

**M A S A R Y K  
U N I V E R S I T Y**

FACULTY OF INFORMATICS

**Temporally Dynamic Learned Index  
for Similarity Search over Complex  
Data**

Master's Thesis

BC. EMMA SOMMEROVÁ

Brno, Spring 2026

**MASARYK  
UNIVERSITY**

FACULTY OF INFORMATICS

**Temporally Dynamic Learned Index  
for Similarity Search over Complex  
Data**

Master's Thesis

BC. EMMA SOMMEROVÁ

Advisor: RNDr. David Procházka

Department of Machine Learning and Data Processing

Brno, Spring 2026



## Declaration

Hereby, I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

During the preparation of the thesis, I used the following AI tools:

- Grammarly for grammar check.
- GitHub Copilot for faster code development, writing documentation, and code consultations.
- Claude to improve my writing style.

I declare that I used these tools in accordance with the principles of academic integrity. I checked the content and took full responsibility for it.

Bc. Emma Sommerová

**Advisor:** RNDr. David Procházka

## Acknowledgements

I would like to thank my advisor RNDr. David Procházka, and consultant doc. RNDr. Vlastislav Dohnal, Ph.D. for giving me the opportunity to participate in their research, as well as their continuous guidance, feedback, and patience. I am also grateful to my family, especially my mom, for nurturing my curiosity, and to my friends for encouraging my ambitions and not judging me for drinking coffee at all hours of the day. Computational resources were provided by the e-INFRA CZ project (ID:90254), supported by the Ministry of Education, Youth and Sports of the Czech Republic.

## Abstract

Approximate nearest neighbor (ANN) search over high-dimensional vectors underpins modern retrieval, recommendation systems, and retrieval-augmented generation for large language models. In learned indexing, small neural models direct each query to the relevant region of the dataset. On static collections, learned indexes are competitive with other established methods. However, their advantage disappears in dynamic settings. As inserts and deletes accumulate, the data drifts away from the distribution on which the routing model was trained. Search accuracy degrades, latency increases, and the standard remedy—retraining the model on the entire dataset—becomes computationally prohibitive past tens of millions of vectors. Learned indexes thus remain restricted to static settings.

To this end, we present a general framework that dynamizes any learned index exposing a minimal static interface, by adapting the Bentley-Saxe decomposition to ANN search. Incoming vectors are absorbed into a small in-memory buffer and migrated through compaction into a sequence of static learned indexes (levels), each trained only on its own subset of the data. We develop multi-level search strategies—including one using probability-normalized scoring to resolve cross-level bucket incomparability—compaction mechanisms, rebuild policies for deletions, and retraining strategies for managing distribution shift. We also introduce three diagnostic metrics that decompose recall loss into partitioning and routing quality components.

We instantiate the framework as the Dynamic Learned Index (DLI) using the Learned Metric Index as the internal static learned index, and evaluate it on several large-scale datasets of up to 100M vectors. We study DLI’s performance under continuous insert–delete cycles, distribution shift, and out-of-distribution queries, comparing it against HNSW, IVFFlat, and a periodically retrained LMI. Based on the evaluation, DLI attains the smallest memory footprint—over an order of magnitude smaller than HNSW—while delivering balanced throughput across insert, delete, and search. This positions DLI as a strong candidate for unpredictable dynamic workloads, where current indexes each optimize one axis at the expense of others.

## **Keywords**

learned indexing, dynamic learned index, learned metric index, similarity search, approximate nearest neighbors, dynamicity, distribution shift

# Contents

|                                                                      |           |
|----------------------------------------------------------------------|-----------|
| <b>Introduction</b>                                                  | <b>1</b>  |
| <b>1 Similarity Search and Approximate Nearest Neighbor Indexing</b> | <b>3</b>  |
| 1.1 Similarity Search . . . . .                                      | 3         |
| 1.1.1 Metric Space . . . . .                                         | 3         |
| 1.1.2 Similarity in Embedding Spaces . . . . .                       | 4         |
| 1.1.3 Nearest Neighbor Search . . . . .                              | 5         |
| 1.2 Evaluation Metrics for ANN . . . . .                             | 6         |
| 1.2.1 Search Effectiveness . . . . .                                 | 6         |
| 1.2.2 Search Efficiency . . . . .                                    | 7         |
| 1.3 Traditional Indexing Methods for ANN Search . . . . .            | 7         |
| 1.3.1 Graph-Based Indexes . . . . .                                  | 8         |
| 1.3.2 Partitioning-Based Indexes . . . . .                           | 9         |
| 1.4 Learned Indexing . . . . .                                       | 10        |
| 1.4.1 Degradation Under Dynamic Updates . . . . .                    | 11        |
| 1.4.2 Dynamization of Learned Indexes . . . . .                      | 14        |
| <b>2 Static Learned Index Interface</b>                              | <b>16</b> |
| 2.1 Required Operations . . . . .                                    | 16        |
| 2.2 Index Structure . . . . .                                        | 17        |
| 2.2.1 Expandable Buckets . . . . .                                   | 18        |
| 2.2.2 Routing Model . . . . .                                        | 18        |
| 2.2.3 OID to Bucket Mapping . . . . .                                | 19        |
| 2.3 Insert . . . . .                                                 | 19        |
| 2.3.1 Training Phase . . . . .                                       | 20        |
| 2.3.2 Vector Organization . . . . .                                  | 22        |
| 2.4 Delete . . . . .                                                 | 22        |
| 2.5 Search . . . . .                                                 | 22        |
| 2.6 Performance Evaluation . . . . .                                 | 23        |
| 2.6.1 Understanding Performance Degradation . . . . .                | 24        |
| <b>3 Dynamic Learned Index</b>                                       | <b>30</b> |
| 3.1 Components . . . . .                                             | 31        |
| 3.1.1 Buffer . . . . .                                               | 32        |
| 3.1.2 Levels . . . . .                                               | 32        |

