Towards Systematic Index Dynamization

Douglas B. Rumbaugh
The Pennsylvania State University
drumbaugh@psu.edu

Dong Xie
The Pennsylvania State University
dongx@psu.edu

Zhuoyue Zhao
University at Buffalo
zzhao35@buffalo.edu

ABSTRACT

There is significant interest in examining large datasets using complex domain-specific queries. In many cases, these queries can be accelerated using specialized indexes. Unfortunately, the development of a practical index is difficult, because databases generally require additional features such as updates, concurrency support, crash recovery, etc. There are three major lines of work to alleviate the pain: (1) automatic index composition/tuning which composes indexes out of core data structure primitives to optimize for specific workloads; (2) generalized index templates which generalize common data structures such as B+-trees for custom queries over custom data types, and (3) data structure dynamization frameworks such as the Bentley-Saxe method which converts a static data structure into an updatable data structure with bounded additional query cost. The first two are limited to very specific queries and/or data structures and, thus, are not suitable for building a general index dynamization framework. The last one is more promising in its generality but also has limitations on query types, deletion support, and performance tuning. In this paper, we discuss the limitations of the classic index dynamization techniques and propose a path towards a more general and systematic solution. We demonstrate the viability of our framework by realizing it as a C++20 metaprogramming library and conducting case studies on four example queries with their corresponding static index structures. With this framework, many theoretical/early-stage index designs can easily be extended with support for updates, along with a wide tuning space for query/update performance trade-offs. This allows index designers to focus on efficient data layouts and query algorithms, thereby dramatically narrowing the gap between novel index designs and deployment.

PVLDB Reference Format:
Douglas B. Rumbaugh, Dong Xie, and Zhuoyue Zhao. Towards Systematic Index Dynamization. PVLDB, 17(11): 2867 - 2879, 2024.
doi:10.14778/3681954.3681969

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/psu-db/dynamic-extension.

1 INTRODUCTION

Each year, the database community proposes dozens of novel indexing techniques that improve the performance of particular types of workloads. While some of these new indexes seek to accelerate queries that are already well served, many instead aim to expand the support of database systems for novel types of queries. As the variety of query types in which users are interested continues to explode, these new indexes are increasingly important. Recent examples of this can be seen in areas such as vector databases [19, 23] and DNA sequence search [20, 25], where such specialized indexes are critical for ensuring reasonable query performance.

In spite of this large body of work, the state of database practice remains largely unaltered: the majority of modern indexes are built upon a core set of classical data structures: the B+-tree, LSM-tree, and hash table. This state of affairs does not arise from a lack of interest, but rather from the extensive list of requirements placed upon an index by modern data systems. These requirements include concurrent operations, fault tolerance, insertion and deletion of data, performance tuning, deployment in distributed systems, and more. For example, a recent study found that specialized data structures allow for large improvements in both space and time costs for query processing in Datalog, but demands the introduced index structures support *concurrent updates* [34]. This presents a large barrier to entry for newly proposed data structures, which must manually integrate these features before they can be considered ready for use in practical systems.

There are three major lines of work with the aim of reducing this development burden. The first is automatic index composition/tuning [5, 17, 18]. This technique automatically composes indexes out of a core set of data structure primitives to optimize for particular workloads. Existing work in this area is focused exclusively on 1-D point lookup and range queries. The second is generalized index templates, such as GiST [15, 21] and GIN [14]. These templates generalize commonly used data structures to support custom queries over custom data types. For example, GiST [15, 21] generalizes B+-tree for efficient predicate filtering against a variety of user-defined, complex data types, and GIN [14] generalizes inverted indexes to support text search in unstructured/semi-structured data types (e.g., text documents, JSON objects etc.). Both of the aforementioned techniques can only support a limited set of queries that can be answered by the particular data structures they are based upon. They cannot automate the development of new indexes based on specialized data structures that do not fit their models, such as succinct tries [43], nor special query types such as independent range sampling [16]. The third approach is the dynamization of existing static data structures. This takes a static data structure and produces an updatable data structure with bounded additional query cost. The most commonly used dynamization approach is the Bentley-Saxe method [6], which has been deployed to provide update support to various static structures in the past [4, 12, 26, 32, 33, 39]. Unfortunately, this method has a number of limitations that restrict its applicability and requires significant per-data-structure customization in practice. Specifically, we note the following limitations: (1)

Proceedings of the VLDB Endowment, Vol. 17, No. 11 ISSN 2150-8097.
doi:10.14778/3681954.3681969

restrictions on the types of query that can be supported, (2) limited support for deletion, and (3) lack of configurability and poor performance.

Of the three lines of work, we believe dynamization is the most promising for producing indexes for novel query types. While there are restrictions on supported query types, these are less onerous than the other two approaches, which are tightly linked to specific data layouts. As a preliminary study, we applied the principles of dynamization to a specific class of data structures for independent range sampling, a query *not* supportable under the classic dynamization models, and achieved good results [33]. This work focuses on alleviating the pain points of dynamization approaches in general, with the aim of producing a generalized framework capable of extending a broader class of static data structures with support for inserts, deletes, and, ultimately, other necessary index features. Overall, we make the following contributions:

(1) We introduce two new classes of search problems to expand the applicability and efficient deletion support of the classic dynamization methods. In addition, we conduct a theoretical analysis on the cost of applying a dynamization framework on these new classes of queries and offer a comprehensive taxonomy of search problems to guide the usage of various techniques. (Section 3)
(2) We propose a more general dynamization framework with well-defined programming interfaces and a wide tuning space to address the poor performance and configurability limitations of the classic dynamization methods. (Section 4.1 and 4.2)
(3) We discuss the basic strategies to extend our framework with other practical index features like concurrency support and fault tolerance. (Section 4.3)
(4) We conduct a comprehensive empirical case study of a prototype version of our framework for a variety of data structures and query types to show how it can support more query/index types and provide better performance than the classic solutions. (Section 5)

Apart from that, we introduce the background on the classic dynamization methods in Section 2. We discuss other related work in Section 6, and conclude the paper in Section 7.

2 BACKGROUND

The Bentley-Saxe Method [6] is one of the most foundational and widely used techniques for the dynamization of static data structures designed to answer specific types of query. In this section, we provide a detailed review of the Bentley-Saxe method and its limitations to motivate our technical contributions.

Formally, given a dataset domain D, a **query** $Q : \mathcal{PS}(D) \times Q \to R$ is[1] a function which accepts a set of records $d \subseteq D$ and a tuple of query parameters $q \in Q$ as inputs, and maps them to a result $r \in R$. For example, consider a 1-dimensional range query Q over the x column a database table $d(x : \texttt{INT4}, y : \texttt{TEXT})$, which is a subset of $D = \texttt{dom(INT4)} \times \texttt{dom(TEXT)}$. The parameters are a closed range $[x_l, x_h] \in Q = \texttt{dom(INT4)} \times \texttt{dom(INT4)}$. The function Q is defined as returning all the records where x is in a closed range $[x_l, x_h]$, i.e., $Q(d, [x_l, x_h]) = \sigma_{x \in [x_l, x_h]} d$. A query can often be answered more efficiently when a data structure *instance* $\dot{\imath} \in \mathcal{I}$ is[2] pre-built or maintained on the queried dataset d. In the 1-D range query example, this data structure could be a conventional search tree (e.g., ISAM, B+-tree, and their variants), or learned index (e.g., ALEX [10], PGM [12], RMI [22], TrieSpline [35], etc.). However, it can be costly to maintain such data structures under updates. This is because some of them are static by design (e.g., ISAM, RMI, TrieSpline), and those that are dynamic may suffer from performance deterioration under adversarial update workloads. This can be seen in the phenomenon of "waves of misery" in R-trees [40] and B-trees [13], where spikes of many node splits occur over short period of time under insertion-heavy workload, as well as in the significant performance regressions experienced by dynamic learned indexes in the face of skewed key distributions [38, 41].

Decomposable Search Problems and the Bentley-Saxe Method. The Bentley-Saxe method was designed under the assumption that many queries supported by specialized data structures can be framed as *decomposable search problems* (**DSP**):

Definition 1 (Decomposable Search Problem). *A search problem is a query $Q : \mathcal{PS}(D) \times Q \to R$ that can be answered with a data structure (design) $\mathcal{I}$ efficiently. It is decomposable if and if only there exists a* $\mathbf{O(1)}$***-time commutative and associative binary query answer combine operator*** $\square$ *such that, $\forall q \in Q, \forall d \subseteq D, \forall d_1, d_2$ where $d = d_1 \uplus d_2$,*

$$Q(d, q) = Q(d_1, q) \square Q(d_2, q)$$

Consider a data structure $\dot{\imath} \in \mathcal{I}$ which answers a DSP and is constructed from a set of records $d \in D$ by a function $\dot{\imath} = \texttt{build}(d)$ with cost $B(|d|)$. This structure can be extended with support for inserts and deletes if it satisfies an additional property,

Definition 2 (Record Identity). *A data structure $\dot{\imath} \in \mathcal{I}$ built over a domain of records D satisfies* record identity *if and only if it supports a function,* $\texttt{unbuild} : \mathcal{I} \to \mathcal{PS}(D)$*, such that* $\dot{\imath} = \texttt{build}(\texttt{unbuild}(\dot{\imath}))$.

