, as well as an evaluation of the performance of the framework compared to existing dynamic sampling solutions.

The remainder of the paper is structured as follows: In Section 2, the sampling problem is formalized, existing solutions described, and the dichotomous relationship between update support and sampling performance for sampling indexes discussed. Next, a general framework for extending static sampling indexes with support for updates, inspired the Bentley-Saxe method [44] and the LSM tree [40] is proposed in Section 3. Section 4 demonstrates specific instantiations of the framework to extend three static sampling indexes with support for updates, and formally analyzes the resulting update and sampling costs. Larger-than-memory and concurrent extensions are discussed in Section 5. A comprehensive evaluation and exploration of the design space of the framework and comparison against existing solutions is presented in Section 6. Finally, Section 7 provides a more detailed connection to related works, and Section 8 concludes the paper.

2 BACKGROUND

This section formalizes the sampling problem, describes relevant existing solutions, and presents the dichotomy between update support and sampling performance for sampling indexes. Before discussing these topics, though, a clarification of definition is in order. The nomenclature used to describe sampling varies slightly throughout the literature. In this paper, the term *sample* is used to indicate a single record selected by a sampling operation, and a collection of these samples is called a *sample set*; the number of samples within a sample set is the *sample size*. The term *sampling* is used to indicate the selection of either a single sample or a sample set; the specific usage should be clear from context.

Independent Sampling Problem. When conducting sampling, it is often desirable for the drawn samples to have *statistical independence*. This requires that the sampling of a record does not affect the probability of any other record being sampled in the future. Independence is a requirement for the application of statistical tools such as the Central Limit Theorem [14], which is the basis for many concentration bounds. A failure to maintain independence in sampling invalidates any guarantees provided by these statistical methods.

In each of the problems considered, sampling can be performed either with replacement (WR) or without replacement (WoR). It is possible to answer any WoR sampling query using a constant number of WR queries, followed by a deduplication step [28], and so this paper focuses exclusively on WR sampling.

A basic version of the independent sampling problem is *weighted set sampling* (WSS),[1] in which each record is associated with a weight that determines its probability of being sampled. More formally, WSS is defined as:

Definition 1 (Weighted Set Sampling [49]). *Let D be a set of data whose members are associated with positive weights $w : D \longrightarrow \mathbb{R}^+$. Given an integer $k \geq 1$, a weighted set sampling query returns k independent random samples from D with each data point $d \in D$ having a probability of $\frac{w(d)}{\sum_{p \in D} w(p)}$ of being sampled.*

Each query returns a sample set of size k, rather than a single sample. Queries returning sample sets are the common case, because the robustness of analysis relies on having a sufficiently large sample size [12]. The common *simple random sampling* (SRS) problem is a special case of WSS, where every element has unit weight.

[1]This nomenclature is adopted from Tao's recent survey of sampling techniques [46]. This problem is also called *weighted random sampling* (WRS) in the literature.

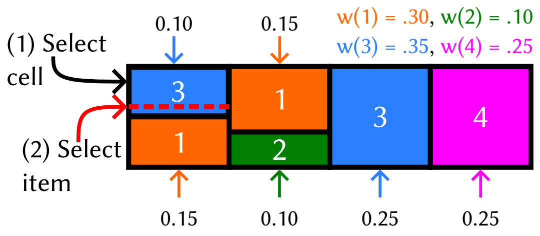


Fig. 1. A pictorial representation of an alias structure, built over a set of weighted records. Sampling is performed by first (1) selecting a cell by uniformly generating an integer index on $[0, n)$, and then (2) selecting an item by generating a second uniform float on $[0, 1]$ and comparing it to the cell's normalized cutoff values. In this example, the first random number is 0, corresponding to the first cell, and the second is .7. This is larger than $.15/.25$, and so 3 is selected as the result of the query. This allows $O(1)$ independent weighted set sampling, but adding a new element requires a weight adjustment to every element in the structure, and so isn't generally possible without performing a full reconstruction.

In the context of databases, it is also common to discuss a more general version of the sampling problem, called *independent query sampling* (IQS) [27]. An IQS query samples a specified number of records from the result set of a database query. In this context, it is insufficient to merely ensure individual records are sampled independently; the sample sets returned by repeated IQS queries must be independent as well. This provides a variety of useful properties, such as fairness and representativeness of query results [46]. As a concrete example, consider simple random sampling on the result set of a single-dimensional range reporting query. This is called independent range sampling (IRS), and is formally defined as:

Definition 2 (Independent Range Sampling [46]). *Let D be a set of n points in $\mathbb{R}$. Given a query interval $q = [x, y]$ and an integer k, an independent range sampling query returns k independent samples from $D \cap q$ with each point having equal probability of being sampled.*

A generalization of IRS exists, called *Weighted Independent Range Sampling* (WIRS) [6], which is similar to WSS. Each point in D is associated with a positive weight $w : D \to \mathbb{R}^+$, and samples are drawn from the range query results $D \cap q$ such that each data point has a probability of $w(d)/\sum_{p \in D \cap q} w(p)$ of being sampled.

Existing Solutions. While many sampling techniques exist, few are supported in practical database systems. The existing `TABLESAMPLE` operator provided by SQL in all major DBMS implementations [3] requires either a linear scan (e.g., Bernoulli sampling) that results in high sample retrieval costs, or relaxed statistical guarantees (e.g., block sampling [3] used in PostgreSQL).

Index-assisted sampling solutions have been studied extensively. Olken's method [38] is a classical solution to independent sampling problems. This algorithm operates upon traditional search trees, such as the B+tree used commonly as a database index. It conducts a random walk on the tree uniformly from the root to a leaf, resulting in a $O(\log n)$ sampling cost for each returned record. Should weighted samples be desired, rejection sampling can be performed. A sampled record, r, is accepted with probability $w(r)/w_{max}$, with an expected number of w_{max}/w_{avg} samples to be taken per element in the sample set. Olken's method can also be extended to support general IQS by rejecting all sampled records failing to satisfy the query predicate. It can be accelerated by adding aggregated weight tags to internal nodes [36, 53], allowing rejection sampling to be performed during the tree-traversal to abort dead-end traversals early.

There also exist static data structures, referred to in this paper as static sampling indexes (SSIs), that are capable of answering sampling queries in near-constant time[2] relative to the size of the dataset. An example of such a structure is used in Walker's alias method [48, 49], a technique for answering WSS queries with $O(1)$ query cost per sample, but requiring $O(n)$ time to construct. It distributes the weight of items across n cells, where each cell is partitioned into at most two items, such that the total proportion of each cell assigned to an item is its total weight. A query selects one cell uniformly at random, then chooses one of the two items in the cell by weight; thus, selecting items with probability proportional to their weight in $O(1)$ time. A pictorial representation of this structure is shown in Figure 1.

The alias method can also be used as the basis for creating SSIs capable of answering general IQS queries using a technique called alias augmentation [46]. As a concrete example, previous papers [6, 46] have proposed solutions for WIRS queries using $O(\log n + k)$ time, where the $\log n$ cost is only be paid only once per query, after which elements can be sampled in constant time. This structure is built by breaking the data up into disjoint chunks of size $n/\log n$, called *fat points*, each with an alias structure. A B+tree is then constructed, using the fat points as its leaf nodes. The internal nodes are augmented with an alias structure over the total weight of each child. This alias structure is used instead of rejection sampling to determine the traversal path to take through the tree, and then the alias structure of the fat point is used to sample a record. Because rejection sampling is not used during the traversal, two traversals suffice to establish the valid range of records for sampling, after which samples can be collected without requiring per-sample traversals. More examples of alias augmentation applied to different IQS problems can be found in a recent survey by Tao [46].

There do exist specialized sampling indexes [27] with both efficient sampling and support for updates, but these are restricted to specific query types and are often very complex structures, with poor constant factors associated with sampling and update costs, and so are of limited practical utility. There has also been work [8, 26, 34] on extending the alias structure to support weight updates over a fixed set of elements. However, these solutions do not allow insertion or deletion in the underlying dataset, and so are not well suited to database sampling applications.

The Dichotomy. Among these techniques, there exists a clear trade-off between efficient sampling and support for updates. Tree-traversal based sampling solutions pay a dataset size based cost per sample, in exchange for update support. The static solutions lack support for updates, but support near-constant time sampling. While some data structures exist with support for both, these are restricted to highly specialized query types. Thus in the general case there exists a dichotomy: existing sampling indexes can support either data updates or efficient sampling, but not both.

3 DYNAMIC SAMPLING INDEX FRAMEWORK

This paper is an attempt to achieve the best of both worlds: the construction of a general sampling index capable of reasonably efficient updates whilst maintaining a near-constant cost per sample. Creating updatable indexes from scratch is necessarily problem-specific, and so instead a more general solution has been developed. Existing static indexes are extended with support for updates by breaking them into smaller structures and systematically reconstructing these structures to included updated records.

In practice, it is common to update static indexes with new data by occasionally reconstructing them during periods of low system utilization. While this ensures that updates will eventually appear in the index, it introduces a freshness problem. New data will not be available until the

[2]The designation "near-constant" is *not* used in the technical sense of being constant to within a polylogarithmic factor (i.e., $\tilde{O}(1)$). It is instead used to mean constant to within an additive polylogarithmic term, i.e., $f(x) \in O(\log n + 1)$.

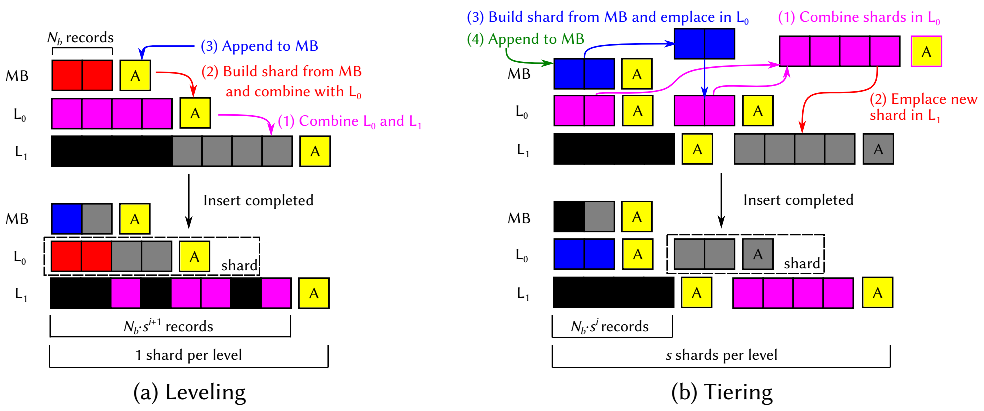
index is next rebuilt. This can be ameliorated by increasing the frequency of reconstruction, thereby reducing the delay before new updates are present within the index. Pushing this approach to its natural conclusion, a full reconstruction on each update would ensure constant availability of the latest data. Unfortunately, this naïve strategy is too costly to be generally viable.