|          |                                             |           |
|----------|---------------------------------------------|-----------|
| 3.2      | Functionality . . . . .                     | 32        |
| 3.3      | Insert . . . . .                            | 33        |
| 3.3.1    | Compaction . . . . .                        | 33        |
| 3.3.2    | Level Retraining . . . . .                  | 34        |
| 3.4      | Delete . . . . .                            | 36        |
| 3.4.1    | Level Underutilization . . . . .            | 36        |
| 3.4.2    | Rebuild . . . . .                           | 37        |
| 3.5      | Search . . . . .                            | 38        |
| 3.5.1    | Search strategies . . . . .                 | 38        |
| <b>4</b> | <b>Experiments</b>                          | <b>41</b> |
| 4.1      | Experiment Methodology . . . . .            | 41        |
| 4.1.1    | Datasets . . . . .                          | 45        |
| 4.1.2    | Hardware . . . . .                          | 45        |
| 4.1.3    | Implementation . . . . .                    | 45        |
| 4.1.4    | Memory measurements . . . . .               | 46        |
| 4.1.5    | Search parameters . . . . .                 | 47        |
| 4.2      | DLI comparison to LMI . . . . .             | 47        |
| 4.2.1    | DLI vs LMI Overall Search Quality . . . . . | 48        |
| 4.3      | Structural Experiments . . . . .            | 49        |
| 4.3.1    | Search Strategy . . . . .                   | 49        |
| 4.3.2    | Compaction Strategy . . . . .               | 50        |
| 4.3.3    | Arity and Buffer Size . . . . .             | 53        |
| 4.4      | Delete Experiments . . . . .                | 54        |
| 4.5      | Training Experiments . . . . .              | 56        |
| 4.5.1    | Retrain Strategy . . . . .                  | 56        |
| 4.6      | Evaluation on multiple Datasets . . . . .   | 58        |
| 4.7      | Comparison to Other Indexes . . . . .       | 59        |
| 4.7.1    | Memory Efficiency . . . . .                 | 59        |
| 4.7.2    | Throughput Comparison . . . . .             | 60        |
| 4.7.3    | Search Quality . . . . .                    | 62        |
| 4.8      | Summary . . . . .                           | 63        |
| <b>5</b> | <b>Conclusion</b>                           | <b>65</b> |
| 5.0.1    | Future Work . . . . .                       | 66        |
| <b>A</b> | <b>An appendix</b>                          | <b>67</b> |
| A.1      | Configuration parameters . . . . .          | 67        |

|                     |                                                     |           |
|---------------------|-----------------------------------------------------|-----------|
| A.1.1               | DLI Configuration . . . . .                         | 67        |
| A.1.2               | LMI Configuration . . . . .                         | 68        |
| A.1.3               | Training Configuration . . . . .                    | 68        |
| A.2                 | Static Learned Metric Index Functionality . . . . . | 70        |
| A.2.1               | Delete . . . . .                                    | 70        |
| A.2.2               | Search . . . . .                                    | 70        |
| A.2.3               | Insert . . . . .                                    | 71        |
| A.3                 | DLI functionality . . . . .                         | 73        |
| A.3.1               | Insert . . . . .                                    | 73        |
| A.3.2               | Delete . . . . .                                    | 75        |
| A.3.3               | Search . . . . .                                    | 77        |
| A.4                 | Experiment Specifications . . . . .                 | 79        |
| A.4.1               | Environment . . . . .                               | 79        |
| A.4.2               | Memory Estimation . . . . .                         | 79        |
| A.4.3               | HNSW Memory Estimation . . . . .                    | 79        |
| A.4.4               | IVFFlat Memory Estimation . . . . .                 | 80        |
| A.5                 | DLI Partitioning and Routing Quality . . . . .      | 80        |
| A.6                 | Arity and Buffer Size Experiments . . . . .         | 80        |
| A.7                 | Delete Experiments . . . . .                        | 87        |
| A.7.1               | Early Stopping and Epochs . . . . .                 | 91        |
| <b>Bibliography</b> |                                                     | <b>93</b> |