This property allows for inserting and deleting records through reconstruction of the data structure,

$$\text{ins}(\dot{\imath}_i, r) = \texttt{build}(\texttt{unbuild}(\dot{\imath}_i) \cup \{r\})$$
$$\text{del}(\dot{\imath}_i, r) = \texttt{build}(\texttt{unbuild}(\dot{\imath}_i) / \{r\})$$

Given a DSP Q and a data structure $\dot{\imath} \in \mathcal{I}$ that can efficiently answer Q, the Bentley-Saxe method provides support for insertion and queries efficiently even if $\mathcal{I}$ is static. It does so by creating up to $h = \lceil \log_2 n \rceil$ data structure instances $\dot{\imath}_0, \dot{\imath}_1, \ldots, \dot{\imath}_{h-1}$, called levels, where each data structure $\dot{\imath}_l$ is empty or contains exactly 2^l unique records such that $\bigcup_{l=0}^{h-1} \texttt{unbuild}(\dot{\imath}_l) = d$. For a particular $n = |d|$, there is a unique set of levels that are non-empty, which forms a binary decomposition of the current dataset size n. Whenever a new element $x \in D$ is inserted into d, there are two possible cases: (1) $\dot{\imath}_0$ is empty, in which case we can construct $\dot{\imath}_0 = \texttt{build}(\{x\})$; (2) $\dot{\imath}_0, \dot{\imath}_1, \ldots, \dot{\imath}_{l'}$ is a maximal sequence of non-empty levels, for some $l' \geq 0$. In this case, we build $\dot{\imath}_{l'+1} = \texttt{build}(\{x\} \cup \bigcup_{l=0}^{l'} \texttt{unbuild}(\dot{\imath}_l))$, possibly creating a new level, and

[1] We use $\mathcal{PS}(S)$ to denote the power set of S.

[2] $\mathcal{I}$ denotes a specific data structure *design*, e.g., B-tree, or TrieSpline, where it is used denote the set of all instances of this data structure for a domain D.

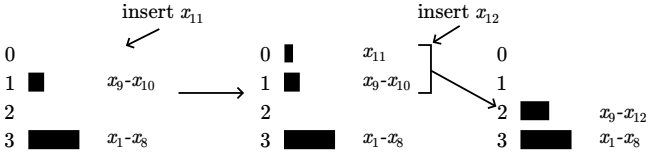


Figure 1: Illustration of inserts in the Bentley-Saxe Method.

then empty $\mathscr{i}_0, \mathscr{i}_1, \ldots, \mathscr{i}_{l'}$. Approaches exist for improving reconstruction performance when $\mathscr{I}$ supports a more efficient construction procedure based on merging existing structures, a property called *merge decomposable* (**MDSP**).

Once again, let's take 1-D range queries using a static learned index (e.g., TrieSpline) as an example (Figure 1). Suppose we already have $n = 10$ elements in $d = \{x_1, x_2, \ldots x_{10}\}$, where the subscripts represent the order in which they were inserted. Since $(10)_{10} = (0101)_2$, we have two empty learned indexes $\mathscr{i}_0$ and $\mathscr{i}_2$, while the other two are $\mathscr{i}_1 = \texttt{build}(\{x_9, x_{10}\})$, and $\mathscr{i}_3 = \texttt{build}(\{x_1, x_2, \ldots, x_8\})$. If we insert x_{11} now, we will create $\mathscr{i}_0 = \texttt{build}(\{x_{11}\})$ since $\mathscr{i}_0$ is empty. If we now insert another element x_{12}, we will unbuild $\mathscr{i}_0$, $\mathscr{i}_1$, build a new learned index $\mathscr{i}_2$ over their union, together with $\{x_{12}\}$, and then empty $\mathscr{i}_0$ and $\mathscr{i}_1$. Clearly, the **amortized** insertion cost into this decomposed structure is $O\left(\frac{B(n)}{n}\log_2 n\right)$ for the first n insertions starting from an empty dataset. In the case of TrieSpline, the $B(n) = O(n)$, so the amortized insertion cost becomes $O(\log_2 n)$, which is on par with standard tree indexes.

Support for efficient insertion in this way may come at a moderate cost to the query complexity. Let $\mathscr{Q}(n)$ be the query complexity for a single instance of $\mathscr{I}$. For any query parameter $q \in Q$, we can utilize the query answer combine operator $\square$ to answer the query over the decomposed structure in $O(\mathscr{Q}(n)\log_2 n)$ time as follows:

$$\mathscr{Q}(q, d) = \square_{l \in [0, \lceil \log_2 n \rceil] \wedge \mathscr{i}_l \texttt{ is nonempty}} \mathscr{Q}(q, \mathscr{i}_l)$$

Here, we slightly abuse the notation $\mathscr{Q}(q, \mathscr{i}_l)$ to mean performing query $\mathscr{Q}(q, \texttt{unbuild}(\mathscr{i}_l))$ using the index $\mathscr{i}_l$. It is worth noting that when $\mathscr{Q}(n)$ is $\Omega(n^\epsilon)$ for some $\epsilon > 0$, then the additional $\log_2 n$ factor is absorbed in the final query complexity. This makes the Bentley-Saxe method appealing as, in the worst-case, it only adds up to a logarithmic factor of query overhead, and, for polynomial time queries, it has no asymptotic effect at all.

(Restricted) Deletion Support. The Bentley-Saxe method also supports deleting records. Most generally, a record d_r can be deleted using the following scheme,

$$\mathscr{i}_r = \texttt{build}(\texttt{unbuild}(\mathscr{i})/d_r)$$

Unfortunately, it is provably impossible [6] to provide reasonable performance bounds of this operation. This conclusion roots from the fact that deletes might not affect a decomposed structure in a deterministic manner. Any record can be deleted at any time, irrespective of where it falls within the decomposed structure. This makes it trivial to force worst-case behavior, such as by deleting the records in the reverse order they were inserted.

However, more efficient delete support is possible, contingent upon additional properties of the search problem. The first approach requires that a search problem has a $O(1)$-time inverse combine operation (Δ), which can negate the effect of a set of records from the query results provided by another set. More formally, it must satisfy the following condition [30],

$$\forall A \in \mathcal{PS}(D), B \subseteq A, q \in Q, \quad \mathscr{Q}(A/B, q) = \Delta(\mathscr{Q}(A, q), \mathscr{Q}(B, q))$$

Such problems are called *invertible* (**INV**) [6]. This allows for deletes to be handled by introducing a second "ghost" structure and inserting deleted records into this. Queries are then run against both the main and the ghost structure, and the inverse combine is used to filter all the deleted records out of the final result. Another approach is to rely upon data-structure level support for deletes, a property called *Deletion Decomposability* (**DDSP**) [30]. For structures with this property, deletes are supported by first using a global dictionary to map each record to the structure to which it belongs (which must be kept up to date as structures are reconstructed). Deletes can then be performed by using this dictionary to find the structure containing the record, and calling that structure's delete routine. A common approach for adding delete support to a static structure is to use *weak deletes*. This approach doesn't remove the deleted record (leaving the structure static), but instead locates it within the structure and marks it as deleted [29]. Queries must be written so as to ignore these deleted records. We will generally consider a DDSP to be a problem supporting deletes by means of weak deletes in this work. Note that, for both DDSP and INV problems, the number of deleted records within a structure is usually unbounded. The commonly proposed solution to this problem is to track the number of deleted records and periodically rebuild the entire structure to remove them when the count exceeds some user-defined threshold [29].

Advantages of the Bentley-Saxe Method. Unlike the other two index design frameworks discussed in Section 1, namely automated index composition/tuning and generalized index templates, the Bentley-Saxe method is not tied to a specific generalization of data structure, data layout, or query type. Thus, it is applicable to a broader range of queries and data structures than these techniques. In addition, static data structures have their own advantages, which the Bentley-Saxe method can, at least in part, preserve. They are often tightly packed, which reduces storage requirements and provides better spatial locality; both of which can have significant impacts on performance [32]. They are also often built with full knowledge of the data distribution, allowing them to avoid query performance deterioration problems due to adversarial workloads.

Limitations of Bentley-Saxe. Despite its clear advantages, there are a number of problems which stand in the way of using the Bentley-Saxe method as a general solution. We note the following three limitations: (1) decomposability is a strong requirement that restricts its applicability; (2) deletion support is dependent on query or data structure specific properties; (3) it suffers from poor average and tail insertion latencies and lacks any performance configuration. Additionally, from the standpoint of ultimately producing a practical database index, the Bentley-Saxe method does not itself have any provisions for features beyond inserts (and sometimes deletes), such as concurrency or fault-tolerance.

3 BEYOND QUERY DECOMPOSABILITY

In this section, we consider extensions to the definition of decomposable search problems that allow the Bentley-Saxe method to

Table 1: Frequently Used Notation

Symbol	Meaning
$\mathscr{Q}_s(n)$	Cost of **local_query**
$C_e(n)$	Cost of **combine**
$P(n)$	Cost of **local_preproc**
$D(n)$	Cost of **distribute_query**
$R(n)$	Number of iterations necessary for an IDSP
$\mathcal{R}$	Query result
q	Query parameters
$\dot{\imath}_i$	Index over data in partition i
q_i	Local query parameters for partition i
$\mathcal{S}_i$	Local query state for partition i
$\mathcal{R}_i$	Local query result for partition i

support more general query types and achieve wider deletion support. Further, we will develop a comprehensive taxonomy of search problems to compare our new definitions with the classic theoretical work and provide a guide of what classes of search problems can be supported effectively by index dynamization.

3.1 Extended Decomposability

The classic Bentley-Saxe method leverages an execution model where the given query is broadcast, identically, to each data partition, and then results are combined. This limits the applicability of the Bentley-Saxe method to decomposable search problems. In this subsection, we will use the problem of Independent Range Sampling (**IRS**) as a pathological case to identify the pain points of query decomposability and pave the way towards supporting more general queries.