A more systematic approach to index reconstruction is provided by the Bentley-Saxe method and its derivatives [42, 44]. This method partitions the dataset into multiple disjoint shards, and builds data structures over each shard. The insertion or removal of an element only requires the reconstruction of one of these shards, reducing average reconstruction costs. However, this method can only be applied to data structures which answer a *decomposable search problem* (DSP). A query is a DSP if there is an efficient result merge operation, such that querying each shard and merging the results together produces the same result as would have been achieved by querying the entire dataset as a whole. For performance reasons, Bentley-Saxe requires that the merge operation be constant time, but this is not a correctness requirement and more general DSP definitions exist without it [41]. Unfortunately, the concept of decomposability is not cleanly applicable to sampling queries, because the distribution of records in the result set, rather than the records themselves, must be matched following the result merge. Efficiently controlling the distribution requires each sub-query to access information external to the shard against which it is being processed, a contingency unaccounted for by Bentley-Saxe. Further, the process of reconstruction used in Bentley-Saxe provides poor worst-case complexity bounds [44], and attempts to modify the procedure to provide better worst-case performance are complex and have worse performance in the common case [42]. Despite these limitations, this paper will argue that sharding can be profitably applied to sampling indexes, once a system for controlling result set distributions and a more effective shard reconstruction scheme have been devised. The solution to the former will be discussed in Section 3.3. For the latter, inspiration is drawn from the literature on the LSM tree.

The LSM tree [40] is a data structure proposed to optimize write throughput in disk-based storage engines. It consists of a memory table of bounded size, used to buffer recent changes, and a hierarchy of external levels containing indexes of exponentially increasing size. When the memory table has reached capacity, it is emptied into the external levels. Random writes are avoided by treating the data within the external levels as immutable; all writes go through the memory table. This introduces write amplification but maximizes sequential writes, which is important for maintaining high throughput in disk-based systems. The LSM tree is associated with a broad and well studied design space [11, 17–19, 21] containing trade-offs between three key performance metrics: read performance, write performance, and auxiliary memory usage. The challenges faced in reconstructing predominately in-memory indexes are quite different from those which the LSM tree is intended to address, having little to do with disk-based systems and sequential IO operations. But, the LSM tree possesses a rich design space for managing the periodic reconstruction of data structures in a manner that is both more practical and more flexible than that of Bentley-Saxe. By borrowing from this design space, this preexisting body of work can be leveraged, and many of Bentley-Saxe's limitations addressed.

3.1 Framework Overview

The goal of this paper is to build a general framework that extends most SSIs with efficient support for updates by splitting the index into small shards to reduce reconstruction costs, and then distributing the sampling process across these shards. This framework is designed to work efficiently with any SSI, so long as it has the following properties,



N_b records
(3) Append to MB
(2) Build shard from MB and combine with L_0
(1) Combine L_0 and L_1
MB
L_0
L_1
A
Insert completed
shard
$N_b \cdot s^{i+1}$ records
1 shard per level
(a) Leveling
(3) Build shard from MB and emplace in L_0
(4) Append to MB
(1) Combine shards in L_0
(2) Emplace new shard in L_1
$N_b \cdot s^i$ records
s shards per level
(b) Tiering

Fig. 2. A graphical overview of the framework and its insert procedure. A mutable buffer (MB) sits atop two levels (L0, L1) containing shards (pairs of SSIs and auxiliary structures [A]) using the leveling (Figure 2a) and tiering (Figure 2b) layout policies. Records are represented as black/colored squares, and grey squares represent unused capacity. An insertion requiring a multi-level reconstruction is illustrated.

(1) The underlying full query Q supported by the SSI from whose results samples are drawn satisfies the following property: for any dataset $D = \cup_{i=1}^{n} D_i$ where $D_i \cap D_j = \emptyset$, $Q(D) = \cup_{i=1}^{n} Q(D_i)$.
(2) *(Optional)* The SSI supports efficient point-lookups.
(3) *(Optional)* The SSI is capable of efficiently reporting the total weight of all records returned by the underlying full query.

The first property applies to the query being sampled from, and is essential for the correctness of sample sets reported by extended sampling indexes.[3] The latter two properties are optional, but reduce deletion and sampling costs respectively. Should the SSI fail to support point-lookups, an auxiliary hash table can be added to the shards. Should it fail to support query result weight reporting, rejection sampling can be used in place of the more efficient scheme discussed in Section 3.3. The analysis of this framework will generally assume that all three conditions are satisfied.

Given an SSI with these properties, a dynamic extension can be produced as shown in Figure 2. The extended index consists of disjoint shards containing an instance of the SSI being extended, and optional auxiliary data structures. The auxiliary structures allow acceleration of certain operations that are required by the framework, but which the SSI being extended does not itself support efficiently. Examples of possible auxiliary structures include hash tables, Bloom filters [13], and range filters [33, 52]. The shards are arranged into levels of increasing record capacity, with either one shard, or up to a fixed maximum number of shards, per level. The decision to place one or many shards per level is called the *layout policy*. The policy names are borrowed from the literature on the LSM tree, with the former called *leveling* and the latter called *tiering*.

To avoid a reconstruction on every insert, an unsorted array of fixed capacity (N_b), called the *mutable buffer*, is used to buffer updates. Because it is unsorted, it is kept small to maintain reasonably efficient sampling and point-lookup performance. All updates are performed by appending new

[3]This condition is stricter than the definition of a decomposable search problem in the Bentley-Saxe method, which allows for *any* constant-time merge operation, not just union. However, this condition is satisfied by many common types of database query, such as predicate-based filtering queries.

Table 1. Frequently Used Notation

Variable	Description
N_b	Capacity of the mutable buffer
s	Scale factor
$C_c(n)$	SSI initial construction cost
$C_r(n)$	SSI reconstruction cost
$L(n)$	SSI point-lookup cost
$P(n)$	SSI sampling pre-processing cost
$S(n)$	SSI per-sample sampling cost
$W(n)$	Shard weight determination cost
$R(n)$	Shard rejection check cost
δ	Maximum delete proportion

records to the tail of this buffer. If a record currently within the index is to be updated to a new value, it must first be deleted, and then a record with the new value inserted. This ensures that old versions of records are properly filtered from query results.

When the buffer is full, it is flushed to make room for new records. The flushing procedure is based on the layout policy in use. When using leveling (Figure 2a) a new SSI is constructed using both the records in L_0 and those in the buffer. This is used to create a new shard, which replaces the one previously in L_0. When using tiering (Figure 2b) a new shard is built using only the records from the buffer, and placed into L_0 without altering the existing shards. Each level has a record capacity of $N_b \cdot s^{i+1}$, controlled by a configurable parameter, s, called the scale factor. Records are organized in one large shard under leveling, or in s shards of $N_b \cdot s^i$ capacity each under tiering. When a level reaches its capacity, it must be emptied to make room for the records flushed into it. This is accomplished by moving its records down to the next level of the index. Under leveling, this requires constructing a new shard containing all records from both the source and target levels, and placing this shard into the target, leaving the source empty. Under tiering, the shards in the source level are combined into a single new shard that is placed into the target level. Should the target be full, it is first emptied by applying the same procedure. New empty levels are dynamically added as necessary to accommodate these reconstructions. Note that shard reconstructions are not necessarily performed using merging, though merging can be used as an optimization of the reconstruction procedure where such an algorithm exists. In general, reconstruction requires only pooling the records of the shards being combined and then applying the SSI's standard construction algorithm to this set of records.

Table 1 lists frequently used notation for the various parameters of the framework, which will be used in the coming analysis of the costs and trade-offs associated with operations within the framework's design space. The remainder of this section will discuss the performance characteristics of insertion into this structure (Section 3.2), how it can be used to correctly answer sampling queries (Section 3.2), and efficient approaches for supporting deletes (Section 3.4). Finally, it will close with a detailed discussion of the trade-offs within the framework's design space (Section 3.5).

3.2 Insertion

The framework supports inserting new records by first appending them to the end of the mutable buffer. When it is full, the buffer is flushed into a sequence of levels containing shards of increasing capacity, using a procedure determined by the layout policy as discussed in Section 3. This method allows for the cost of repeated shard reconstruction to be effectively amortized.

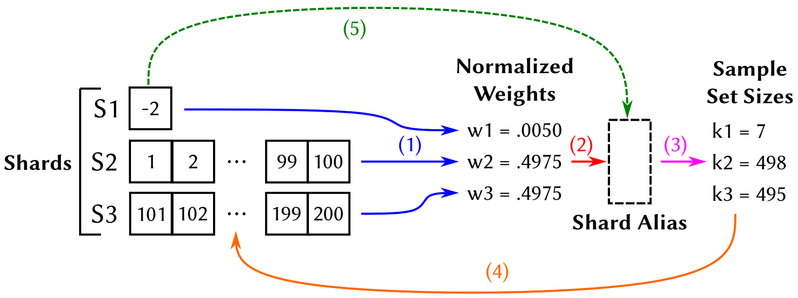


Fig. 3. Overview of the sampling query process for Example 1 with $k = 1000$. First, (1) the normalized weights of the shards is determined, then (2) these weights are used to construct an alias structure. Next, (3) the alias structure is queried k times to determine per shard sample sizes, and then (4) sampling is performed. Finally, (5) any rejected samples are retried starting from the alias structure, and the process is repeated until the desired number of samples has been retrieved.