## List of Tables

|     |                                                                                 |    |
|-----|---------------------------------------------------------------------------------|----|
| 4.1 | Experimental scenarios. . . . .                                                 | 44 |
| 4.2 | Datasets used in experiments. . . . .                                           | 45 |
| 4.3 | Arity and buffer size experiments. . . . .                                      | 54 |
| 4.4 | Delete experiments — oldest delete method. . . . .                              | 55 |
| 4.5 | Delete experiments — random delete method. . . . .                              | 55 |
| 4.6 | Retrain strategy experiments. . . . .                                           | 57 |
| 4.7 | Evaluation of DLI on multiple datasets. . . . .                                 | 59 |
| 4.8 | Comparison of DLI with other indexes — data and query<br>distributions. . . . . | 60 |
| 4.9 | Comparison of DLI with other indexes — delete workloads. . . . .                | 61 |
| A.1 | DLI configuration parameters. . . . .                                           | 67 |
| A.2 | LMI configuration parameters. . . . .                                           | 68 |
| A.3 | Training configuration. . . . .                                                 | 69 |
| A.4 | Used Metacentrum hardware. . . . .                                              | 79 |
| A.5 | Arity and buffer size experiments — 100M dataset. . . . .                       | 86 |
| A.6 | From Scratch and Full Retrain experiments. . . . .                              | 92 |

## List of Figures

|     |                                                               |    |
|-----|---------------------------------------------------------------|----|
| 1.1 | Effects of Insertions and Deletions on Index Partitions . . . | 13 |
| 2.1 | Static Learned Metric Index Architecture . . . . .            | 17 |
| 2.2 | Static LMI Insert . . . . .                                   | 20 |
| 2.3 | LMI Recall Degradation after Dynamic Updates . . . . .        | 24 |
| 2.4 | LMI Partitioning Quality . . . . .                            | 27 |
| 2.5 | LMI Wasted Effort after Dynamic Updates . . . . .             | 29 |
| 3.1 | Dynamic Learned Index Structure . . . . .                     | 31 |
| 4.1 | Search Quality: LMI vs DLI . . . . .                          | 48 |
| 4.2 | Search Strategy Evaluation . . . . .                          | 50 |
| 4.3 | Compaction Strategy Evaluation — Search . . . . .             | 51 |
| 4.4 | Compaction Strategy Evaluation — Search Time Analysis         | 52 |
| 4.5 | Compaction Strategy Comparison — Insert . . . . .             | 53 |
| A.1 | Partitioning Quality: LMI vs DLI . . . . .                    | 81 |
| A.2 | Routing Quality: LMI vs DLI . . . . .                         | 82 |
| A.3 | Arity and Buffer size: Recall and Cost in Batches . . . . .   | 83 |
| A.4 | Arity and Buffer Size: Recall and Cost at 100M Vectors . .    | 85 |
| A.5 | Rebuild Strategy: Constant Insert . . . . .                   | 88 |
| A.6 | Rebuild Strategy: Medium Insert . . . . .                     | 89 |
| A.7 | Rebuild Strategy: Heavy Insert . . . . .                      | 90 |

## Introduction

Similarity search in high-dimensional vector spaces has become crucial for modern data-intensive applications, including content-based image retrieval [1], recommendation systems [2], protein structure analysis [3, 4], and large language model inference [5], by efficiently finding approximate nearest neighbors within large collections of embedding vectors.

Most indexing methods assume a static dataset available at build time, further reinforced by existing benchmarks [6, 7], that primarily focus on static scenarios. In practice, however, applications must accommodate continuous updates as new content arrives, outdated entries are removed, and the underlying data distribution shifts.

Recent work on learned indexes for similarity search [8, 9] proposes replacing traditional index structures with machine learning models that can be trained to capture the underlying data distribution. However, existing learned indexes [10, 11, 12, 13, 8, 14] assume a static setting where the entire dataset is available during construction, enabling the routing model to be trained on the complete data distribution. When deployed in dynamic environments with continuous insertions and deletions, the routing model produces increasingly inaccurate predictions as the actual data diverges from the distribution it was trained on. The degradation makes learned indexes unsuitable for dynamic workloads unless the index is periodically rebuilt, which is computationally expensive [15].

We address this problem using the temporal dynamicity approach, adapting the Bentley-Saxe decomposition [16] for learned indexes. This thesis introduces a general framework for the dynamization of learned indexes, and creates Dynamic Learned Index (DLI) — an implementation of the proposed framework with Learned Metric Index [14] (LMI) as its base component.

The framework itself is compatible with any static learned index that satisfies a minimal set of operations, and we demonstrate how to adapt a learned index to this interface.

We evaluate DLI on various datasets up to 100M vectors, and compare it against traditional indexes, specifically HNSW [17] and IVFFlat [18], as well as a periodically-rebuilt LMI baseline.

The remainder of this thesis is structured as follows. Chapter 1 reviews the approximate nearest neighbor problem, traditional indexing methods, learned indexes for similarity search. Chapter 2 describes the static learned index interface required by the dynamic framework and demonstrates it on the LMI. Chapter 3 introduces the Dynamic Learned Index (an instantiation of the dynamic framework with LMI) and uses it to describe the architecture and query processing strategy of the framework based on temporal decomposition. Chapter 4 presents experimental evaluation on various datasets, comparing the dynamic framework with baselines and analyzing the impact of design choices. Chapter 5 concludes with a summary of contributions and directions for future work.

# 1 Similarity Search and Approximate Nearest Neighbor Indexing

Many modern applications deal with so-called *complex data*, such as documents [19] or protein structures [4]. *Similarity search* is an area of research focused on this such data and the problem of finding objects that are close to a given query object [20]. Understanding why dynamic learned indexes are important for similarity search requires establishing several concepts: (i) the formalization of similarity, its connection to high-dimensional embeddings, and the need for approximate indexing methods, (ii) the definition of approximate nearest neighbor queries and how they are evaluated, (iii) traditional indexing methods and their limitations, and (iv) the motivation and general principles of learned indexes.