Formally, the IRS problem is defined as follows,

Definition 3 (Independent Range Sampling [36]). *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.*

While drawing a sample from a single set of records is easy, it becomes difficult when there are multiple partitions of records. This is because the k samples being drawn need to be distributed across the different partitions in accordance with their weights under the range query being sampled from (i.e., the number of records in the partition that are also contained in the query interval). With the original Bentley-Saxe method, one would have to provide two interfaces for an arbitrary sampling index $\dot{\imath}$ over a key field x as follows: (1) a single (sampling) query interface $\boldsymbol{Q}((k, l, r), \dot{\imath})$ which draws an independent and uniform random sample with replacement of size k from $\sigma_{x \in [l,r]} \dot{\imath}$; (2) a combine operator $S_1 \square S_2$ which combines two samples of size k into a single sample of size k. However, as shown in Algorithm 1, this requires augmenting each sampled record with its inverse sampling probability (`IRSQueryD` in Algorithm 1). Even with this adjustment, the combine operator cannot be implemented in constant time in terms of sample size k (`IRSCombineD` in Algorithm 1).

In fact, plugging Algorithm 1 into the original Bentley-Saxe method, we obtain a query algorithm that runs in $O(\log^2 n + k \log n)$ time, even though the underlying IRS data structure can achieve

Algorithm 1: IRS with decomposability

Input: k: sample size, $[l, r]$: key range to sample from
Output: S: a sample size of size k where each sample is additionally annotated with its inverse probability

```
1  def IRSQueryD((k,l,r), ί)
       //draw a sample of size k in range [l,r]
2      S_0 ← ί.sample(k, l, r) ;
       //count how many records are in range [l,r]
3      c ← ί.count(l, r) ;
4      return S_0.map(lambda x : (x, c)) ;
5  def IRSCombineD (S_1, S_2)
6      if S_1 = φ or S_2 = φ then
7          return S_1 ∪ S_2 ;
8      k ← |S_1| ; //both sets have size k
9      c_1 ← S_1.reduce(lambda (c0, (x, c)) : c0 + c, 0)/k ;
10     c_2 ← S_2.reduce(lambda (c0, (x, c)) : c0 + c, 0)/k ;
11     k_1 ← Binomial(k, 1.0 × (c_1 + c_2)/c_2) ;
       //sample(k', S') draws k' independent and
         uniform samples from set S', which is
         O(|S'| + k) time
12     S ← sample(k_1, S_1) ∪ sample(k_2, S_2) ;
13     return S.map(lambda (x, _) : (x, c_1 + c_2)) ;
```

the optimal $O(\log n + k)$ time [16]. Astute readers might have noticed that the `IRSCombineD` function is non-constant in terms of sample size k, and thus does not satisfy the original Bentley-Saxe method's requirement that the combine operator run in $O(1)$ time. [29] denoted this more general class of search problems as $C(n)$-*Decomposable Search Problems* (**C(n)-DSP**), where the Bentley-Saxe method can only provide $O((\mathscr{Q}(n) + C(n)) \log n)$ worst-case complexity. Practically speaking, if a search problem has to admit a non-$O(1)$ time combine function, it generally involves larger intermediate result sizes and more expensive combine operators, which makes it much less attractive than bespoke dynamic indexes. In the particular case of IRS, it might be better to use a standard aggregate sampling B-tree [27], which supports updates and has a slightly lower $O(k \log n)$ query complexity.

In [33], we addressed this problem by introducing a two-stage sampling query process involving preprocessing to obtain the weights of each partition and assigning appropriate sample sizes to each partition. In this work, we formally extract an abstraction of this two-stage query process to make it work for a broader class of search problems, which we call *extended Decomposable Search Problems* (**eDSP**). In eDSP, a query is answered by running identical preliminary queries against each partition, and using the results of these to produce individualized local queries, and combining these local results to create the query answer. This process is represented using the following interface,

- **local_preproc**$(\dot{\imath}_i, q) \to \mathcal{S}_i$
  Pre-processes each partition $\mathcal{D}_i$ using index $\dot{\imath}_i$ to produce meta-information objects $\mathcal{S}_i$ about the query result within this partition.
- **distribute_query**$(\mathcal{S}_1, \ldots, \mathcal{S}_m, q) \to q_1, \ldots, q_m$
  Processes the list of meta-information $\mathcal{S}_i$ and emits a list of local query parameters q_i.

Algorithm 2: IRS with Extended Decomposability

```
Input: k: sample size, [l, r]: key range to sample from
def local_preproc(q = (k, l, r), ι_i)
    //count how many records are in range [l, r]
    return ι_i.count(l, r) ;
def distribute_query(S_1, ..., S_m, q = (k, l, r))
    //Determine local query based on local states
    k_1, ... k_m ← multinomial(k, S_1, ... S_m) ;
    for i = 1..m do
        q_i ← (k_i, l, r) ;
    return q_1 ... q_m ;
def local_query(ι_i, q_i = (k_i, l, r))
    //Sample using the local sample size
    return ι_i.sample(k_i, l, r) ;
def combine(R_1, ..., R_m, q = (k, l, r))
    return ⋃_{i=1}^{m} R_i ;        //Union results together
```

- **local_query**$(\dot{\iota}_i, q_i) \rightarrow \mathcal{R}_i$
  Executes the local query q_i over partition $\mathcal{D}_i$ using index $\dot{\iota}_i$ and returns a partial result $\mathcal{R}_i$.
- **combine**$(\mathcal{R}_1, \ldots \mathcal{R}_m, q) \rightarrow \mathcal{R}$
  Combines the partial results to produce the final answer.

Next, we use IRS as an example of using the eDSP interface and analyze its query complexity (see Algorithm 2). During query processing, we first run **local_preproc** to obtain meta-information about each partition. In IRS, these are local partition weights. Then we invoke **distribute_query** to aggregate the meta-information and generate individualized parameters for each partition. In IRS, these parameters are the random sample sizes drawn from a multinomial distribution with the local partition weights. After that, we execute each local query $\mathcal{Q}_i$ with its corresponding index using **local_query**, and finally call **combine** over all the results to obtain the final result. In IRS, **local_query** can easily be implemented as an IRS query over the local sampling index with the same query range and the previously computed random sample size, and **combine** is simply the union of all samples. To reduce the combination overhead, we switched from a binary operator that is applied pairwise over the local query results, to a variadic one that may process all the query results at once.

Let $\mathcal{Q}_s(n)$ be the worst-case cost of **local_query**, $C_e(n)$ be the cost of **combine**, $P(n)$ be the cost of **local_preproc**, and $D(n)$ be the cost of **distribute_query**. Then, the worst-case cost of answering an extended-decomposable query against an index decomposed using the Bentley-Saxe method is,

$$O\left(\log_2 n \cdot P(n) + D(n) + \log_2 n \cdot \mathcal{Q}_s(n) + C_e(n)\right)$$

Algorithm 2 demonstrates the usefulness of preprocessing for improving query complexity. In particular, the ability to determine the number of records within the query range allows for individualized local queries to be constructed, which can be executed independently and combined trivially using a simple union. Contrasting this with Algorithm 1, the eDSP version is not only far simpler, but is also asymptotically and practically faster for a typical sample size in the thousands. For the dynamic IRS structure we create based on eDSP, this interface enables sampling with a cost of $O(\log^2 n + k)$

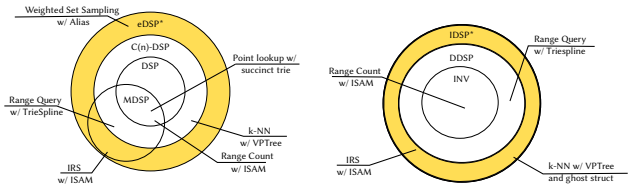
Algorithm 3: K-nn with Iterative Decomposability

```
Input: k: result size, p: query point
def local_preproc(q = (k, p), ι_i))
    //Initialize local query state
    return ι_i.initialize_state(k, p) ;
def distribute_query(S_1, ..., S_m, q = (k, p))
    for i ← 1 ... m do
        //Local queries contain their current
          index traversal state
        q_i ← (k, p, S_i) ;
    return q_1 ... q_m ;
def local_query(ι_i, q_i = (k, p, S_i))
    //Get k nearest neighbors from the structure
    (r_i, S_i) ← ι_i.knn_from(k, p, S_i) ;
    return (r_i, S_i) ; //The local result includes the
     records and query state
def combine(R_1, ..., R_m, q = (k, p))
    R = {} ;
    pq ← PriorityQueue() ;
    for i ← 1 ... m do
        pq.enqueue(i, D(p, R_i.front())) ;
    while |R| < k ∧ there are still local results do
        i, d_1 ← pq.dequeue() ;
        //Ensure records from ghost structures
          sort directly before their corresponding
          record.
        if is_from_ghost(R_i).front() then
            Omit ghost and its record from the result set
        else
            R ← R ∪ R_i.dequeue() ;
            pq.enqueue(i, D(p, R_i.front())) ;
    return R ;
def repeat(q = (k, p), R, q_1, ... q_m)
    missing ← k − R.size() ;
    if missing > 0 then
        for i ← 1 ... m do
            q_i ← (missing, p, q_i.S_i) ;
        return (True, q_1 ... q_m) ;
    return (False, q_1 ... q_m) ;
```

as opposed to the $O(\log^2 n + k \log n)$ cost of Algorithm 1. This solution achieves the same complexity as our bespoke dynamization framework for sampling indexes in [33] did for IRS. However, instead of being a highly specialized system, eDSP provides a general interface that can be used to solve a variety of problems (e.g., kNN which will be described in more detail shortly).