Let the cost of constructing the SSI from an arbitrary set of n records be $C_c(n)$ and the cost of reconstructing the SSI given two or more shards containing n records in total be $C_r(n)$. The cost of an insert is composed of three parts: appending to the mutable buffer, constructing a new shard from the buffered records during a flush, and the total cost of reconstructing shards containing the record over the lifetime of the index. The cost of appending to the mutable buffer is constant, and the cost of constructing a shard from the buffer can be amortized across the records participating in the buffer flush, giving $C_c(N_b)/N_b$. These costs are paid exactly once for each record. To derive an expression for the cost of repeated reconstruction, first note that each record will participate in at most s reconstructions on a given level, resulting in a worst-case amortized cost of $O\left(s \cdot C_r(n)/n\right)$ paid per level. The index itself will contain at most $\log_s n$ levels. Thus, over the lifetime of the index a given record will pay $O\left(s \cdot C_r(n)/n \log_s n\right)$ cost in repeated reconstruction.

Combining these results, the total amortized insertion cost is $O\left(C_c(N_b)/N_b + s \cdot C_r(n)/n \log_s n\right)$. This can be simplified by noting that s is constant, and that $N_b \ll n$ and also a constant. By neglecting these terms, the amortized insertion cost of the framework is,

$$O\left(\frac{C_r(n)}{n} \log_s n\right) \tag{1}$$

3.3 Sampling

For many SSIs, sampling queries are completed in two stages. Some preliminary processing is done to identify the range of records from which to sample, and then samples are drawn from that range. For example, IRS over a sorted list of records can be performed by first identifying the upper and lower bounds of the query range in the list, and then sampling records by randomly generating indexes within those bounds. The general cost of a sampling query can be modeled as $P(n) + kS(n)$, where $P(n)$ is the cost of preprocessing, k is the number of samples drawn, and $S(n)$ is the cost of sampling a single record.

When sampling from multiple shards, the situation grows more complex. For each sample, the shard to select the record from must first be decided. Consider an arbitrary sampling query $X(D, k)$ asking for a sample set of size k against dataset D. The framework splits D across m disjoint shards, such that $D = \bigcup_{i=1}^{m} D_i$ and $D_i \cap D_j = \emptyset, \forall i, j < m$. The framework must ensure that $X(D, k)$ and $\bigcup_{i=0}^{m} X(D_i, k_i)$ follow the same distribution, by selecting appropriate values for the k_is. If care is not taken to balance the number of samples drawn from a shard with the total weight of the shard under X, then bias can be introduced into the sample set's distribution. The selection of k_is can be viewed as an instance of WSS, and solved using the alias method.

When sampling using the framework, first the weight of each shard under the sampling query is determined and a *shard alias structure* built over these weights. Then, for each sample, the shard alias is used to determine the shard from which to draw the sample. Let $W(n)$ be the cost of determining this total weight for a single shard under the query. The initial setup cost, prior to drawing any samples, will be $O\left([W(n) + P(n)] \log_s n\right)$, as the preliminary work for sampling from each shard must be performed, as well as weights determined and alias structure constructed. In many cases, however, the preliminary work will also determine the total weight, and so the relevant operation need only be applied once to accomplish both tasks.

To ensure that all records appear in the sample set with the appropriate probability, the mutable buffer itself must also be a valid target for sampling. There are two generally applicable techniques that can be applied for this, both of which can be supported by the framework. The query being sampled from can be directly executed against the buffer and the result set used to build a temporary SSI, which can be sampled from. Alternatively, rejection sampling can be used to sample directly from the buffer, without executing the query. In this case, the total weight of the buffer is used for its entry in the shard alias structure. This can result in the buffer being over-represented in the shard selection process, and so any rejections during buffer sampling must be retried starting from shard selection. These same considerations apply to rejection sampling used against shards, as well.

Example 1. *Consider executing a WSS query, with* $k = 1000$, *across three shards containing integer keys with unit weight.* S_1 *contains only the key* -2, S_2 *contains all integers on* $[1, 100]$, *and* S_3 *contains all integers on* $[101, 200]$. *These structures are shown in Figure 3. Sampling is performed by first determining the normalized weights for each shard:* $w_1 = 0.005$, $w_2 = 0.4975$, $w_3 = 0.4975$, *which are then used to construct a shard alias structure. The shard alias structure is then queried* k *times, resulting in a distribution of* k_i*s that is commensurate with the relative weights of each shard. Finally, each shard is queried in turn to draw the appropriate number of samples.*

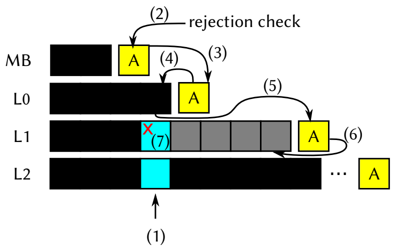
Assuming that rejection sampling is used on the mutable buffer, the worst-case time complexity for drawing k samples from an index containing n elements with a sampling cost of $S(n)$ is,

$$O\left([W(n) + P(n)] \log_s n + kS(n)\right) \tag{2}$$

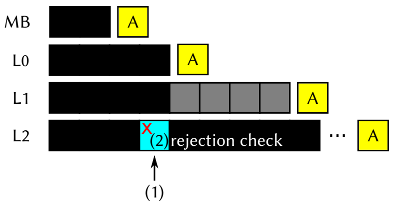
3.4 Deletion

Because the shards are static, records cannot be arbitrarily removed from them. This requires that deletes be supported in some other way, with the ultimate goal being the prevention of deleted records' appearance in sampling query result sets. This can be realized in two ways: locating the record and marking it, or inserting a new record which indicates that an existing record should be treated as deleted. The framework supports both of these techniques, the selection of which is called the *delete policy.* The former policy is called *tagging* and the latter *tombstone.*

Tagging a record is straightforward. Point-lookups are performed against each shard in the index, as well as the buffer, for the record to be deleted. When it is found, a bit in a header attached to the record is set. When sampling, any records selected with this bit set are automatically rejected. Tombstones represent a lazy strategy for deleting records. When a record is deleted using tombstones, a new record with identical key and value, but with a "tombstone" bit set, is inserted into the index. A record's presence can be checked by performing a point-lookup. If a tombstone with the same key and value exists above the record in the index, then it should be rejected when sampled.



(a) Tombstone Rejection Check



(b) Tagging Rejection Check

Fig. 4. The rejection check procedure when (1) a deleted record is sampled. When using the tombstone delete policy (Figure 4a), the rejection check starts by (2) querying the bloom filter of the mutable buffer. The filter indicates the record is not present, so (3) the filter on L_0 is queried next. This filter returns a false positive, so (4) a point-lookup is executed against L_0. The lookup fails to find a tombstone, so the search continues and (5) the filter on L_1 is checked, which reports that the tombstone is present. This time, it is not a false positive, and so (6) a lookup against L_1 (7) locates the tombstone. The record is thus rejected. When using the tagging policy (Figure 4b), (1) the record is sampled and (2) checked directly for the delete tag. It is set, so the record is immediately rejected.

Two important aspects of performance are pertinent when discussing deletes: the cost of the delete operation, and the cost of verifying the presence of a sampled record. The choice of delete policy represents a trade-off between these two costs. Beyond this simple trade-off, the delete policy also has other implications that can affect its applicability to certain types of SSI. Most notably, tombstones do not require any in-place updating of records, whereas tagging does. This means that using tombstones is the only way to ensure total immutability of the data within shards, which avoids random writes and eases concurrency control. The tombstone delete policy, then, is particularly appealing in external and concurrent contexts.

Deletion Cost. The cost of a delete under the tombstone policy is the same as an ordinary insert. Tagging, by contrast, requires a point-lookup of the record to be deleted, and so is more expensive. Assuming a point-lookup operation with cost $L(n)$, a tagged delete must search each level in the index, as well as the buffer, requiring $O\left(N_b + L(n)\log_s n\right)$ time.

Rejection Check Costs. In addition to the cost of the delete itself, the delete policy affects the cost of determining if a given record has been deleted. This is called the *rejection check cost*, $R(n)$. When using tagging, the information necessary to make the rejection decision is local to the sampled record, and so $R(n) \in O(1)$. However, when using tombstones it is not; a point-lookup must be performed to search for a given record's corresponding tombstone. This look-up must examine the buffer, and each shard within the index. This results in a rejection check cost of $R(n) \in O\left(N_b + L(n)\log_s n\right)$. The rejection check process for the two delete policies is summarized in Figure 4.

Two factors contribute to the tombstone rejection check cost: the size of the buffer, and the cost of performing a point-lookup against the shards. The latter cost can be controlled using the framework's ability to associate auxiliary structures with shards. For SSIs which do not support efficient point-lookups, a hash table can be added to map key-value pairs to their location within the SSI. This allows for constant-time rejection checks, even in situations where the index would not otherwise support them. However, the storage cost of this intervention is high, and in situations where the SSI does support efficient point-lookups, it is not necessary. Further performance improvements can be achieved by noting that the probability of a given record having an associated tombstone in any particular shard is relatively small. This means that many point-lookups will be executed against shards that do not contain the tombstone being searched for. In this case, these unnecessary lookups can be partially avoided using Bloom filters [13] for tombstones. By inserting tombstones into these filters during reconstruction, point-lookups against some shards which do not contain the tombstone being searched for can be bypassed. Filters can be attached to the buffer as well, which may be even more significant due to the linear cost of scanning it. As the goal is a reduction of rejection check costs, these filters need only be populated with tombstones. In a later section, techniques for bounding the number of tombstones on a given level are discussed, which will allow for the memory usage of these filters to be tightly controlled while still ensuring precise bounds on filter error.

Sampling with Deletes. The addition of deletes to the framework alters the analysis of sampling costs. A record that has been deleted cannot be present in the sample set, and therefore the presence of each sampled record must be verified. If a record has been deleted, it must be rejected. When retrying samples rejected due to delete, the process must restart from shard selection, as deleted records may be counted in the weight totals used to construct that structure. This increases the cost of sampling to,

$$O\left([W(n) + P(n)] \log_s n + \frac{kS(n)}{1 - \mathbf{Pr}[\text{rejection}]} \cdot R(n)\right) \tag{3}$$

where $R(n)$ is the cost of checking if a sampled record has been deleted, and $^{k}/_{1-\mathbf{Pr}[\text{rejection}]}$ is the expected number of sampling attempts required to obtain k samples, given a fixed rejection probability. The rejection probability itself is a function of the workload, and is unbounded.