With this context established, at the end of this chapter, we explore why learned indexes degrade under dynamic workloads, categorize existing approaches to dynamization, and explain the choice of temporal decomposition as the dynamization approach explored in this thesis.

## 1.1 Similarity Search

Standard database queries rely on exact matching and ordering of structured data like numerical values, text strings, and timestamps. However, for complex data types, canonical orderings do not correspond to similarity relationships, and exact matches are rare [21, 20]. So, instead of relying on absolute ordering, similarity search focuses on the relationship (*similarity*) between data points and retrieving those that are most similar to a given query.

### 1.1.1 Metric Space

*Metric spaces*, first formalized in topology [22], provide the mathematical foundation for similarity search. By abstracting the concept of similarity into this formalism, we can decouple the algorithm de-

sign from the data domain and thus create general-purpose methods applicable to different data types [21].

A metric space [20] is defined as a tuple  $(\mathcal{D}, \delta)$ , where  $\mathcal{D}$  is a domain of objects and  $\delta : \mathcal{D} \times \mathcal{D} \rightarrow \mathbb{R}$  is a *distance function* which captures the similarity between objects. A true metric space requires  $\delta$  to be a *metric*, meaning it satisfies non-negativity, identity of indiscernibles, symmetry, and triangle inequality [20]. In practice, applications sometimes employ non-metric distance functions that violate these properties.

### 1.1.2 Similarity in Embedding Spaces

Finding appropriate distance functions for complex data is a non-trivial task that requires domain-specific knowledge. Historically, hand-crafted features and domain-specific distance functions were used to capture similarity, such as extracted features for images [23] or edit distance for text [24]. However, these approaches are limited by the quality of manually designed features, require extensive domain expertise, and make rigid assumptions about what constitutes similarity [25].

Deep learning approaches have demonstrated that features can be learned directly from data [26, 27], which led to a shift from engineered features to learned representations [25]. Neural networks are effective at mapping complex data into high-dimensional vector spaces where distances between vectors correspond to semantic similarity of the original data. For example, word embeddings like Word2Vec [28] have shown that meaningful semantic relationships can be captured through vector arithmetic in the learned embedding space. Therefore, modern applications typically embed complex data into vectors using neural networks, then perform all subsequent analysis on these embeddings rather than the original objects.

This thesis focuses on such embedding-based similarity search, assuming that for a given index, inserted data and queries are outputs of the same embedding model and compared using a predefined distance function.

### 1.1.3 Nearest Neighbor Search

Similarity search encompasses various operations, including range query or the reverse nearest neighbor query [20]. However, the most widely used operation is the search for the  $k$  nearest objects to a given query object. Unlike fixed-threshold range queries, which search for all objects within a specified distance of the query object,  $k$ -nearest neighbor (KNN) queries naturally adapt to varying data densities through the parameter  $k$ . Additionally, KNN search can also serve as a building block for many other similarity operations [21].

#### Exact $k$ -Nearest Neighbor Search

Given a metric space  $(\mathcal{D}, \delta)$ , a dataset  $X \subset \mathcal{D}$  of  $n$  objects, a query object  $q \in \mathcal{D}$ , and an integer  $k \in \mathbb{N}$ , the KNN query returns the  $k$  objects in  $X$  with the smallest distance to  $q$  [20]:

$$\text{NN}_k(q, X) = \{S \subset X : |S| = k \wedge \forall s \in S, \forall y \in X \setminus S : \delta(q, s) \leq \delta(q, y)\}. \quad (1.1)$$

Without specialized index structures, computing exact KNN requires evaluating  $\delta(q, v)$  for every  $v \in X$ , resulting in  $O(n)$  distance computations [21]. Tree-based indexes can achieve sub-linear query time in low-dimensional spaces [29, 30], but in high-dimensional spaces, even indexed exact search becomes computationally expensive [31]. Moreover, traditional spatial indexing structures suffer from the *curse of dimensionality* [32]. In high-dimensional spaces, the ratio of the distance to the nearest neighbor versus the farthest neighbor approaches 1 as dimensionality increases [32]. This distance concentration makes traditional space partitioning methods ineffective, as most points fall within similar distances from the query, causing indexes to degenerate into linear scans [33, 31].

#### Approximate Nearest Neighbor Query

Because the notion of similarity itself is not a precise concept, exact KNN search can be viewed as unnecessarily strict, particularly given its computational complexity in high-dimensional spaces. Consequently,

research efforts often focus on approximate methods that trade small accuracy losses for substantial computational gains [33].

Formally, **approximate nearest neighbor** (ANN) search relaxes the exactness requirement by accepting any result within a factor  $(1 + \epsilon)$  of the true nearest neighbor distance [33], is a parameter controlling the allowed error.

While exact indexing methods limit the explored space, they only discard parts that provably cannot contain any of the true nearest neighbors. As a result, for a dataset of size  $n$ , in a worst-case scenario, an exact index may have to consider all  $n$  objects as potential KNN candidates and has to compare them to the given query.