3.2 Iterative Deletion Decomposability

Efficient support for deletion using the Bentley-Saxe method is provably impossible in the general case [6]. Section 2 discussed two special properties that enable efficient deletion: (1) INV where one uses a ghost structure to store deleted records and perform inverse combine to produce final results; (2) DDSP where one relies on data structure specific weak deletions to ignore deleted records. Of the



(a) General Taxonomy **(b) Deletion Taxonomy**

Figure 2: An overview of the Taxonomy of Search Problems, as relevant to our discussion of data structure dynamization. Our proposed extensions are marked with an asterisk (*) and colored yellow.

two, INV is preferable as it places no additional requirements on the data structure and doesn't require modification of the query, aside from the addition of the inverse merge operation. Thus, it seems reasonable to consider approaches for extending the set of search problems that can be supported in an invertible manner.

One major limitation of invertible queries is that the size of the result is not controllable. For queries such as top-k, k-NN, and sampling, the query result must be of a specified size. However, the total size of the result cannot be known until *after* the local results have been merged.[3] In principle, such queries could be answered by repeating the entire query (expanding the value of k for deterministic queries, or repeating as is for sampling queries), until a sufficient number of records has been obtained. But this requires throwing away and repeating a large amount of work. Thanks to the local query state objects introduced in our definition of eDSP, a more efficient solution is available. From this observation, we propose a new class of *Iterative Deletion Decomposable* search problems (**IDSP**). Iterative Deletion Decomposability extends the definition of extended decomposability, with a fifth operation,

- **repeat**$(Q, \mathcal{R}, Q_1, \ldots, Q_m) \to (\mathbb{B}, Q_1, \ldots, Q_m)$
  Evaluates the combined query result in light of the query. If a repetition is necessary to satisfy constraints in the query (e.g., result set size), optionally update the local queries as needed and return true. Otherwise, return false.

If this routine returns true, then the query is repeated from the `distribute_query` stage. Otherwise, the result set is returned. The solution to an iterative decomposable query, which requires $R(n)$ iterations, will have a worst-case query time of,

$$O\left(\log_2 n \cdot P(n) + R(n)\left(D(n) + \log_2 n \cdot Q_s(n) + C_e(n)\right)\right)$$

Note that it is very important for $R(n)$ to be *bounded* for such queries. This can be handled by bounding the number of deleted records within the structure, such as by using the heavy-handed full reconstruction approach [6, 29], or partial-reconstruction based techniques [33].

As an example, consider k-NN queries using a ghost structure for deletes (see Algorithm 3). Following `combine`, sufficiently many ghost records could possibly be present in the local results to drop the size of the output below k. In this case, the query can be repeated by storing local state information, $\mathcal{S}_i$, within the local query object, allowing it to be "resumed" from where it left off to return more records. The local queries can then be repeatedly performed, requesting the relevant number of missing records, until a sufficient number of non-deleted records is returned. This algorithm also demonstrates a complex `combine` operation that is facilitated by the eDSP variadic combine function. Performing this same task using a binary operator would be both slower and more complicated.

3.3 A Taxonomy of Search Problems

In this subsection, we combine our new definitions with classical ones to yield a taxonomy of search problems. We summarize this taxonomy with the Venn diagram in Figure 2, which shows the relationship among the different classes of search problems mentioned thus far, and places several important search problems within this taxonomy. For clarity, the general taxonomy (Figure 2a) is presented as separate from the taxonomy of search problems with efficient deletion support (Figure 2b). Importantly, these two taxomomies are not inclusive of all possible search problems. Queries which can be positioned within these taxomomies are supported by our techniques; however, there exist types of query (most notably, graph queries) which do fall into this framework and are thus not supportable using our techniques.

Figure 2a shows the various search problem classes discussed so far. ISAM, TrieSpline [35], and succinct trie [43] based queries are considered MDSPs because the data structures support a more efficient construction via merging sorted runs of records, rather than re-sorting in full each time. VPTrees [42] and alias structures [37], on the other hand, must be fully reconstructed and don't admit any merge-based optimizations. Sampling queries benefit greatly from the eDSP query interface, and so are classified as eDSPs rather than $C(n)$-DSPs. k-NN requires performing a second round of k-NN in order to select the k-closest records from the result set during `combine`, and so is classified here as a $C(n)$-DSP. Range queries could be considered either a normal DSP, or $C(n)$-DSP, depending on whether the cost of copying records from the local results into the query result is considered,[4] and is a $C(n)$-DSP if the final result set should be sorted. Point lookups against unique indexes are normal DSPs, with `combine` simply accepting the record.

Figure 2b shows the various search problem classes pertaining to deletes. We have positioned INV as a subset of DDSP, because in principle any INV search problem admits a DDSP solution where the data structure itself contains a ghost structure for its records, and the `local_query` operation handles the inverse combine internally. Our definition of IDSP allows delete support for both queries that would otherwise not be supported at all (such as IRS), and with queries for which ghost structures would not otherwise be possible (VPTree). Range counts are considered fully invertible because their results can be merged and inverted in constant time (using + for $\square$ and $-$ for Δ), although these also can be answered less efficiently using weak deletions under deletion decomposability. On the other

[3] A similar problem is faced by sampling, even if it is supported using weak deletes, due to statistical requirements for independence [33]. The approach discussed here works for these queries as well.

[4] In [33], we assumed a constant-time combine operation. In that work, the supported queries allowed for the writing of sampled records directly into a result set, without materializing and combining local result sets explicitly. This is *not* generally possible, and relied on specific properties of the query. As a result, we do not make the same assumption here, when considering a more general problem.

hand, a full range query is *not* considered invertible because the Δ operator would need to perform a merge of sorted runs to properly cancel out deletes. K-NN, as we've already seen, requires the IDSP interface to support ghost structures, and sampling requires the IDSP interface to support weak deletes. Sampling, generally, cannot support the Δ operator and thus cannot support INV deletes with a ghost structure at all under the original INV.

4 FRAMEWORK IMPLEMENTATION

This section details our reference implementation of a framework based upon the above theoretical discussion. Our implementation uses meta-programming in C++20 with concepts to allow the user to plug in their own data structures and queries, so long as they implement the necessary interfaces.

4.1 Interfaces

Records. The framework builds data structures containing user-defined records. These records must be a C++ `struct` with an equality comparison operator, but are otherwise unconstrained. The framework itself makes no assumptions about record contents, ordering among records (or lack thereof), etc. Variable-length data can be supported in off-record storage using pointers. The framework wraps each record automatically with a header for facilitating deletion support.

Shards. Within our framework, the underlying data structure is represented using an abstraction called a *shard*. Shards must be constructable in two ways: (1) from a set of records and (2) from a set of shards. The former constructor allows for shard construction from arbitrary sets of unordered records, while the latter facilitates various optimizations for the construction of shards for MDSPs. Shards must also expose a point lookup operation,[5] which is used by the framework for delete support for DDSPs.

Queries. The query object should implement all four interfaces for eDSP (Section 3.1). Further, it may provide an additional version of `local_preproc` and `local_query` for an unsorted array of records,[6] and `repeat` to support IDSP problems.

4.2 Internal Structure and Mechanisms

Our framework partitions data using a relaxed version of the classic Bently-Saxe method's binary decomposition, along with a mutable (unsorted) buffer into which records are initially inserted. When the mutable buffer's size reaches a *user-specified capacity*, N_B, it is flushed to produce a shard. These shards are arranged into a series of increasingly large *levels*. A configurable parameter called the *scale factor* controls the level growth: The ith level has a capacity of $N_B \times s^{i+1}$ records. Unlike in Bentley-Saxe, these levels can be partially full, which facilitates the less aggressive reconstruction procedure shown in Algorithm 5. Additionally, the records on each level can be arranged into either a single shard, or in up to s shards with capacity $N_B \times s^i$ each. We call this decision the *layout policy*, with the former policy being called *leveling* and the latter *tiering*.

[5]For data structures which do not natively support efficient point-lookups, this feature can be added by including auxiliary data structures, such as hash tables, to the shard.
[6]In the worst case, when there is no trivial solution for queries on unsorted array, one can still call `build` on these data to get a temporary static data structure and answer the query using it.

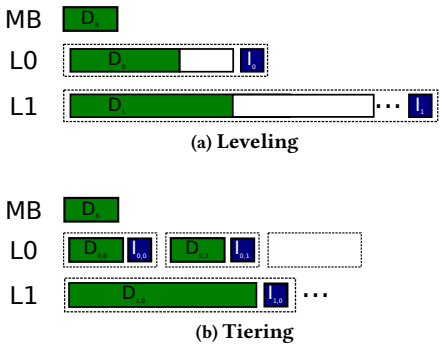


Figure 3: An overview of the general structure of the dynamization framework using (a) leveling and (b) tiering layout policies, with a scale factor 3. Each shard is shown as a dotted box, wrapping its associated dataset (D_i) and index (I_i).