Bounding the Rejection Probability. Rejections during sampling constitute wasted memory accesses and random number generations, and so steps should be taken to minimize their frequency. The probability of a rejection is directly related to the number of deleted records, which is itself a function of workload and dataset. This means that, without building counter-measures into the framework, tight bounds on sampling performance cannot be provided in the presence of deleted records. It is therefore critical that the framework support some method for bounding the number of deleted records within the index.

While the static nature of shards prevents the direct removal of records at the moment they are deleted, it doesn't prevent the removal of records during reconstruction. When using tagging, all tagged records encountered during reconstruction can be removed. When using tombstones, however, the removal process is non-trivial. In principle, a rejection check could be performed for each record encountered during reconstruction, but this would increase reconstruction costs and introduce a new problem of tracking tombstones associated with records that have been removed. Instead, a lazier approach can be used: delaying removal until a tombstone and its associated record participate in the same shard reconstruction. This delay allows both the record and its tombstone to be removed at the same time, an approach called *tombstone cancellation*. In general, this can be implemented using an extra linear scan of the input shards before reconstruction to identify tombstones and associated records for cancellation, but potential optimizations exist for many SSIs, allowing it to be performed during the reconstruction itself at no extra cost.

The removal of deleted records passively during reconstruction is not enough to bound the number of deleted records within the index. It is not difficult to envision pathological scenarios where deletes result in unbounded rejection rates, even with this mitigation in place. However, the dropping of deleted records does provide a useful property: any specific deleted record will eventually be removed from the index after a finite number of reconstructions. Using this fact, a bound on the number of deleted records can be enforced. A new parameter, δ, is defined, representing the maximum proportion of deleted records within the index. Each level, and the buffer, tracks the number of deleted records it contains by counting its tagged records or tombstones. Following each buffer flush, the proportion of deleted records is checked against δ. If any level is found to exceed it, then a proactive reconstruction is triggered, pushing its shards down into the next level. The process is repeated until all levels respect the bound, allowing the number of deleted records to be precisely controlled, which, by extension, bounds the rejection rate. This process is called *compaction*.

Assuming every record is equally likely to be sampled, this new bound can be applied to the analysis of sampling costs. The probability of a record being rejected is $\mathbf{Pr}[\text{rejection}] = \delta$. Applying this result to Equation 3 yields,

$$O\left([W(n) + P(n)]\log_s n + \frac{kS(n)}{1-\delta} \cdot R(n)\right) \tag{4}$$

Asymptotically, this proactive compaction does not alter the analysis of insertion costs. Each record is still written at most s times on each level, there are at most $\log_s n$ levels, and the buffer insertion and SSI construction costs are all unchanged, and so on. This results in the amortized insertion cost remaining the same.

This compaction strategy is based upon tombstone and record counts, and the bounds assume that every record is equally likely to be sampled. For certain sampling problems (such as WSS), there are other conditions that must be considered to provide a bound on the rejection rate. To account for these situations in a general fashion, the framework supports problem-specific compaction triggers that can be tailored to the SSI being used. These allow compactions to be triggered based on other properties, such as rejection rate of a level, weight of deleted records, and the like.

3.5 Trade-offs on Framework Design Space

The framework has several tunable parameters, allowing it to be tailored for specific applications. This design space contains trade-offs among three major performance characteristics: update cost, sampling cost, and auxiliary memory usage. The two most significant decisions when implementing this framework are the selection of the layout and delete policies. The asymptotic analysis of the previous sections obscures some of the differences between these policies, but they do have significant practical performance implications.

Layout Policy. The choice of layout policy represents a clear trade-off between update and sampling performance. Leveling results in fewer shards of larger size, whereas tiering results in a larger number of smaller shards. As a result, leveling reduces the costs associated with point-lookups and sampling query preprocessing by a constant factor, compared to tiering. However, it results in more write amplification: a given record may be involved in up to s reconstructions on a single level, as opposed to the single reconstruction per level under tiering.

Delete Policy. There is a trade-off between delete performance and sampling performance that exists in the choice of delete policy. Tagging requires a point-lookup when performing a delete, which is more expensive than the insert required by tombstones. However, it also allows constant-time rejection checks, unlike tombstones which require a point-lookup of each sampled record. In situations where deletes are common and write-throughput is critical, tombstones may be more useful. Tombstones are also ideal in situations where immutability is required, or random writes must be avoided. Generally speaking, however, tagging is superior when using SSIs that support it, because sampling rejection checks will usually be more common than deletes.

Mutable Buffer Capacity and Scale Factor. The mutable buffer capacity and scale factor both influence the number of levels within the index, and by extension the number of distinct shards. Sampling and point-lookups have better performance with fewer shards. Smaller shards are also faster to reconstruct, although the same adjustments that reduce shard size also result in a larger number of reconstructions, so the trade-off here is less clear.

The scale factor has an interesting interaction with the layout policy: when using leveling, the scale factor directly controls the amount of write amplification per level. Larger scale factors mean more time is spent reconstructing shards on a level, reducing update performance. Tiering does not have this problem and should see its update performance benefit directly from a larger scale factor, as this reduces the number of reconstructions.

The buffer capacity also influences the number of levels, but is more significant in its effects on point-lookup performance: a lookup must perform a linear scan of the buffer. Likewise, the unstructured nature of the buffer also will contribute negatively towards sampling performance, irrespective of which buffer sampling technique is used. As a result, although a large buffer will reduce the number of shards, it will also hurt sampling and delete (under tagging) performance. It is important to minimize the cost of these buffer scans, and so it is preferable to keep the buffer small, ideally small enough to fit within the CPU's L2 cache. The number of shards within the index is, then, better controlled by changing the scale factor, rather than the buffer capacity. Using a smaller buffer will result in more compactions and shard reconstructions; however, the empirical evaluation in Section 6.1 demonstrates that this is not a serious performance problem when a scale factor is chosen appropriately. When the shards are in memory, frequent small reconstructions do not have a significant performance penalty compared to less frequent, larger ones.

Auxiliary Structures. The framework's support for arbitrary auxiliary data structures allows for memory to be traded in exchange for insertion or sampling performance. The use of Bloom filters for accelerating tombstone rejection checks has already been discussed, but many other options exist. Bloom filters could also be used to accelerate point-lookups for delete tagging, though such filters would require much more memory than tombstone-only ones to be effective. An auxiliary hash table could be used for accelerating point-lookups, or range filters like SuRF [52] or Rosetta [33] added to accelerate pre-processing for range queries like in IRS or WIRS.

4 FRAMEWORK INSTANTIATIONS

In this section, the framework is applied to three sampling problems and their associated SSIs. All three sampling problems draw random samples from records satisfying a simple predicate, and so result sets for all three can be constructed by directly merging the result sets of the queries executed against individual shards, the primary requirement for the application of the framework. The SSIs used for each problem are discussed, including their support of the remaining two optional requirements for framework application.

4.1 Dynamically Extended WSS Structure

As a first example of applying this framework for dynamic extension, the alias structure for answering WSS queries is considered. This is a static structure that can be constructed in $O(n)$ time and supports WSS queries in $O(1)$ time. The alias structure will be used as the SSI, with the shards containing an alias structure paired with a sorted array of records. The use of sorted arrays for storing the records allows for more efficient point-lookups, without requiring any additional space. The total weight associated with a query for a given alias structure is the total weight of all of its records, and can be tracked at the shard level and retrieved in constant time.

Using the formulae from Section 3, the worst-case costs of insertion, sampling, and deletion are easily derived. The initial construction cost from the buffer is $C_c(N_b) \in O(N_b \log N_b)$, requiring the sorting of the buffer followed by alias construction. After this point, the shards can be reconstructed in linear time while maintaining sorted order. Thus, the reconstruction cost is $C_r(n) \in O(n)$. As each shard contains a sorted array, the point-lookup cost is $L(n) \in O(\log n)$. The total weight can be tracked with the shard, requiring $W(n) \in O(1)$ time to access, and there is no necessary preprocessing, so $P(n) \in O(1)$. Samples can be drawn in $S(n) \in O(1)$ time. Plugging these results into the formulae for insertion, sampling, and deletion costs gives,

$$\begin{aligned}
\text{Insertion:} \quad & O\left(\log_s n\right) \\
\text{Sampling:} \quad & O\left(\log_s n + \frac{k}{1-\delta} \cdot R(n)\right) \\
\text{Tagged Delete:} \quad & O\left(\log_s n \log n\right)
\end{aligned}$$

where $R(n) \in O(1)$ for tagging and $R(n) \in O(\log_s n \log n)$ for tombstones.

Bounding Rejection Rate. In the weighted sampling case, the framework's generic record-based compaction trigger mechanism is insufficient to bound the rejection rate. This is because the probability of a given record being sampling is dependent upon its weight, as well as the number of records in the index. If a highly weighted record is deleted, it will be preferentially sampled, resulting in a larger number of rejections than would be expected based on record counts alone. This problem can be rectified using the framework's user-specified compaction trigger mechanism. In addition to tracking record counts, each level also tracks its rejection rate,

$$\rho_i = \frac{\text{rejections}}{\text{sampling attempts}}$$

A configurable rejection rate cap, ρ, is then defined. If $\rho_i > \rho$ on a level, a compaction is triggered. In the case the tombstone delete policy, it is not the level containing the sampled record, but rather the level containing its tombstone, that is considered the source of the rejection. This is necessary to ensure that the tombstone is moved closer to canceling its associated record by the compaction.

4.2 Dynamically Extended IRS Structure

Another sampling problem to which the framework can be applied is independent range sampling (IRS). The SSI in this example is the in-memory ISAM tree. The ISAM tree supports efficient point-lookups directly, and the total weight of an IRS query can be easily obtained by counting the number of records within the query range, which is determined as part of the preprocessing of the query.

The static nature of shards in the framework allows for an ISAM tree to be constructed with adjacent nodes positioned contiguously in memory. By selecting a leaf node size that is a multiple of the record size, and avoiding placing any headers within leaf nodes, the set of leaf nodes can be treated as a sorted array of records with direct indexing, and the internal nodes allow for faster searching of this array. Because of this layout, per-sample tree-traversals are avoided. The start and end of the range from which to sample can be determined using a pair of traversals, and then records can be sampled from this range using random number generation and array indexing.