In contrast, ANN search includes an approximation parameter  $n_{\text{candidates}} \ll n$  which limits the number of vectors examined during search. Based on this parameter, the index identifies a candidate set  $C \subseteq X$ ,  $|C| = n_{\text{candidates}}$ , and performs exhaustive KNN search on this set [20]. The approximation arises from the possibility that some true nearest neighbors may lie outside the candidate set. Then, the ANN query can be expressed as:

$$\text{ANN}_k(q, X, n_{\text{candidates}}) = \text{NN}_k(q, C) : C \subseteq X, |C| = n_{\text{candidates}} \quad (1.2)$$

where  $C$  is the candidate set for the query  $q$ , selected by the index.

To facilitate efficient candidate selection, an index structure is built over the dataset  $X$  during a preprocessing phase.

## 1.2 Evaluation Metrics for ANN

ANN search trades accuracy for efficiency by examining only a subset of vectors, accepting that some true neighbors may be missed. Therefore, evaluation requires measuring both search quality and computational cost. Optimizing for either metric alone is trivial: an exhaustive search guarantees perfect accuracy, while examining fewer vectors reduces cost.

### 1.2.1 Search Effectiveness

The standard metric for evaluating ANN search quality is  $\text{recall}@k$ , which measures the fraction of true nearest neighbors that appear in

the returned result [6]. To make this metric meaningful, recall must be measured as a function of the *candidate budget*  $n_{\text{candidates}}$ , which limits the number of vectors examined during search:

$$\text{Recall@}k(n_{\text{candidates}}) = \frac{|\text{NN}_k(q, X) \cap \text{ANN}_k(q, X, n_{\text{candidates}})|}{k} \quad (1.3)$$

### 1.2.2 Search Efficiency

While recall measures search quality, efficiency requires additional metrics. The number of candidates  $n_{\text{candidates}}$  needed to reach a given recall threshold provides a hardware-independent measure of computational cost, enabling fair comparison across different indexing methods [6].

However, search efficiency depends on more than candidate count alone. Different indexing methods have varying overhead costs from data structure traversal, distance computations, and memory access patterns. *Queries per second* (QPS) measures actual search throughput by timing the complete search pipeline, capturing both distance computation costs and all index-specific overhead that candidate count cannot reflect.

Indexes also exhibit different construction-versus-search tradeoffs. Some build quickly but require more candidates to achieve good recall, while others invest additional time during construction to enable faster search. Therefore, a complete evaluation considers both build time and search performance to characterize the full cost-quality tradeoff [6].

## 1.3 Traditional Indexing Methods for ANN Search

ANN search has motivated diverse indexing approaches, including *graph-based* [34], *space partitioning* [35], and *locality-sensitive hashing* [36] methods.

Locality-sensitive hashing (LSH) has been introduced in [33]. Although indexes based on this technique have been shown to work well in theory [37], in practice they are often outperformed by graph-based and partitioning-based methods [6].

Therefore, this section reviews representative methods from these two dominant paradigms, which serve as baselines for evaluating the framework proposed in this thesis.

### 1.3.1 Graph-Based Indexes

Graph-based indexes construct graphs using probabilistic heuristics to enable efficient navigation through the dataset. At each step, moving to the neighbor closest to the query brings the search toward the true nearest neighbors, which distinguishes graph indexes from arbitrary graphs. This method has been explored in HNSW [17], optimized structures like NSG [38], or disk-based variants like DiskANN [39].

In general, graph indexes achieve superior recall-latency trade-offs [6] but require significant memory overhead to store neighbor references for each point [40]. Dynamic workloads also present inherent challenges for this approach. Physical removal of an object (*hard delete*) requires identifying and removing all incoming edges, then reconnecting neighboring vertices to maintain navigability [17]. This rewiring is computationally expensive and can degrade search quality. As an alternative, many implementations only mark the deleted vertices as inactive without modifying the structure (*soft delete*), leading to memory waste over time [41].

Although insertions can be handled efficiently by adding new vertices and connecting them to existing neighbors, the underlying topology is optimized for the initial dataset. Frequent insertions cause it to become outdated, degrading search performance [41, 40].

A widely used representative of this category is HNSW [17]. It constructs a multi-layer graph where each layer contains a subset of points, with higher layers containing fewer points for coarse navigation and lower layers providing finer granularity. Search begins at the sparse top layer and traverses downward using greedy navigation until reaching the densest bottom layer. Despite being state-of-the-art for static workloads, HNSW's performance degrades in dynamic settings for the reasons described above, and most of its implementations support only soft delete [18, 17].

### 1.3.2 Partitioning-Based Indexes

Partitioning-based methods such as M-Tree [42] or disk optimized D-Index [43], divide the dataset and restrict the search to promising partitions, which are then searched exhaustively. Typically, these methods achieve lower memory usage than graph methods at the cost of reduced search efficiency [6].

These indexes consist of *partitions* containing data vectors and a navigation mechanism for selecting which partitions are searched. Partitions are represented using representative vectors called *pivots*, or through pivot permutations as in M-Index [44]. Pivots can be organized in flat structures where all partitions exist at the same level (e.g., IVFFlat [18]), or hierarchically in tree structures (e.g., M-tree [42], VP-tree [45]). During search, the distance from the query to each pivot serves as a proxy for partition relevance [20].

In general, optimal ANN partitions should be small, balanced, and *knn-consistent* [46], meaning that vectors and their nearest neighbors reside in the same partition. In practice, indexes only approximate this ideal conditions by optimizing subsets of these properties during construction. Modern approaches often use many smaller partitions, which allow for more precise routing and smaller candidate sets [47].