Algorithm 4: Query with Dynamization Framework

Input: q: query parameters, b: mutable buffer, S: static index shards at all levels
Output: R: query results

$\mathcal{S}_b \leftarrow \texttt{local\_preproc}_{buffer}(b, q)$; $\mathcal{S} \leftarrow \{\}$;
for $s \in S$ **do**
  $\mathcal{S} \leftarrow \mathcal{S} \cup (s, \texttt{local\_preproc}(s, q))$;
$(q_b, q_1, \ldots q_m) \leftarrow \texttt{distribute\_query}(\mathcal{S}_b, \mathcal{S}, q)$;
$\mathcal{R} \leftarrow \{\}$; `rpt` $\leftarrow \perp$;
do
  $locR \leftarrow \{\}$;
  $locR \leftarrow locR \cup \texttt{local\_query}_{buffer}(b, q_b)$;
  for $s \in S$ **do**
    $locR \leftarrow locR \cup \texttt{local\_query}(s, q_s)$
  $\mathcal{R} \leftarrow \mathcal{R} \cup \texttt{combine}(locR, q_b, q_1, \ldots, q_m)$;
  $(\texttt{rpt}, q_b, q_1, \ldots, q_m) \leftarrow \texttt{repeat}(q, \mathcal{R}, q_b, q_1, \ldots, q_m)$;
while `rpt`;
return $\mathcal{R}$

An overview of the framework is shown in Figure 3, showing the leveling and tiering layout policies. Note that the classic Bentley-Saxe method (*BSM*) is a special case where the layout policy is leveling with $s = 2, N_B = 1$, and the reconstruction procedure eagerly reconstructs a final target level from a sequence of full (source) levels.

Query Procedure. Queries are processed according to the iterative deletion decomposable interface described in Section 3.2 (Algorithm 4). The worst-case cost of answering a query is,

$$O\left(\mathcal{P}(n, N_B) + R(n)\left(D(n) + \mathcal{Q}_B(N_B) + \log_2 n \cdot \mathcal{Q}_s(n) + C_e(n)\right)\right)$$

where $\mathcal{P}(n, N_B) = P_B(N_B) + P(n) \cdot \log_s n$ is the preprocessing cost for the buffer and the shards, and $\mathcal{Q}_B(N_B)$ is the cost of querying the buffer.

Insertion Procedure. The insertion procedure is shown in Algorithm 5. Records are inserted by wrapping them with a framework header and appending them to the end of the mutable buffer. If

Algorithm 5: Insertion with Dynamization Framework

```
Input: r: new record to insert
if buffer is not full then
    buffer.append(r);
    return
idx ← 0;
for i ← 0 ··· n_levels do
    if level_i can hold records in level_{i-1} then
        idx = i;
        break;
for i ← idx ··· 1 do
    if layout_policy = LEVELING then
        level_i ← merge_shards(level_i, level_{i-1}) ;
    if layout_policy = TIERING then
        new_shard ← merge_shards(level_{i-1}) ;
        level_i ← add_shard(level_i, new_shard) ;
level_0 ← add_shard(level_0, build_shard(buffer));
buffer.append(r);
return
```

the buffer is full, it must be flushed before the new record is inserted. This is done by reconstructing the first level (target) using the records currently within it, and those in the buffer (source). If there is insufficient space in the first level for the new records, then this same procedure is recursively applied until a level that can sustain a reconstruction is found. When using leveling, this reconstruction occurs by building a new shard using the records from the source level or buffer, and the target level, and replacing the shard in the target with the new one. For tiering, a new shard is created using *only* the records from the source level or buffer, and then this new shard is placed within the target level.

Consider a data structure which requires $C_r(n)$ cost to build from a sequence of shards of the same type. The framework will contain $\log_s n$ levels. The amortized insertion cost of the record is the total cost associated with moving the record from the buffer to the final level of a structure with n records, amortized over the number of records. Because N_B is a small constant, the costs associated with the buffer insert and flush can be neglected, giving us an amortized insertion cost of,

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

Note that the difference between layout policies is a constant factor, s, and so it does not show up in these asymptotic analyses. However, the layout policy does affect real performance, and this effect will be demonstrated in Section 5.1.

Delete Procedure. Our implementation supports deletes using two mechanisms: *tagging* and *tombstones*. These mechanisms support the two classes of delete-supporting search problems: tagging for DDSP and tombstones for INV. IDSP support functions transparently with both mechanisms. These two mechanisms differ slightly in the details of their implementation from the theoretical procedures for these classes of problem discussed in Section 2.

Traditionally, DDSPs use a framework-level point-lookup to identify the shard containing the record to be deleted, and then a shard-level delete to remove it. The overhead of maintaining this structure during reconstructions is non-trivial, and so we decided instead to push the point-lookup onto the shard level. Thus, each shard for a DDSP must implement a `point_lookup` routine. Deletes are then performed by calling this routine on each shard (and the buffer) until the record is found, and tagging it as deleted by setting a bit in its header. Assuming the cost of `point_lookup` to be $L(n)$, then the worst case cost of a tagged delete will be,[7]

$$O\left(N_B + L(n)\log_s n\right)$$

Tombstone-based deletes for INV problems work by inserting a new record that is identical to the one being deleted, except with a tombstone bit set in its header. As a result, tombstone deletes have the same cost as a normal insert. One benefit of this design is that tombstones and records appearing together in the local results simplifies the interface, as `combine` can both merge records and cancel deleted ones at the same time, rather than needing two separate routines. Tombstones also allow a principled solution to a significant drawback of ghost structures: difficulty in controlling the number of deleted records in the structure. This is because tombstones, being stored within the *same* structure as the records themselves, allow deleted records to be removed automatically during reconstruction, when the tombstone meets its record. It is the responsibility of the user to implement this functionality in their shard construction routine, if it is desired.

Bounding the number of deleted records within the structure is necessary to control the cost of IDSP queries. Neither deletion mechanism removes records directly, but both have properties that can be used to enforce this bound. Tagged records can be dropped during reconstruction, and tombstones will eventually cancel with their corresponding record when they meet during a reconstruction. To provide a strict bound, each shard can keep track of the number of tagged records or tombstones it contains. When this number exceeds the bound, a preemptive reconstruction can be triggered. For tagging, one round of reconstruction suffices to maintain the bound, but for tombstones multiple reconstructions may need to occur before all shards respect the bound. Whichever technique for deletion is used, the user must slightly modify their eDSP routines so as (1) to ignore tagged records in the `local_preproc` and `local_query` if the search problems is DDSP (or its iterative variant); (2) or to filter their result sets based on tombstone records in `combine` if the search problem is INV (or its iterative variant).

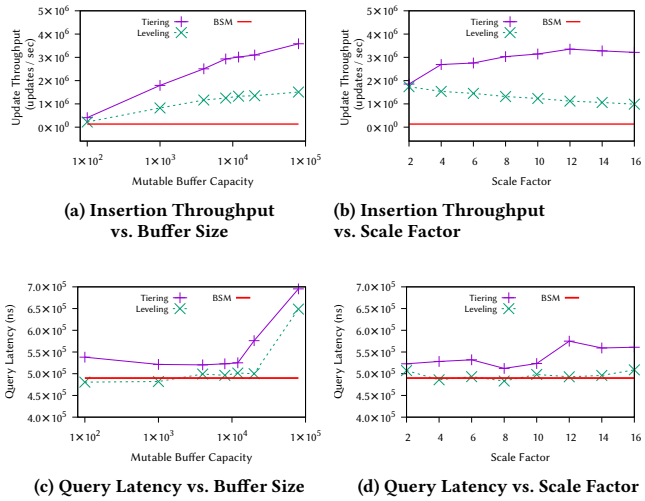
4.3 Further Extensions

Because the framework functions as an append-only structure built over static data, it admits a number of other possible feature extensions beyond dynamization, including concurrency and fault tolerance. These systems are not fully realized within the framework at present, but are left as future work.

Concurrent Operations. The ability to efficiently perform concurrent updates and queries in a serializable manner is important for database indexes. In our framework, inserts, deletes,[8] and queries

[7] Note that N_B will be very small relative to n, but not necessarily relative to $\log_s n$, and so it is left in this cost expression despite us neglecting it for the insert cost.

[8] Tombstones are naturally serialized because they are append-only. When using tagging, however, the static structures will be manipulated in-place. Tagged deletes can still be serialized by time-stamping them, however, and adding this timestamp into the record header.



(a) Insertion Throughput vs. Buffer Size

(b) Insertion Throughput vs. Scale Factor

(c) Query Latency vs. Buffer Size

(d) Query Latency vs. Scale Factor

Figure 4: Design Space Evaluation (Triespline)

can be easily serialized as atomic operations at the framework level, without requiring any data structure changes. This is because all inserts and tombstone deletes are buffer appends, and all data structures are static. This allows a fixed, immutable snapshot to be obtained by each query, made up of a prefix of records in the buffer and a set of the static structures. This model can also help address one of the major performance concerns of dynamization techniques: insertion tail latency. Reconstructions can be performed in the background, while growing the buffer temporarily to sustain further inserts. This can help hide the latencies resulting from the reconstructions. Theoretically, so long as the buffer is allowed to grow, $O(1)$ inserts can be perpetually sustained while the structure gradually catches up. In practice, this isn't feasible because the size of the buffer has negative effects on query performance.

Fault Tolerance. Another important feature for a database index is fault-tolerance/crash recovery. It is necessary for most databases to provide durability guarantees, and to be able to quickly recover from system failures. While our implementation of the framework only supports in-memory indexes, and durability is not the topic of this work, it is worth noting that our framework could easily be extended to automatically add crash recovery features to data structures. Trivially, because all inserts pass through the mutable buffer, a write-ahead log could be added to ensure that the records are written to durable storage prior to appearing in the index. Additionally, the static nature of the data structures facilitates periodic background checkpointing of the shards, allowing recovery to be be accelerated by reading backups of the static structures from disk and building new shards from the records inserted into the log following the last checkpoint. Tagged deletes couldn't be reliably preserved in this manner, as they alter the structures in place and could be missed by checkpointing alone, but these operations could also be written to the log and replayed during recovery. Note that this checkpointing approach would require additional interfaces to be implemented by the shard, whereas the pure write-ahead log approach could be done transparently – albeit with worse recovery time.