Assuming a sorted set of input records, the ISAM tree can be bulk-loaded in linear time. The insertion analysis proceeds like the WSS example previously discussed. The initial construction cost is $C_c(N_b) \in O(N_b \log N_b)$ and reconstruction cost is $C_r(n) \in O(n)$. The ISAM tree supports point-lookups in $L(n) \in O(\log_f n)$ time, where f is the fanout of the tree.

The process for performing range sampling against the ISAM tree involves two stages. First, the tree is traversed twice: once to establish the index of the first record greater than or equal to the lower bound of the query, and again to find the index of the last record less than or equal to the upper bound of the query. This process has the effect of providing the number of records within the query range, and can be used to determine the weight of the shard in the shard alias structure. Its cost is $P(n) \in O(\log_f n)$. Once the bounds are established, samples can be drawn by randomly generating uniform integers between the upper and lower bound, in $S(n) \in O(1)$ time each.

This results in the extended version of the ISAM tree having the following insert, sampling, and delete costs,

$$\begin{aligned}\text{Insertion:}\quad & O\left(\log_s n\right)\\ \text{Sampling:}\quad & O\left(\log_s n \log_f n + \frac{k}{1-\delta}\cdot R(n)\right)\\ \text{Tagged Delete:}\quad & O\left(\log_s n \log_f n\right)\end{aligned}$$

where $R(n) \in O(1)$ for tagging and $R(n) \in O(\log_s n \log_f n)$ for tombstones.

4.3 Dynamically Extended WIRS Structure

As a final example of applying this framework, the WIRS problem will be considered. Specifically, the alias-augmented B+tree approach, described by Tao [46], generalizing work by Afshani and Wei [6], and Hu et al. [27], will be extended. This structure allows for efficient point-lookups, as it is based on the B+tree, and the total weight of a given WIRS query can be calculated given the query range using aggregate weight tags within the tree.

The alias-augmented B+tree is a static structure of linear space, capable of being built initially in $C_c(N_b) \in O(N_b \log N_b)$ time, being bulk-loaded from sorted lists of records in $C_r(n) \in O(n)$ time, and answering WIRS queries in $O(\log_f n + k)$ time, where the query cost consists of preliminary work to identify the sampling range and calculate the total weight, with $P(n) \in O(\log_f n)$ cost, and constant-time drawing of samples from that range with $S(n) \in O(1)$. This results in the following costs,

$$\begin{aligned}\text{Insertion:}\quad & O\left(\log_s n\right)\\ \text{Sampling:}\quad & O\left(\log_s n \log_f n + \frac{k}{1-\delta}\cdot R(n)\right)\\ \text{Tagged Delete:}\quad & O\left(\log_s n \log_f n\right)\end{aligned}$$

where $R(n) \in O(1)$ for tagging and $R(n) \in O(\log_s n \log_f n)$ for tombstones. Because this is a weighted sampling structure, the custom compaction trigger discussed in in Section 4.1 is applied to maintain bounded rejection rates during sampling.

5 EXTENSIONS

In this section, various extensions of the framework are considered. Specifically, the applicability of the framework to external or distributed data structures is discussed, as well as the use of the framework to add automatic support for concurrent updates and sampling to extended SSIs.

Larger-than-Memory Data. This framework can be applied to external static sampling structures with minimal modification. As a proof-of-concept, the IRS structure was extended with support for shards containing external ISAM trees. This structure supports storing a configurable number of shards in memory, and the rest on disk, making it well suited for operating in memory-constrained environments. The on-disk shards contain standard ISAM trees, with 8KiB page-aligned nodes. The external version of the index only supports tombstone-based deletes, as tagging would require random writes. In principle a hybrid approach to deletes is possible, where a delete first searches the in-memory data for the record to be deleted, tagging it if found. If the record is not found, then a tombstone could be inserted. As the data size grows, though, and the preponderance of data is found on disk, this approach would largely revert to the standard tombstone approach in practice. External settings make the framework even more attractive, in terms of performance characteristics, due to the different cost model. In external data structures, performance is typically measured in terms of the number of IO operations, meaning that much of the overhead introduced by the framework for tasks like querying the mutable buffer, building auxiliary structures, extra random number generations due to the shard alias structure, and the like, become far less significant.

Because the framework maintains immutability of shards, it is also well suited for use on top of distributed file-systems or with other distributed data abstractions like RDDs in Apache Spark [51]. Each shard can be encapsulated within an immutable file in HDFS or an RDD in Spark. A centralized control node or driver program can manage the mutable buffer, flushing it into a new file or RDD when it is full, merging with existing files or RDDs using the same reconstruction scheme already discussed for the framework. This setup allows for datasets exceeding the capacity of a single node to be supported. As an example, XDB [32] features an RDD-based distributed sampling structure that could be supported by this framework.

Concurrency. The immutability of the majority of the structures within the index makes for a straightforward concurrency implementation. Concurrency control on the buffer is made trivial by the fact it is a simple, unsorted array. The rest of the structure is never updated (aside from possible delete tagging), and so concurrency becomes a simple matter of delaying the freeing of memory used by internal structures until all the threads accessing them have exited, rather than immediately on merge completion. A very basic concurrency implementation can be achieved by using the tombstone delete policy, and a reference counting scheme to control the deletion of the shards following reconstructions. Multiple insert buffers can be used to improve insertion throughput, as this will allow inserts to proceed in parallel with merges, ultimately allowing concurrency to scale up to the point of being bottlenecked by memory bandwidth and available storage. This proof-of-concept implementation is based on a simplified version of an approach proposed by Golan-Gueta et al. for concurrent log-structured data stores [24].

6 EVALUATION

Experimental Setup. All experiments were run under Ubuntu 20.04 LTS on a dual-socket Intel Xeon Gold 6242R server with 384 GiB of physical memory and 40 physical cores. External tests were run using a 4 TB WD Red SA500 SATA SSD, rated for 95000 and 82000 IOPS for random reads and writes respectively.

Datasets. Testing utilized a variety of synthetic and real-world datasets. For all datasets used, the key was represented as a 64-bit integer, the weight as a 64-bit integer, and the value as a 32-bit integer. Each record also contained a 32-bit header. The weight was omitted from IRS testing. Keys and weights were pulled from the dataset directly, and values were generated separately and were unique for each record. The following datasets were used,

- **Synthetic Uniform.** A non-weighted, synthetically generated list of keys drawn from a uniform distribution.
- **Synthetic Zipfian.** A non-weighted, synthetically generated list of keys drawn from a Zipfian distribution with a skew of 0.8.
- **Twitter [4, 31].** 41 million Twitter user ids, weighted by follower counts.
- **Delicious [1].** 33.7 million URLs, represented using unique integers, weighted by the number of associated tags.

- **OSM [2].** 2.6 billion geospatial coordinates for points of interest, collected by OpenStreetMap. The latitude, converted to a 64-bit integer, was used as the key and the number of its associated semantic tags as the weight.

The synthetic datasets were not used for weighted experiments, as they do not have weights. For unweighted experiments, the Twitter and Delicious datasets were not used, as they have uninteresting key distributions.

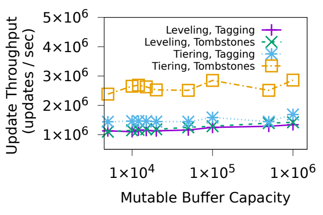
Compared Methods. In this section, indexes extended using the framework are compared against existing dynamic baselines. Specifically, DE-WSS (Section 4.1), DE-IRS (Section 4.2), and DE-WIRS (Section 4.2) are examined. In-memory extensions are compared against the B+tree with aggregate weight tags on internal nodes (AGG B+tree) [39] and concurrent and external extensions are compared against the AB-tree [53]. Sampling performance is also compared against comparable static sampling indexes: the alias structure [49] for WSS, the in-memory ISAM tree for IRS, and the alias-augmented B+tree [6] for WIRS. Note that all structures under test, with the exception of the external DE-IRS and external AB-tree, were contained entirely within system memory. All benchmarking code and data structures were implemented using C++17 and compiled using gcc 11.3.0 at the `-O3` optimization level. The extension framework itself, excluding the shard implementations and utility headers, consisted of a header-only library of about 1200 SLOC.

6.1 Framework Design Space Exploration

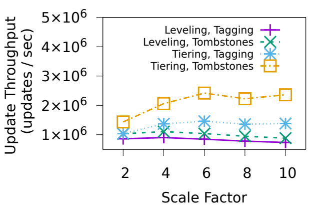
The proposed framework brings with it a large design space, described in Section 3.5. First, this design space will be examined using a standardized benchmark to measure the average insertion throughput and sampling latency of DE-WSS at several points within this space. Tests were run using a random selection of 500 million records from the OSM dataset, with the index warmed up by the insertion of 10% of the total records prior to beginning any measurement. Over the course of the insertion period, 5% of the records were deleted, except for the tests in Figures 5c, 5f, and 5h, in which 25% of the records were deleted. Reported update throughputs were calculated using both inserts and deletes, following the warmup period. The standard values used for parameters not being varied in a given test were $s = 6$, $N_b = 12000$, $k = 1000$, and $\delta = 0.05$, with buffer rejection sampling.

The results of this testing are displayed in Figure 5. The two largest contributors to differences in performance were the selection of layout policy and of delete policy. Figures 5a and 5b show that the choice of layout policy plays a larger role than delete policy in insertion performance, with tiering outperforming leveling in both configurations. The situation is reversed in sampling performance, seen in Figure 5d and 5e, where the performance difference between layout policies is far less than between delete policies.