The most common partitioning method is *k-means* clustering [48], which iteratively assigns vectors to their nearest cluster centroid and updates centroids based on cluster membership until convergence. Although k-means is not guaranteed to find the optimal partitioning and often produces unbalanced partitions, it works well in practice and is widely used due to its simplicity and efficiency [18].

A representative of partitioning-based indexes is IVFFlat (Inverted File with Flat storage) [18]. During index construction, the dataset is clustered into  $k$  partitions using k-means clustering, and an inverted file maps each centroid to its corresponding list of vectors. The search procedure identifies  $n_{\text{probe}}$  nearest partitions to the query by computing distances to all centroids. These partitions are then searched exhaustively to find the result. This partitioning strategy achieves significant memory savings compared to graph-based methods like HNSW, storing only cluster assignments rather than neighbor graphs [18, 17].

However, IVFFlat’s effectiveness depends on parameter tuning. Too few partitions result in large buckets requiring expensive exhaustive

search, while too many partitions scatter nearest neighbors, necessitating higher  $n_{\text{probe}}$  values to maintain recall [18]. Furthermore, the partitioning and centroids remain fixed after the initial construction. When newly inserted data follows different patterns than the original dataset, vectors are assigned to suboptimal partitions, which degrades search quality [15].

## 1.4 Learned Indexing

The aforementioned limitations of existing approaches motivate the development of learned indexes, which replace distance computations with machine learning models that can learn complex data patterns, potentially leading to better routing than distance-based navigation. Additionally, many existing indexes require that the used distance function is a true metric, as defined in Section 1.1.1. Machine learning models are not limited by this assumption and can adapt to any distance function.

Kraska et al. [49] introduced the notion of learned indexing by using machine learning models as index structures for one-dimensional sorted data. This approach has since been extended to approximate nearest neighbor search, in both multi-dimensional [50] and high-dimensional vector spaces [9].

Learned indexes for ANN can be viewed as an evolution of partitioning indexes, since they also split the dataset into partitions (*buckets*) that are searched exhaustively. However, partitioning-based indexes rely on the assumption that the distance from the query to the pivot is a sufficient proxy for the bucket’s relevance to the query [20]. In contrast, learned indexes employ a machine learning model, e.g., a neural network, which takes a query vector as an input and outputs a relevance score for each bucket. This predicted relevance can be used for both organizing the vectors into buckets and for selecting buckets during search. Because the model is trained on the actual data distribution, it should be able to learn more complex relationships between the query and the buckets than the distance function, improving routing quality [8, 51].

Learned index implementations vary significantly in their (i) structure, (ii) model architecture, and (iii) training procedures.

Similar to tree-based indexes, many learned indexes are organized hierarchically. The Learned Metric Index [8] has a tree structure with a routing model at each node. BATL [12] organizes the buckets into a tree structure but uses a single routing model to predict the full paths of the tree.

The routing model can be any machine learning model, including linear regression, random forests, or Gaussian mixture models, as explored in [8, 9]. However, most works are focused on using neural networks, ranging from simple MLP [8], to attention-based architectures [12], and deep neural networks [52, 11].

Training can be based on an existing partitioning (*supervised*) [8], or the index can learn its own partitioning (*unsupervised*) as proposed in [9]. BLISS [10] uses an iterative re-partitioning approach, where the routing model is trained on the current partitioning, then the partitioning is updated based on the model’s predictions, and this process is repeated until convergence.

This thesis focuses on the Learned Metric Index (LMI), specifically the scalable version introduced in [14]. The details of its architecture and training are discussed in Chapter 2.

#### 1.4.1 Degradation Under Dynamic Updates

Before discussing the performance of learned indexes in dynamic workloads, we need to understand how updated data relates to the original dataset. *Distribution shift* [53], a phenomenon mainly studied in machine learning, describes how the relationship between inputs and outputs changes between training and testing phases. A subset of this issue also occurs in indexing. When the dataset on which an index is built differs significantly from the inserted data, performance can degrade [54]. This challenge affects both standard and learned index structures.

We identify three distinct update scenarios based on how new data relates to the partition boundaries created by the index (for the original data distribution): (i) *uniform insert*, where new data follows the same distribution as existing data; (ii) *distribution-shift insert*, where new data either does not align with existing partitions or concentrates in a single partition; and (iii) *delete shift*, where deletions create gaps in the original distribution.

The quality of a learned index depends on two factors: *partitioning quality*, based on the properties defined in Section 1.3.2, and *routing quality*, which refers to how accurately the routing model predicts bucket relevance for queries. As illustrated in Figure 1.1, all the mentioned dynamic scenarios degrade one or both factors.

Under uniform insertion, (see Figure 1.1a), routing remains correct since the data distribution is preserved. However, as buckets accumulate more objects, the search must process additional candidates to maintain the same recall.

*Out-of-distribution* (OOD) inserts seen in Figure 1.1c produce a similar issue. The OOD vectors accumulate in a single bucket, so if the search navigates into that bucket, it has to process a disproportionately large number of candidates.

Distribution shift inside the bounds of the partitioning, shown in Figure 1.1b, introduces routing error in addition to bucket size issues. The new data does not align with the partitions, meaning that buckets ranked as less relevant by the routing model may now contain objects closer to the query than those in more relevant buckets.