5 EVALUATION

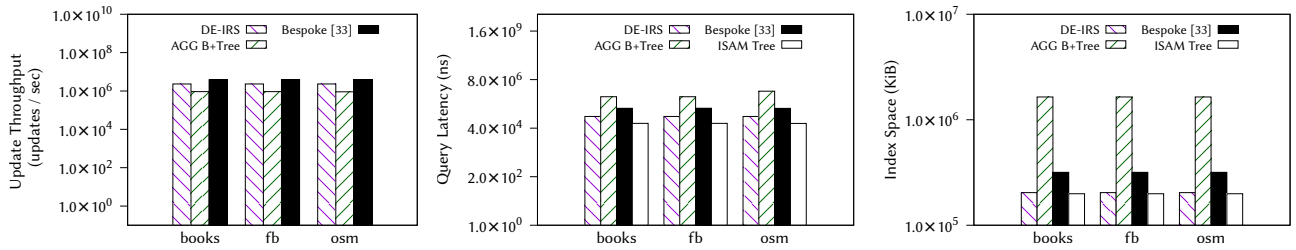
In this section, we demonstrate the effectiveness of our proposed dynamization framework with comprehensive case studies. We first verify its configurability with a parameter sweep. Then, we apply our framework to four different queries and their corresponding static indexes: (1) independent range sampling (IRS) query with ISAM; (2) k-NN query with VP-Tree; (3) string match with fast succinct trie; (4) 1D range queries with learned indexes.

Experimental Setup. Our testing was performed on a dual-socket Intel server with 384GiB of memory and 40 physical cores. Update throughputs were measured starting after 10% of the dataset had been inserted, and include a mixture of 95% inserts and 5% deletes. Query latencies are averages and were measured by repeatedly querying the index after completing all the inserts and deletes. Index size figures do not include the raw storage for the records. We ran all tests on a single-thread and without background compaction, unless otherwise stated. We tested with several datasets,

- For range and sampling indexes, we used the `book`, `fb`, and `osm` datasets from SOSD [24]. Each has 200 million 64-bit keys (to which we added 64-bit values).
- For vector indexes, we used the Spanish Billion Words (SBW) dataset [7], containing about 1 million 300-dimensional vectors of doubles, and a sample of 10 million 128-dimensional vectors of unsigned longs from the BigANN dataset [1].
- For string indexes, we used the genome of the brown bear (ursarc) broken into 30 million unique 70-80 character chunks [2], and a list of about 400,000 English words (english) [3].

5.1 Design Parameter Examination

First, we evaluated a dynamized version of Triespline [35] for answering range queries across a variety of different configuration parameters to assess the effect that these configurations have on the performance of the structure. The full SOSD OSM dataset was used for this testing. In Figures 4a and 4c we examine the effect of buffer size with a fixed scale factor of 8. Figures 4b and 4d show the effect of varying the scale factor with a fixed buffer size of 12000. The Bentley-Saxe method's performance under the same circumstances is represented on each chart by the solid red line. These results show that our leveling policy can approach the Bentley-Saxe method's query performance while generally maintaining better insertion throughput, while tiering allows for even more insertion throughput, at the cost of query latency. Raising the buffer size results in a direct improvement in insertion throughput, however significantly harms query latency over values of around 12,000 due to the cost of scanning the buffer. It's worth noting that this particular buffer capacity is roughly half of our test machine's L1 cache size. It makes sense that it is important to size the buffer to fit well into the cache for good query performance, as this significantly reduces the cost of scanning it. Raising the scale factor has very different results depending upon layout policy. For leveling, higher scale factors hurt insertion performance (due to write amplification) while query latency is relatively unchanged. For tiering, higher scale factors improve insertion throughput at the cost of query latency. Generally, these results demonstrate that the framework provides a large space for performance configuration based on the needs of the situation, allowing for meaningful trade-offs between insertion



(a) Update Throughput (b) Query Latency (c) Index Overhead

Figure 5: IRS Index Evaluation

and query performance to be made. Included in this trade-off space are points which are strictly superior to Bentley-Saxe in terms of both insertion and query performance.

Based on these results, we standardized on a layout policy of tiering, a scale factor of $s = 8$ and a buffer size of about half of the L1 cache size. These were selected to strike a reasonable balance between insertion and query performance, but using the results discussed above, the framework can be tuned according to a user's requirements for query and insertion performance. As a final note, the selection of delete mechanism is largely a function of the search problem. Not all search problems are invertible (which is the requirement for using tombstones). When both are supported, tombstones are generally preferable for better update throughput.

5.2 Case Studies

Independent Range Sampling. We use a static ISAM tree structure for $\mathcal{I}$, which lays the records out in a sorted array and generates internal nodes with routing information on top of this array to accelerate searches, and answer queries using Algorithm 2. This results in the following asymptotic costs for framework operations,

$$\begin{aligned} \text{Insert:} &\quad O\left(\log_s n\right) \\ \text{Query:} &\quad O\left(\log_s n \log_f n + \frac{k}{1-\delta}\right) \\ \text{Delete:} &\quad O\left(\log_s n \log_f n\right) \end{aligned}$$

where δ is a bound on the number of deleted records (0.05) and f is the fanout of the ISAM Tree (16).

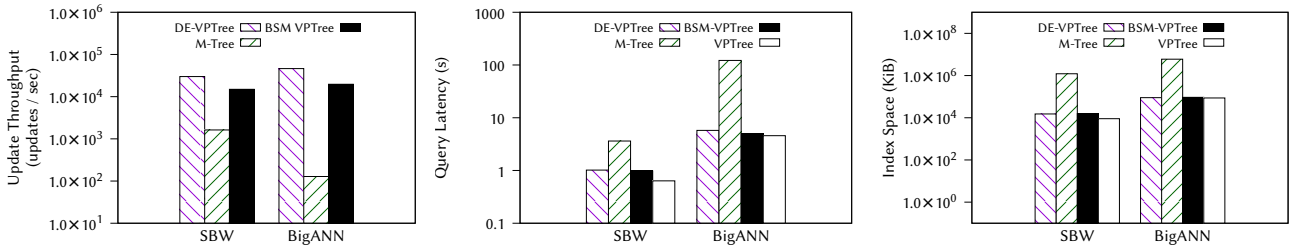
We compare the extended IRS structure produced by our framework (**DE-IRS**) to Olken's method [28] for sampling using a B+-Tree with aggregate weight tags (**AGG B+Tree**), as well as against a single, static instance of the ISAM tree (**ISAM**). We also include the numbers from our previous work [33] which proposed a bespoke dynamization framework for sampling indexes (**Bespoke [33]**) to demonstrate the overhead of generalizing it. We performed random range sampling queries with a controlled selectivity of 0.01% and a sample size of $k = 1000$. **DE-IRS** was configured with $N_B = 12000$, $s = 8$, tiering, and tagging deletes. The results of this testing can be seen in Figure 5. Compared to the conventional, B-tree based solution, DE-IRS sees significantly improved sampling latency *and* update throughputs. Our framework was able to retain some of the static ISAM tree's read performance advantage over the conventional index, while also allowing for high update throughput. In addition, our new generalized framework doesn't introduce significant cost over the specialized implementation.

k-NN Queries on High-dimensional Metric Spaces. We extend the static Vantage Point Tree (VPTree) [42] to support k-NN queries on high dimensional space. A VPTree is a binary tree that is constructed by recursively selecting a point, and partitioning records based on their distance from that point. This results in a hard-to-update structure that can be constructed in $O(n \log n)$ time and can answer k-NN queries in $O(k \log n)$ time. Our dynamized VPTree answers queries using the process detailed in Algorithm 3. This results in the following asymptotic costs for each operation,

$$\begin{aligned} \text{Insert:} &\quad O\left(\log n \cdot \log_s n\right) \\ \text{Query:} &\quad O\left(N_B \log k + k \log n \cdot \log_f n + \log n\right) \\ \text{Delete:} &\quad O\left(\log_s n\right) \end{aligned}$$

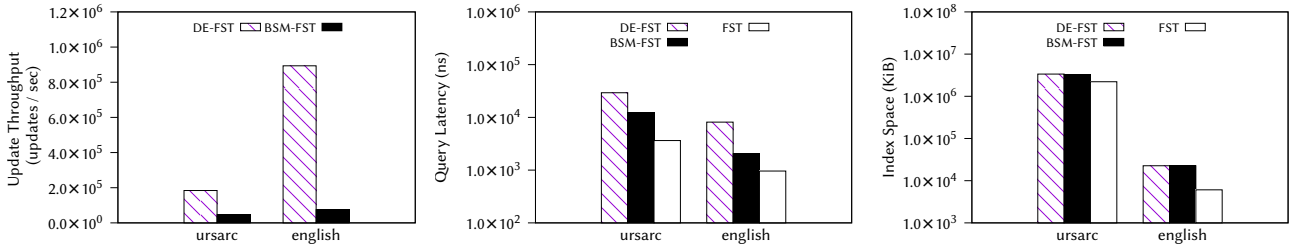
In our experiments, we set up our extended VPTree (**DE-VPTree**) with $N_B = 1400$, $s = 8$, tiering, and delete tagging. We compare it against a dynamic baseline **M-Tree** [9], which partitions records based on high-dimensional spheres and supports updates by splitting and merging these partitions, a standard Bentley-Saxe dynamization of VPTree [26] (**BSM-VPTree**), and a static VPTree (**VPTree**). The tested M-Tree use a random selection process for node splitting. We used L2 distance as our metric, and executed random k-NN queries with $k = 1000$. The results are shown in Figure 6. In this case, the static nature of the VPTree allows it to dominate the M-tree in query latency, and the simpler reconstruction procedure shows a significant insertion performance improvement as well. It's interesting to note that the VPTree performs better, in terms of insertion, on the BigANN dataset. This dataset has more records than SBW, but the records have fewer dimensions. This is due to the repeated reconstructions of the dynamized version benefiting greatly from a faster record comparison function. The M-tree on the other hand sees significant performance degradation as the number of records grows. DE-VPTree sees significantly improved update performance, and similar query performance, to BSM-VPTree.