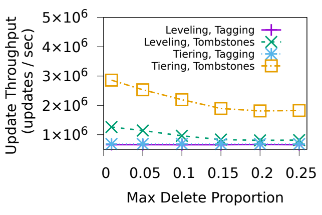
The values used for the scale factor and buffer size have less influence than layout and delete policy. Sampling performance is largely independent of them over the ranges of values tested, as shown in Figures 5d and 5e. This isn't surprising, as these parameters adjust the number of shards, which only contributes to shard alias construction time during sampling and is is amortized over all samples taken in a query. The buffer also contributes rejections, but the cost of a rejection is small and the buffer constitutes only a small portion of the total weight, so these are negligible. However, under tombstones there is an upward trend in latency with buffer size, as delete checks occasionally require a full buffer scan. The effect of buffer size on insertion is shown in Figure 5a. There is only a small improvement in insertion performance as the mutable buffer grows. This is because a larger buffer results in fewer reconstructions, but these reconstructions individually take longer, and so the net positive effect is less than might be expected. Finally, Figure 5b shows the effect of scale factor on insertion performance. As expected, tiering performs better with higher



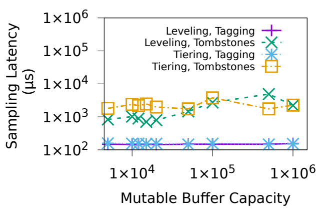
(a) Insertion Throughput vs. Mutable Buffer Capacity



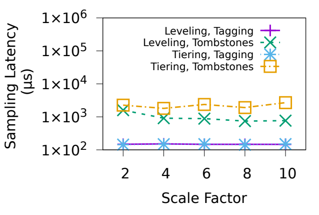
(b) Insertion Throughput vs. Scale Factor



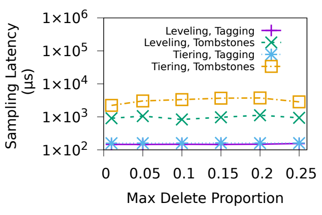
(c) Insertion Throughput vs. Max Delete Proportion



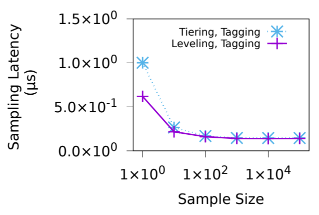
(d) Per 1000 Sampling Latency vs. Mutable Buffer Capacity



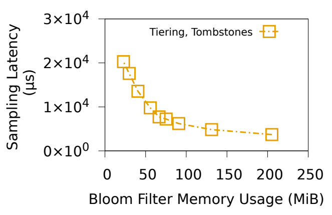
(e) Per 1000 Sampling Latency vs. Scale Factor



(f) Per 1000 Sampling Latency vs. Max Delete Proportion



(g) Sampling Latency vs. Sample Size



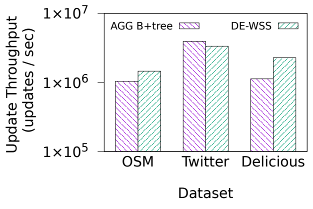
(h) Per 1000 Sampling Latency vs. Bloom Filter Memory

Fig. 5. DE-WSS Design Space Exploration

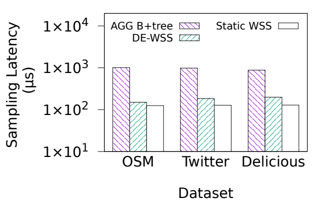
scale factors, whereas the insertion performance of leveling trails off as the scale factor is increased, due to write amplification.

Figures 5c and 5f show the cost of maintaining δ with a base delete rate of 25%. The low cost of an in-memory sampling rejection results in only a slight upward trend in the sampling latency as the number of deleted records increases. While compaction is necessary to avoid pathological cases, there does not seem to be a significant benefit to aggressive compaction thresholds. Figure 5c shows the effect of compactions on insert performance. There is little effect on performance under tagging, but there is a clear negative performance trend associated with aggressive compaction when using tombstones. Under tagging, a single compaction is guaranteed to remove all deleted records on a level, whereas with tombstones a compaction can cascade for multiple levels before the delete bound is satisfied, resulting in a larger cost per incident.

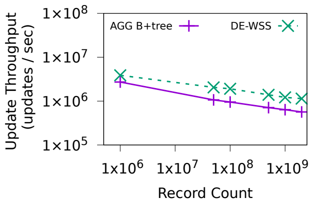
Figure 5h demonstrates the trade-off between memory usage for Bloom filters and sampling performance under tombstones. This test was run using 25% incoming deletes with no compaction, to maximize the number of tombstones within the index as a worst-case scenario. As expected, allocating more memory to Bloom filters, decreasing their false positive rates, accelerates sampling. Finally, Figure 5g shows the relationship between average per sample latency and the sample set



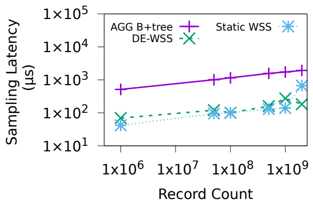
(a) WSS Insertion Throughput vs. Baselines



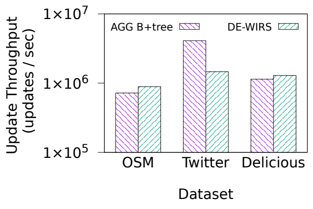
(b) WSS Sampling Latency vs. Baselines



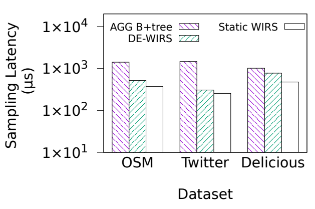
(c) WSS Insertion Scalability vs. Baselines



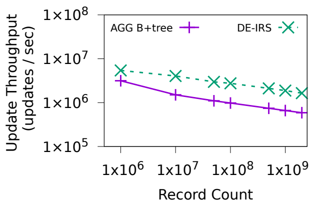
(d) WSS Sampling Scalability vs. Baselines



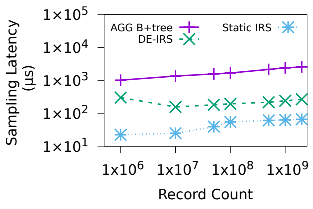
(e) WIRS Insertion Throughput vs. Baselines



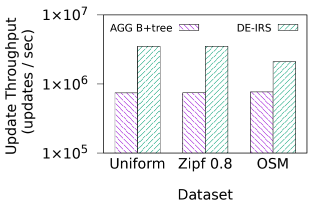
(f) WIRS Sampling Latency vs. Baselines



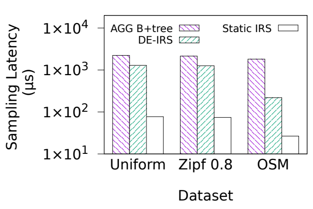
(g) IRS Insertion Scalability vs. Baselines



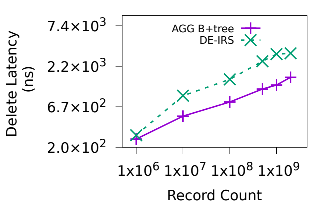
(h) IRS Sampling Scalability vs. Baselines



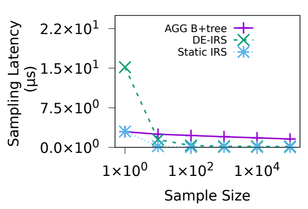
(i) IRS Insertion Throughput vs. Baselines



(j) IRS Sampling Latency vs. Baselines



(k) IRS Delete Scalability vs. Baselines



(l) IRS Sampling Latency vs. Sample Size

Fig. 6. Index Comparisons to Baselines

size. It shows the effect of amortizing the initial shard alias setup work across an increasing number of samples, with $k = 100$ as the point at which latency levels off.

Based upon these results, a set of parameters was established for the extended indexes, which is used in the next section for baseline comparisons. This standard configuration uses tagging as the delete policy and tiering as the layout policy, with $k = 1000$, $N_b = 12000$, $\delta = 0.05$, and $s = 6$.

6.2 Comparison to Baselines

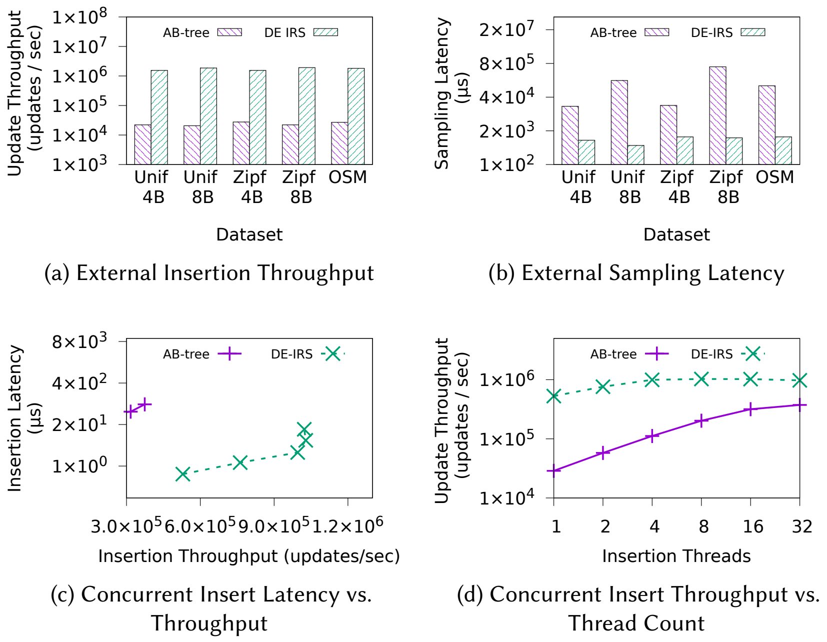
Next, the performance of indexes extended using the framework is compared against tree sampling on the aggregate B+tree, as well as problem-specific SSIs for WSS, WIRS, and IRS queries. Unless otherwise specified, IRS and WIRS queries were executed with a selectivity of 0.1% and 500 million randomly selected records from the OSM dataset were used. The uniform and zipfian synthetic datasets were 1 billion records in size. All benchmarks warmed up the data structure by inserting 10% of the records, and then measured the throughput inserting the remaining records, while deleting 5% of them over the course of the benchmark. Once all records were inserted, the sampling performance was measured. The reported update throughputs were calculated using both inserts and deletes, following the warmup period.

Starting with WSS, Figure 6a shows that the DE-WSS structure is competitive with the AGG B+tree in terms of insertion performance, achieving about 85% of the AGG B+tree's insertion throughput on the Twitter dataset, and beating it by similar margins on the other datasets. In terms of sampling performance in Figure 6b, it beats the B+tree handily, and compares favorably to the static alias structure. Figures 6c and 6d show the performance scaling of the three structures as the dataset size increases. All of the structures exhibit the same type of performance degradation with respect to dataset size.