*Delete shift* (see Figure 1.1d) creates similar routing problems through data removal. The model continues to predict based on the original distribution, directing queries to buckets that once contained relevant neighbors but now hold only the remaining, farther data.

Index quality would stabilize only if uniform insertions and deletions occurred at the same rate, maintaining stable bucket sizes and partition boundaries. However, this idealized scenario rarely occurs in practice, which means that dynamic workloads require periodic index updates to preserve performance.

## 1. SIMILARITY SEARCH AND APPROXIMATE NEAREST NEIGHBOR INDEXING

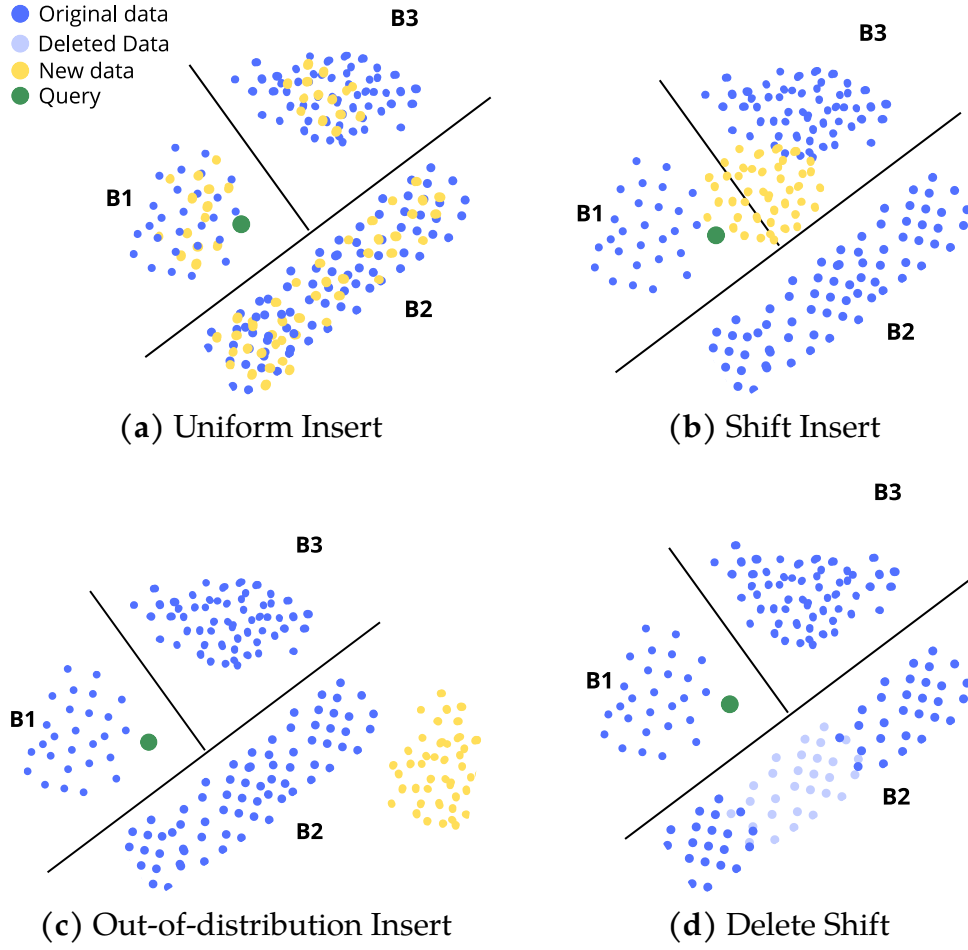


**Figure 1.1:** Effects of insertions and deletions on index partitions. Original data is shown in blue, inserts in yellow, deletions in light blue, and the query point in green. (a) When newly inserted data follows the original distribution, the existing partition boundaries remain relevant. Distribution shift occurs when new data (b) crosses partition boundaries or (c) disproportionately concentrates in a single partition. (d) Deletions create gaps in the distribution, making boundaries inaccurate.

### 1.4.2 Dynamization of Learned Indexes

The degradation described in Section 1.4.1 is caused by the index being static. In this context, the term *static index* does not refer to the index’s ability to support insert and delete operations, but to the fact that it does not adapt in any way to the changing dataset. In contrast, a *dynamic index* modifies its structure in response to data updates.

The naive approach to achieving dynamicity by rebuilding the entire index is computationally expensive. Effective methods for dynamic workloads must be able to maintain search quality and efficiency without incurring the cost of full retraining.

Since partitions and the routing model are tightly coupled, we identify three possible approaches to updating learned indexes for dynamic workloads: (i) *continual dynamicity*, which updates both the partitions and the routing model together; (ii) *structural dynamicity*, which keeps the routing model fixed while extending the partitioning; and (iii) *temporal dynamicity*, which treats the model and buckets as a single component, not updating either, unless it involves rebuilding the index as in the initial constructions.

*Continual dynamicity* updates the whole index incrementally as data evolves, potentially adding or removing buckets and extending the model architecture accordingly. Updating the model for new buckets without full retraining requires continual learning techniques [55]. This introduces challenges such as catastrophic forgetting [56], where the model gradually loses performance on the original distribution as it adapts to new data. Although there are various approaches to mitigate these issues, such as replaying old training data or modifying the loss function [56], they only delay the need for a full retraining of the model.