Exact String Search with Fast Succinct Tries. We next consider point lookups against variable length string data using the Fast



(a) Update Throughput (b) Query Latency (c) Index Overhead

Figure 6: k-NN Index Evaluation



(a) Update Throughput (b) Query Latency (c) Index Overhead

Figure 7: FST Evaluation

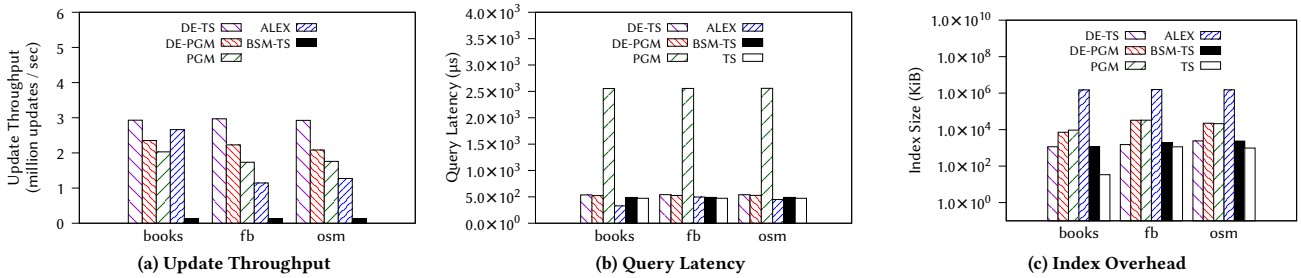
Succinct Trie [43]. This is an example of a succinct data structure, which uses a highly compact representation that is difficult to update. Each shard stores a sorted list of pointers to the strings, along with an instance of the FST built over it.

$$\begin{aligned} \text{Insert:} &\quad O\left(\log_s n\right) \\ \text{Query:} &\quad O\left(N_B + \log n \log_s n\right) \\ \text{Delete:} &\quad O\left(\log_s n\right) \end{aligned}$$

We compare a dynamized version of FST (**DE-FST**) using our framework with tiering, tombstone deletes, $s = 8$ and $N_b = 12000$ with a Bentley-Saxe dynamization (**BSM-FST**) as well as the static baseline (**FST**). Figure 7a shows that the our framework enables a significantly larger update throughput compared to BSM, and Figure 7b demonstrates that it does this without introducing significantly higher query latency due to the less aggressive reconstruction. Further, Figure 7c shows that, particularly for larger data (the ursarc dataset had 100 times more strings than english), the extra storage cost introduced by the framework (due to fragmentation of the data structure) is relatively small.

Range Queries with Learned Indexes. Finally, we examine the application of our framework to static learned indexes for 1D range queries. We compare to two dynamic baselines: **PGM** [12], which uses a Bentley-Saxe inspired system for its own update support, and **ALEX** [10], which is designed from the ground up to support updates. We use the framework to add update support to the static learned index TrieSpline [35] (**DE-TS**), as well as the static version of PGM (**DE-PGM**). We also evaluated a standard Bentley-Saxe dynamization of TrieSpline (**BSM-TS**). DE-TS and DE-PGM both used $N_B = 12000$ and $s = 8$, tiering, and tombstone deletes. Each shard stores D_i as a sorted array of records, uses an instance of the learned index for $\mathcal{I}_i$. The local query routine uses the learned index to locate the first key in the query range and then iterates over the sorted array until the end of the range is reached. The mutable buffer query performs filtering over a full scan. No local preprocessing is needed, and the combine operation combines the result sets. The asymptotic complexity analysis of learned index are not well-established, and so we omit it for our extended indexes.

We examine range queries with a selectivity of 0.1% and exclude the time required to copy records into a result set. Figure 8a shows the update throughput of all competitors. The pure Bentley-Saxe solution performs the worst in all cases by a very large margin, and our DE-TS consistently performs best. Our DE-PGM performs better than PGM in these tests, however note that we are using PGM in its default configuration. PGM supports a similar set of configurable parameters for trading off between insertion and query performance to our own framework. It's inclusion here is simply to demonstrate that our general solution has performance that is on-par with the specialized one used by PGM. ALEX performs relatively well on the books dataset (which has a simple key distribution), but see significant performance regressions for more complex distributions of keys. Figure 8b shows that PGM, in its default configuration, has terrible query performance relative to the other solutions, which all perform fairly similarly beyond that. This result that PGM performs well on inserts and poorly on queries is aligned with the results of [38], and so isn't terribly surprising.



(a) Update Throughput (b) Query Latency (c) Index Overhead

Figure 8: Learned Index Evaluation

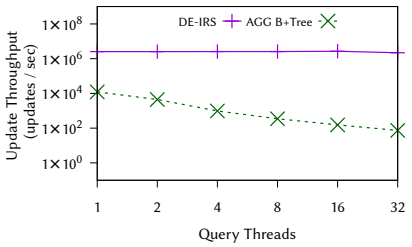


Figure 9: IRS Thread Scaling

Figure 8c shows that the static solutions, dynamized or not, have a massive advantage over ALEX in terms of index size. This is because ALEX, like a B+-Tree, leaves gaps to store future inserts, vastly inflating its storage requirements.

Preliminary Concurrency Support. We additionally demonstrate the capacity of our framework to support concurrent inserts and queries in Figure 9. In this test, a single thread attempts to insert data into the index at a constant rate, while a specified number of query threads repeatedly execute IRS queries. For AGG B+Tree, the aggregate weight tags require locking the entire path, from root to leaf, during inserts and queries, to ensure consistency in weight updates. Our approach has very little interference between the query threads and insert threads, and so maintains its insert performance, compared to the lock-based approach required by AGG B+Tree. This proof-of-concept implementation is not fully developed, but these preliminary results show promise.

6 RELATED WORKS

Bentley-Saxe dynamization techniques have been applied to a large number of problems in the past. For example, [26] applies the techniques to metric indexing structures, but does not apply any special mitigations for performance or concurrency as we do. Other applications, such as to sampling data structures [33, 39], genetic sequence search [31], and learned indexes [12], are highly specialized to a particular data structure or limited set of data structures, and don't present a consistent framework capable of dynamizing arbitrary structures. With this work, we present a dynamization framework capable of covering a very large number of structures, beyond those demonstrated in the paper itself.

Beyond dynamization, other techniques have been applied to ease the creation of indexes. The classic examples of this are the Generalized Search Tree (GiST) [15, 21] and Generalized Inverted Index (GIN) [14]. Like our work, these provide a set of interfaces which the user can implement, based on which a concurrently updatable index is generated. However, GiST is built on a generalization of a search tree and GIN on an inverted index, and so they are limited in their support for specialized queries that cannot be easily represented using their underlying structures. Our approach requires the user to implement more of the index, but in exchange for this extra work allows much more flexibility in terms of query types that can be supported.

Finally, a recent line of work seeks to optimize range indexes for specific workloads by automatically composing them out of a set of structural primitives [5, 8, 11, 17, 18]. These techniques have been shown to be highly effective at creating instance optimized indexes [11], but they are restricted to range indexes, and most of them do not support updates. While similar to our work in the sense that they seek to automate parts of index design, they are targeted at performance optimization, and not broader data structure and/or query support, and so address a different problem than our framework.

7 CONCLUSION

In this paper, we have presented extensions to traditional Bentley-Saxe dynamization targeted at helping to address the three major problems with the technique: limited support for queries due to the requirements of decomposability, limited support for deletes, and poor performance and configurability. Based on these extensions, we developed and presented a general framework that is capable of extending many static data structures with support for inserts and deletes, requiring only a small amount of shim code between the framework and the data structure. We also discussed how this framework could add support for fault-tolerance and concurrency, and including basic preliminary support for concurrency in our presented implementation. Our implementation of this framework was demonstrated across four different types of data structure: learned indexes, metric indexes, sampling indexes, and succinct tries. In all cases, it exhibited good insertion and query performance compared to the Bentley-Saxe method and dynamic baselines, as well as enabling support for queries that otherwise could not be supported in Bentley-Saxe.