Figures 6e and 6f show the performance of the DE-WIRS index, relative to the AGG B+tree and the alias-augmented B+tree. This example shows the same pattern of behavior as was seen with DE-WSS, though the margin between the DE-WIRS and its corresponding SSI is much narrower. Additionally, the constant factors associated with the construction cost of the alias-augmented B+tree are much larger than the alias structure. The loss of insertion performance due to this is seen clearly in Figure 6e, where the margin of advantage between DE-WIRS and the AGG B+tree in insertion throughput shrinks compared to the DE-WSS index, and the AGG B+tree's advantage on the Twitter dataset is expanded. Finally, Figures 6i and 6j show a comparison of the in-memory DE-IRS index against the in-memory ISAM tree and the AGG B+tree for answering IRS queries. The cost of bulk-loading the ISAM tree is less than the cost of building the alias structure, or the alias-augmented B+tree, and so here DE-IRS defeats the AGG B+tree by wider margins in insertion throughput, though the margin narrows significantly in terms of sampling performance advantage.

DE-IRS was further tested to evaluate scalability. Figure 6g shows average insertion throughput, Figure 6k shows average delete latency (under tagging), and Figure 6h shows average sampling latencies for DE-IRS and AGG B+tree over a range of data sizes. In all cases, DE-IRS and B+tree show similar patterns of performance degradation as the datasize grows. Note that the delete latencies of DE-IRS are worse than AGG B+tree, because of the B+tree's cheaper point-lookups.

Figure 6h also includes one other point of interest: the sampling performance of DE-IRS *improves* when the data size grows from one million to ten million records. While at first glance the performance increase may appear paradoxical, it actually demonstrates an important result concerning the effect of the unsorted mutable buffer on index performance. At one million records, the buffer constitutes approximately 1% of the total data size; this results in the buffer being sampled from with greater frequency (as it has more total weight) than would be the case with larger data. The greater the frequency of buffer sampling, the more rejections will occur, and the worse the sampling performance will be. This illustrates the importance of keeping the buffer small, even when a scan is not used for buffer sampling. Finally, Figure 6l shows the decreasing per-sample cost as the number of records requested by a sampling query grows for DE-IRS, compared to AGG B+tree. Note that DE-IRS benefits significantly more from batching samples than AGG B+tree, and that the improvement is greatest up to $k = 100$ samples per query.



(a) External Insertion Throughput

(b) External Sampling Latency

(c) Concurrent Insert Latency vs. Throughput

(d) Concurrent Insert Throughput vs. Thread Count

Fig. 7. External and Concurrent Extensions of DE-IRS

6.3 External and Concurrent Extensions

Proof of concept implementations of external and concurrent extensions were also tested for IRS queries. Figures 7b and 7a show the performance of the external DE-IRS sampling index against AB-tree. DE-IRS was configured with 4 in-memory levels, using at most 350 MiB of memory in testing, including bloom filters. For DE-IRS, the `O_DIRECT` flag was used to disable OS caching, and CGroups were used to limit process memory to 1 GiB to simulate a memory constrained environment. The AB-tree implementation tested had a cache, which was configured with a memory budget of 64 GiB. This extra memory was provided to be fair to AB-tree. Because it uses per-sample tree-traversals, it is much more reliant on caching for good performance. DE-IRS was tested without a caching layer. The tests were performed with 4 billion (80 GiB) and 8 billion (162 GiB) uniform and zipfian records, and 2.6 billion (55 GiB) OSM records. DE-IRS outperformed the AB-tree by over an order of magnitude in both insertion and sampling performance.

Finally, Figures 7c and 7d show the multi-threaded insertion performance of the in-memory DE-IRS index with concurrency support, compared to AB-tree running entirely in memory, using the synthetic uniform dataset. Note that in Figure 7c, some of the AB-tree results are cut off, due to having significantly lower throughput and higher latency compared with the DE-IRS. Even without concurrent merging, the framework shows linear scaling up to 4 threads of insertion, before leveling off; throughput remains flat even up to 32 concurrent insertion threads. An implementation with support for concurrent merging would scale even better.

7 RELATED WORK

The general IQS problem was first proposed by Hu, Qiao, and Tao [27] and has since been the subject of extensive research [5, 6, 10, 50]. These papers involve the use of specialized indexes to assist in drawing samples efficiently from the result sets of specific types of query, and are largely focused on in-memory settings. A recent survey by Tao [46] acknowledged that dynamization remains a major challenge for efficient sampling indexes. There do exist specific examples of sampling indexes [27] designed to support dynamic updates, but they are specialized, and impractical due to their implementation complexity and high constant-factors in their cost functions. A static index for spatial independent range sampling [50] has been proposed with a dynamic extension similar to the one proposed in this paper, but the method was not generalized, and its design space was not explored. There are also weight-updatable implementations of the alias structure [8, 26, 34] that function under various assumptions about the weight distribution. These are of limited utility in a database context as they do not support direct insertion or deletion of entries. Efforts have also been made to improve tree-traversal based sampling approaches. Notably, the AB-tree [53] extends tree-sampling with support for concurrent updates, which has been a historical pain point.

The Bentley-Saxe method was first proposed by Saxe and Bentley [44]. Overmars and van Leeuwen extended this framework to provide better worst-case bounds [42], but their approach hurts common case performance by splitting reconstructions into small pieces and executing these pieces each time a record is inserted. Though not commonly used in database systems, the method has been applied to address specialized, problems, such as the creation of dynamic metric indexing structures [35], analysis of trajectories [16], and genetic sequence search indexes [9].

8 CONCLUSION

This paper discussed the creation of a framework for the dynamic extension of static indexes designed for various sampling problems. Specifically, extensions were created for the alias structure (WSS), the in-memory ISAM tree (IRS), and the alias-augmented B+tree (WIRS). In each case, the SSIs were extended successfully with support for updates and deletes, without compromising their sampling performance advantage relative to existing dynamic baselines. This was accomplished by leveraging ideas borrowed from the Bentley-Saxe method and the design space of the LSM tree to divide the static index into multiple shards, which could be individually reconstructed in a systematic fashion to accommodate new data. This framework provides a large design space for trading between update performance, sampling performance, and memory usage, which was explored experimentally. The resulting extended indexes were shown to approach or match the insertion performance of the B+tree, while simultaneously performing significantly faster in sampling operations under most situations.

REFERENCES

[1] 2023. *Delicious Dataset.* http://konect.cc/networks/delicious-ti/

[2] 2023. *Open Street Map Dataset.* https://planet.openstreetmap.org/

[3] 2023. *PostgreSQL Documentation.* https://www.postgresql.org/docs/15/sql-select.html

[4] 2023. *Twitter Dataset.* https://github.com/ANLAB-KAIST/traces/releases/tag/twitter_rv.net

[5] Peyman Afshani and Jeff M. Phillips. 2019. Independent Range Sampling, Revisited Again. In *35th International Symposium on Computational Geometry, SoCG 2019, June 18-21, 2019, Portland, Oregon, USA (LIPIcs, Vol. 129)*, Gill Barequet and Yusu Wang (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 4:1–4:13. https://doi.org/10.4230/LIPIcs.SoCG.2019.4

[6] Peyman Afshani and Zhewei Wei. 2017. Independent Range Sampling, Revisited. In *25th Annual European Symposium on Algorithms, ESA 2017, September 4-6, 2017, Vienna, Austria (LIPIcs, Vol. 87)*, Kirk Pruhs and Christian Sohler (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 3:1–3:14. https://doi.org/10.4230/LIPIcs.ESA.2017.3

[7] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. 2013. BlinkDB: queries with bounded errors and bounded response times on very large data. In *Eighth Eurosys Conference 2013, EuroSys '13, Prague, Czech Republic, April 14-17, 2013*, Zdenek Hanzálek, Hermann Härtig, Miguel Castro, and M. Frans Kaashoek (Eds.). ACM, 29–42. https://doi.org/10.1145/2465351.2465355

[8] Daniel Allendorf. 2023. A Simple Data Structure for Maintaining a Discrete Probability Distribution. *CoRR* abs/2302.05682 (2023). https://doi.org/10.48550/arXiv.2302.05682 arXiv:2302.05682

[9] Fatemeh Almodaresi, Jamshed Khan, Sergey Madaminov, Michael Ferdman, Rob Johnson, Prashant Pandey, and Rob Patro. 2022. An incrementally updatable and scalable system for large-scale sequence search using the Bentley-Saxe transformation. *Bioinform.* 38, 12 (2022), 3155–3163. https://doi.org/10.1093/bioinformatics/btac142

[10] Martin Aumüller, Rasmus Pagh, and Francesco Silvestri. 2020. Fair Near Neighbor Search: Independent Range Sampling in High Dimensions. In *Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2020, Portland, OR, USA, June 14-19, 2020*, Dan Suciu, Yufei Tao, and Zhewei Wei (Eds.). ACM, 191–204. https://doi.org/10.1145/3375395.3387648

[11] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. 2019. SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores. In *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, Dahlia Malkhi and Dan Tsafrir (Eds.). USENIX Association, 753–766. https://www.usenix.org/conference/atc19/presentation/balmau

[12] Omri Ben-Eliezer and Eylon Yogev. 2020. The Adversarial Robustness of Sampling. In *Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2020, Portland, OR, USA, June 14-19, 2020*, Dan Suciu, Yufei Tao, and Zhewei Wei (Eds.). ACM, 49–62. https://doi.org/10.1145/3375395.3387643

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

[14] M.G. Bulmer. 1979. *Principles of Statistics*. Dover, New York.