REFERENCES

[1] 2024. *BigANN Dataset.* https://big-ann-benchmarks.com/neurips21.html

[2] 2024. *Brown Bear Genome, v1.* https://www.ncbi.nlm.nih.gov/datasets/genome/GCF_023065955.1/

[3] 2024. *English Words Dataset.* https://github.com/dwyl/english-words?tab=readme-ov-file

[4] 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

[5] Darshana Balakrishnan, Lukasz Ziarek, and Oliver Kennedy. 2019. Fluid data structures. In *Proceedings of the 17th ACM SIGPLAN International Symposium on Database Programming Languages*, Alvin Cheung and Kim Nguyen (Eds.). ACM, 3–17. https://doi.org/10.1145/3315507.3330197

[6] Jon Louis Bentley and James B. Saxe. 1980. Decomposable Searching Problems I: Static-to-Dynamic Transformation. *J. Algorithms* 1, 4 (1980), 301–358. https://doi.org/10.1016/0196-6774(80)90015-2

[7] Cristian Cardellino. 2019. Spanish Billion Words Corpus and Embeddings. https://crscardellino.github.io/SBWCE/

[8] Subarna Chatterjee, Meena Jagadeesan, Wilson Qin, and Stratos Idreos. 2021. Cosine: A Cloud-Cost Optimized Self-Designing Key-Value Storage Engine. *PVLDB* 15, 1 (2021), 112–126. https://doi.org/10.14778/3485450.3485461

[9] Paolo Ciaccia, Marco Patella, and Pavel Zezula. 1997. M-tree: An Efficient Access Method for Similarity Search in Metric Spaces. In *VLDB*.

[10] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, David B. Lomet, and Tim Kraska. 2020. ALEX: An Updatable Adaptive Learned Index. In *SIGMOD*.

[11] Jens Dittrich, Joris Nix, and Christian Schön. 2021. The next 50 Years in Database Indexing or: The Case for Automatically Generated Index Structures. *PVLDB* 15, 3 (2021), 527–540. https://doi.org/10.14778/3494124.3494136

[12] Paolo Ferragina and Giorgio Vinciguerra. 2020. The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds. *PVLDB* 13, 8 (2020).

[13] Nikolaus Glombiewski, Bernhard Seeger, and Goetz Graefe. 2019. Waves of Misery After Index Creation. In *Datenbanksysteme für Business, Technologie und Web (BTW 2019), 18. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme" (DBIS), 4.-8. März 2019, Rostock, Germany, Proceedings (LNI, Vol. P-289)*, Torsten Grust, Felix Naumann, Alexander Böhm, Wolfgang Lehner, Theo Härder, Erhard Rahm, Andreas Heuer, Meike Klettke, and Holger Meyer (Eds.). Gesellschaft für Informatik, Bonn, 77–96. https://doi.org/10.18420/BTW2019-06

[14] The PostgreSQL Global Development Group. 2024. GIN Indexes. Retrieved April, 2024 from https://www.postgresql.org/docs/16/gin.html

[15] Joseph M. Hellerstein, Jeffrey F. Naughton, and Avi Pfeffer. 1995. Generalized Search Trees for Database Systems. In *VLDB*, Umeshwar Dayal, Peter M. D. Gray, and Shojiro Nishio (Eds.). Morgan Kaufmann, 562–573. http://www.vldb.org/conf/1995/P562.PDF

[16] Xiaocheng Hu, Miao Qiao, and Yufei Tao. 2014. Independent range sampling. In *PODS*.

[17] Stratos Idreos, Kostas Zoumpatianos, Manos Athanassoulis, Niv Dayan, Brian Hentschel, Michael S. Kester, Demi Guo, Lukas M. Maas, Wilson Qin, Abdul Wasay, and Yiyou Sun. 2018. The Periodic Table of Data Structures. *IEEE Data Eng. Bull.* 41, 3 (2018), 64–75. http://sites.computer.org/debull/A18sept/p64.pdf

[18] Stratos Idreos, Kostas Zoumpatianos, Subarna Chatterjee, Wilson Qin, Abdul Wasay, Brian Hentschel, Mike S. Kester, Niv Dayan, Demi Guo, Minseo Kang, and Yiyou Sun. 2019. Learning Data Structure Alchemy. *IEEE Data Eng. Bull.* 42, 2 (2019), 47–58. http://sites.computer.org/debull/A19june/p47.pdf

[19] Suhas Jayaram Subramanya, Fnu Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnawamy, and Rohan Kadekodi. 2019. DiskANN: Fast Accurate Billion-point Nearest Neighbor Search on a Single Node. In *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.), Vol. 32. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2019/file/09853c7fb1d3f8ee67a61b6bf4a7f8e6-Paper.pdf

[20] Steven Y. Ko, Lauren Sassoubre, and Jaroslaw Zola. 2018. Applications and Challenges of Real-time Mobile DNA Analysis. In *Proceedings of the 19th International Workshop on Mobile Computing Systems & Applications* (Tempe, Arizona, USA) *(HotMobile '18)*. Association for Computing Machinery, New York, NY, USA, 1–6. https://doi.org/10.1145/3177102.3177114

[21] Marcel Kornacker, C. Mohan, and Joseph M. Hellerstein. 1997. Concurrency and Recovery in Generalized Search Trees. In *SIGMOD*, Joan Peckham (Ed.). ACM Press, 62–72. https://doi.org/10.1145/253260.253272

[22] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In *SIGMOD (SIGMOD '18)*.

[23] Yu A. Malkov and D. A. Yashunin. 2020. Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *IEEE Trans. Pattern Anal. Mach. Intell.* 42, 4 (apr 2020), 824–836. https://doi.org/10.1109/TPAMI.2018.2889473

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

[25] Andrew J Mikalsen and Jaroslaw Zola. 2023. Coriolis: enabling metagenomic classification on lightweight mobile devices. *Bioinformatics* 39, Supplement_1 (06 2023), i66–i75. https://doi.org/10.1093/bioinformatics/btad243 arXiv:https://academic.oup.com/bioinformatics/article-pdf/39/Supplement_1/i66/50741350/btad243.pdf

[26] 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

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

[28] Frank Olken and Doron Rotem. 1989. Random Sampling from B+ Trees. In *VLDB*, Peter M. G. Apers and Gio Wiederhold (Eds.). Morgan Kaufmann, 269–277. http://www.vldb.org/conf/1989/P269.PDF

[29] Mark H. Overmars. 1983. *The Design of Dynamic Data Structures.* Lecture Notes in Computer Science, Vol. 156. Springer. https://doi.org/10.1007/BFB0014927

[30] Mark H. Overmars and Jan van Leeuwen. 1981. Two general methods for dynamizing decomposable searching problems. *Computing* 26, 2 (1981), 155–166. https://doi.org/10.1007/BF02241781

[31] Prashant Pandey, Fatemeh Almodaresi, Michael A. Bender, Michael Ferdman, Rob Johnson, and Rob Patro. 2018. Mantis: A Fast, Small, and Exact Large-Scale Sequence-Search Index. In *Research in Computational Molecular Biology - 22nd Annual International Conference*, Benjamin J. Raphael (Ed.), Vol. 10812. Springer, 271–273. https://link.springer.com/content/pdf/bbm%3A978-3-319-89929-9%2F1.pdf

[32] Octavian Procopiuc, Pankaj K. Agarwal, Lars Arge, and Jeffrey Scott Vitter. 2003. Bkd-Tree: A Dznamic Scalable kd-Tree. In *Advances in Spatial and Temporal Databases, 8th International Symposium, SSTD 2003, Santorini Island, Greece, July 24-27, 2003, Proceedings (Lecture Notes in Computer Science, Vol. 2750)*, Thanasis Hadzilacos, Yannis Manolopoulos, John F. Roddick, and Yannis Theodoridis (Eds.). Springer, 46–65. https://doi.org/10.1007/978-3-540-45072-6_4

[33] Douglas B. Rumbaugh and Dong Xie. 2023. Practical Dynamic Extension for Sampling Indexes. *Proc. ACM Manag. Data* 1, 4 (2023), 254:1–254:26. https://doi.org/10.1145/3626744

[34] Arash Sahebolamri, Langston Barrett, Scott Moore, and Kristopher K. Micinski. 2023. Bring Your Own Data Structures to Datalog. *Proc. ACM Program. Lang.* 7, OOPSLA2 (2023), 1198–1223. https://doi.org/10.1145/3622840

[35] Mihail Stoian, Andreas Kipf, Ryan Marcus, and Tim Kraska. 2021. PLEX: Towards Practical Learned Indexing. *CoRR* abs/2108.05117 (2021).

[36] Yufei Tao. 2022. Algorithmic Techniques for Independent Query Sampling. In *PODS*, Leonid Libkin and Pablo Barceló (Eds.). ACM, 129–138. https://doi.org/10.1145/3517804.3526068

[37] 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.

[38] Chaichon Wongkham, Baotong Lu, Chris Liu, Zhicong Zhong, Eric Lo, and Tianzheng Wang. 2022. Are Updatable Learned Indexes Ready? *PVDLB* 15, 11 (2022).

[39] Dong Xie, Jeff M. Phillips, Michael Matheny, and Feifei Li. 2021. Spatial Independent Range Sampling. In *SIGMOD*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 2023–2035.

[40] Lu Xing, Eric Lee, Tong An, Bo-Cheng Chu, Ahmed Mahmood, Ahmed M. Aly, Jianguo Wang, and Walid G. Aref. 2021. An Experimental Evaluation and Investigation of Waves of Misery in R-trees. *PVLDB* 15, 3 (2021), 478–490. https://doi.org/10.14778/3494124.3494132

[41] Rui Yang, Evgenios M. Kornaropoulos, and Yue Cheng. 2023. Algorithmic Complexity Attacks on Dynamic Learned Indexes. *PVLDB* 17, 4 (2023), 780–793. https://www.vldb.org/pvldb/vol17/p780-yang.pdf

[42] Peter N. Yianilos. 1993. Data Structures and Algorithms for Nearest Neighbor Search in General Metric Spaces. In *Proceedings of the Fourth Annual ACM/SIGACT-SIAM Symposium on Discrete Algorithms*, Vijaya Ramachandran (Ed.).

[43] 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 *SIGMOD*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 323–336. https://doi.org/10.1145/3183713.3196931