[15] Edith Cohen. 2023. Sampling Big Ideas in Query Optimization. In *Proceedings of the 42nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2023, Seattle, WA, USA, June 18-23, 2023*, Floris Geerts, Hung Q. Ngo, and Stavros Sintos (Eds.). ACM, 361–371. https://doi.org/10.1145/3584372.3589935

[16] Bram Custers, Mees van de Kerkhof, Wouter Meulemans, Bettina Speckmann, and Frank Staals. 2019. Maximum Physically Consistent Trajectories. In *Proceedings of the 27th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, SIGSPATIAL 2019, Chicago, IL, USA, November 5-8, 2019*, Farnoush Banaei Kashani, Goce Trajcevski, Ralf Hartmut Güting, Lars Kulik, and Shawn D. Newsam (Eds.). ACM, 79–88. https://doi.org/10.1145/3347146.3359363

[17] 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, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu (Eds.). ACM, 79–94. https://doi.org/10.1145/3035918.3064054

[18] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2018. Optimal Bloom Filters and Adaptive Merging for LSM-Trees. *ACM Trans. Database Syst.* 43, 4 (2018), 16:1–16:48. https://doi.org/10.1145/3276980

[19] Niv Dayan and Stratos Idreos. 2018. Dostoevsky: Better Space-Time Trade-Offs for LSM-Tree Based Key-Value Stores via Adaptive Removal of Superfluous Merging. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 505–520. https://doi.org/10.1145/3183713.3196927

[20] Niv Dayan and Stratos Idreos. 2019. The Log-Structured Merge-Bush & the Wacky Continuum. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 449–466. https://doi.org/10.1145/3299869.3319903

[21] Niv Dayan, Tamar Weiss, Shmuel Dashevsky, Michael Pan, Edward Bortnikov, and Moshe Twitto. 2022. Spooky: Granulating LSM-Tree Compactions Correctly. *Proc. VLDB Endow.* 15, 11 (2022), 3071–3084. https://www.vldb.org/pvldb/vol15/p3071-dayan.pdf

[22] Bolin Ding, Silu Huang, Surajit Chaudhuri, Kaushik Chakrabarti, and Chi Wang. 2016. Sample + Seek: Approximating Aggregates with Distribution Precision Guarantee. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, 679–694. https://doi.org/10.1145/2882903.2915249

[23] 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 (2021), 26:1–26:32. https://doi.org/10.1145/3483840

[24] Guy Golan-Gueta, Edward Bortnikov, Eshcar Hillel, and Idit Keidar. 2015. Scaling concurrent log-structured data stores. In *Proceedings of the Tenth European Conference on Computer Systems, EuroSys 2015, Bordeaux, France, April 21-24, 2015*, Laurent Réveillère, Tim Harris, and Maurice Herlihy (Eds.). ACM, 32:1–32:14. https://doi.org/10.1145/2741948.2741973

[25] Jarek Gryz, Junjie Guo, Linqi Liu, and Calisto Zuzarte. 2004. Query Sampling in DB2 Universal Database. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004*, Gerhard Weikum, Arnd Christian König, and Stefan Deßloch (Eds.). ACM, 839–843. https://doi.org/10.1145/1007568.1007664

[26] Torben Hagerup, Kurt Mehlhorn, and J. Ian Munro. 1993. Maintaining Discrete Probability Distributions Optimally. In *Automata, Languages and Programming, 20nd International Colloquium, ICALP93, Lund, Sweden, July 5-9, 1993, Proceedings (Lecture Notes in Computer Science, Vol. 700)*, Andrzej Lingas, Rolf G. Karlsson, and Svante Carlsson (Eds.). Springer, 253–264. https://doi.org/10.1007/3-540-56939-1_77

[27] Xiaocheng Hu, Miao Qiao, and Yufei Tao. 2014. Independent range sampling. In *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS'14, Snowbird, UT, USA, June 22-27, 2014*, Richard Hull and Martin Grohe (Eds.). ACM, 246–255. https://doi.org/10.1145/2594538.2594545

[28] Xiaocheng Hu, Miao Qiao, and Yufei Tao. 2015. External Memory Stream Sampling. In *Proceedings of the 34th ACM Symposium on Principles of Database Systems, PODS 2015, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, Tova Milo and Diego Calvanese (Eds.). ACM, 229–239. https://doi.org/10.1145/2745754.2745757

[29] Silu Huang, Chi Wang, Bolin Ding, and Surajit Chaudhuri. 2019. Efficient Identification of Approximate Best Configuration of Training in Large Datasets. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*. AAAI Press, 3862–3869. https://doi.org/10.1609/aaai.v33i01.33013862

[30] Srikanth Kandula, Anil Shanbhag, Aleksandar Vitorovic, Matthaios Olma, Robert Grandl, Surajit Chaudhuri, and Bolin Ding. 2016. Quickr: Lazily Approximating Complex AdHoc Queries in BigData Clusters. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, 631–646. https://doi.org/10.1145/2882903.2882940

[31] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue B. Moon. 2010. What is Twitter, a social network or a news media?. In *Proceedings of the 19th International Conference on World Wide Web, WWW 2010, Raleigh, North Carolina, USA, April 26-30, 2010*, Michael Rappa, Paul Jones, Juliana Freire, and Soumen Chakrabarti (Eds.). ACM, 591–600. https://doi.org/10.1145/1772690.1772751

[32] Feifei Li, Bin Wu, Ke Yi, and Zhuoyue Zhao. 2019. Wander Join and XDB: Online Aggregation via Random Walks. *ACM Trans. Database Syst.* 44, 1 (2019), 2:1–2:41. https://doi.org/10.1145/3284551

[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 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 2071–2086. https://doi.org/10.1145/3318464.3389731

[34] Yossi Matias, Jeffrey Scott Vitter, and Wen-Chun Ni. 2003. Dynamic Generation of Discrete Random Variates. *Theory Comput. Syst.* 36, 4 (2003), 329–358. https://doi.org/10.1007/s00224-003-1078-6

[35] Bilegsaikhan Naidan and Magnus Lie Hetland. 2014. Static-to-dynamic transformation for metric indexing structures (extended version). *Inf. Syst.* 45 (2014), 48–60. https://doi.org/10.1016/j.is.2013.08.002

[36] Frank Olken. 1993. *Random Sampling from Databases.* Ph. D. Dissertation. University of California at Berkeley.

[37] Frank Olken and Doron Rotem. 1986. Simple Random Sampling from Relational Databases. In *VLDB'86 Twelfth International Conference on Very Large Data Bases, August 25-28, 1986, Kyoto, Japan, Proceedings*, Wesley W. Chu, Georges Gardarin, Setsuo Ohsuga, and Yahiko Kambayashi (Eds.). Morgan Kaufmann, 160–169. http://www.vldb.org/conf/1986/P160.PDF

[38] Frank Olken and Doron Rotem. 1989. Random Sampling from B+ Trees. In *Proceedings of the Fifteenth International Conference on Very Large Data Bases, August 22-25, 1989, Amsterdam, The Netherlands*, Peter M. G. Apers and Gio Wiederhold (Eds.). Morgan Kaufmann, 269–277. http://www.vldb.org/conf/1989/P269.PDF

[39] Frank Olken and Doron Rotem. 1995. Random sampling from databases: a survey. *Statistics and Computing* 5 (1995), 25–42. https://doi.org/10.1007/BF00140664

[40] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. 1996. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica* 33, 4 (1996), 351–385. https://doi.org/10.1007/s002360050048

[41] Mark H. Overmars. 1983. *The Design of Dynamic Data Structures.* Lecture Notes in Computer Science, Vol. 156. Springer. https://doi.org/10.1007/BFb0014927

[42] Mark H. Overmars and Jan van Leeuwen. 1981. Worst-Case Optimal Insertion and Deletion Methods for Decomposable Searching Problems. *Inf. Process. Lett.* 12, 4 (1981), 168–173. https://doi.org/10.1016/0020-0190(81)90093-4

[43] Yongjoo Park, Barzan Mozafari, Joseph Sorenson, and Junhao Wang. 2018. VerdictDB: Universalizing Approximate Query Processing. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 1461–1476. https://doi.org/10.1145/3183713.3196905

[44] James B. Saxe and Jon Louis Bentley. 1979. Transforming Static Data Structures to Dynamic Structures (Abridged Version). In *20th Annual Symposium on Foundations of Computer Science, San Juan, Puerto Rico, 29-31 October 1979*. IEEE Computer Society, 148–168. https://doi.org/10.1109/SFCS.1979.47

[45] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The Hadoop Distributed File System. In *IEEE 26th Symposium on Mass Storage Systems and Technologies, MSST 2012, Lake Tahoe, Nevada, USA, May 3-7, 2010*, Mohammed G. Khatib, Xubin He, and Michael Factor (Eds.). IEEE Computer Society, 1–10. https://doi.org/10.1109/MSST.2010.5496972

[46] Yufei Tao. 2022. Algorithmic Techniques for Independent Query Sampling. In *PODS '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Leonid Libkin and Pablo Barceló (Eds.). ACM, 129–138. https://doi.org/10.1145/3517804.3526068

[47] Jeffrey Scott Vitter. 1985. Random Sampling with a Reservoir. *ACM Trans. Math. Softw.* 11, 1 (1985), 37–57. https://doi.org/10.1145/3147.3165

[48] Michael D. Vose. 1991. A Linear Algorithm For Generating Random Numbers With a Given Distribution. *IEEE Trans. Software Eng.* 17, 9 (1991), 972–975. https://doi.org/10.1109/32.92917

[49] A.J. Walker. 1974. New fast method for generating discrete random numbers with arbitrary frequency distributions. *Electronics Letters* 10 (1974), 127–128(1). Issue 8.

[50] Dong Xie, Jeff M. Phillips, Michael Matheny, and Feifei Li. 2021. Spatial Independent Range Sampling. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 2023–2035. https://doi.org/10.1145/3448016.3452806

[51] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, Steven D. Gribble and Dina Katabi (Eds.). USENIX Association, 15–28. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia

[52] 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, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 323–336. https://doi.org/10.1145/3183713.3196931

[53] Zhuoyue Zhao, Dong Xie, and Feifei Li. 2022. AB-tree: Index for Concurrent Random Sampling and Updates. *Proc. VLDB Endow.* 15, 9 (2022), 1835–1847. https://www.vldb.org/pvldb/vol15/p1835-zhao.pdf

[54] Zichen Zhu, Ju Hyoung Mun, Aneesh Raman, and Manos Athanassoulis. 2021. Reducing Bloom Filter CPU Overhead in LSM-Trees on Modern Storage Devices. In *Proceedings of the 17th International Workshop on Data Management on New Hardware, DaMoN 2021, 21 June 2021, Virtual Event, China*, Danica Porobic and Spyros Blanas (Eds.). ACM, 1:1–1:10. https://doi.org/10.1145/3465998.3466002