*Structural dynamicity* maintains a hierarchical index structure with local modifications at the leaf level. Overpopulated buckets are handled by either (i) replacing the bucket with a new sub-index containing multiple partitions, or (ii) reorganizing sibling buckets under their parent model into a new model with more partitions. Underpopulated buckets are discarded with their parent model adjusted accordingly. This strategy was first applied to one-dimensional learned indexes [57] and later adapted for metric spaces in [15], enabling the

LMI to adapt to density variations through localized restructuring rather than global retraining.

Finally, *temporal dynamicity* views the entire learned index as a static black box component within a larger dynamic structure. The *Bentley-Saxe decomposition* [16] was developed for dynamizing static solutions to decomposable searching problems. A conceptually similar approach has been successfully employed in Log-Structured Merge trees (LSM-trees) [58], which provide write-optimized storage in NoSQL databases such as Cassandra, LevelDB, and HBase [59]. This method can also be applied to learned indexes, by organizing data across multiple static indexes of geometrically increasing sizes, with new data residing in small, lightweight indexes while older data migrates to progressively larger structures. The multi-level hierarchy maintains search quality by triggering rebuilds of individual static components in response to inserts, avoiding full index reconstruction. The components are ordered by insertion time, with recent data in smaller indexes and older data in larger structures. These *temporal properties* enable efficient filtering or bulk deletion of historical records by targeting specific components.

Building on the success of temporal decomposition in key-value indexes, this thesis applies the temporal dynamicity approach to enable learned indexes in dynamic settings.

## 2 Static Learned Index Interface

The temporal decomposition approach treats static learned indexes as black-box components. The static index must therefore expose a minimal interface to be integrated into the framework we propose in Chapter 3.

The requirements can be implemented by any learned index, regardless of its specific architecture or training procedure. In this chapter, we demonstrate this by implementing the interface using the Learned Metric Index (LMI) [8], specifically the scalable version from [14]. Most notably, we show how to easily extend the LMI to support naive insert and delete operations, which are typically not supported by static indexes, but are required for our framework.

As discussed in Section 1.4.2, despite supporting these operations, the learned index remains static because it does not adapt to the updates. Section 2.6 empirically evaluates this behavior on LMI, showing how performance degrades under dynamic workloads.

### 2.1 Required Operations

To be compatible with our framework, the static learned index must support the following operations:

- **INITIALIZE**( $n_{\text{buckets}}$ ): Initializes an empty index with  $n_{\text{buckets}}$  buckets. Allocates internal data structures but does not train the routing model.
- **INSERT**( $X, I$ ): Inserts a batch of  $m$  vectors  $X \in \mathbb{R}^{d \times m}$  with corresponding identifiers  $I \in \mathbb{N}^m$  into the index. Must handle (i) initial insertion into a new index, (ii) insertion into a partially filled index, and (iii) insertion into an empty index. Details of these scenarios are discussed in Section 2.3.
- **DELETE**( $oid$ ): Removes the vector with identifier  $oid$  from the index. Returns true if successful, false otherwise.
- **PREDICT**( $q$ ): Evaluates the routing model on query vector  $q$  and returns a list of bucket-probability pairs, sorted from the highest probability.

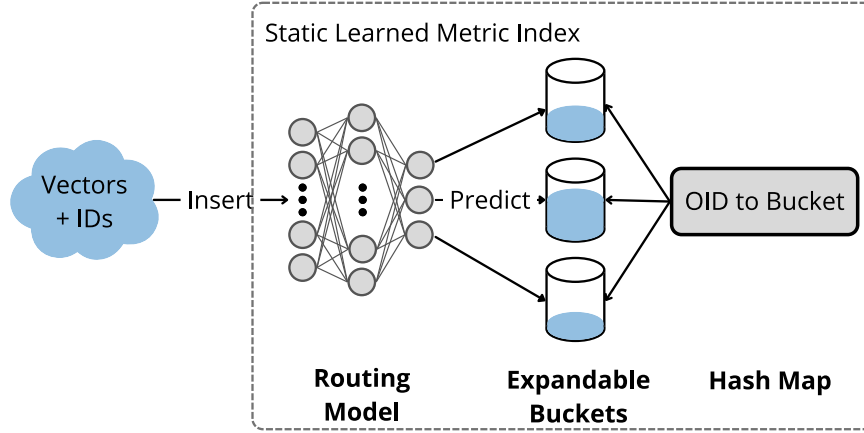


**Figure 2.1:** Static LMI consists of: (i) expandable buckets which store vectors and their ids, (ii) a neural network routing model that takes a vector as an input and predicts the relevance of each bucket to the vector, and (iii) a hash map that maps vector ids to their assigned buckets.

- **EMPTY()**: Removes all vectors and identifiers from the index while preserving the routing model and bucket structure for reuse.
- **GET\_BUCKETS()**: Returns the current bucket collection with stored vectors and identifiers.
- **GET\_N\_OBJECTS()**: Returns the number of vectors currently indexed.

The following sections provide a conceptual overview of the index structure and functionality. The detailed description of the LMI configuration and pseudocode of its non-trivial operations is included in the Appendix A.2.

## 2.2 Index Structure

As depicted in Figure 2.1, the static LMI is composed of a neural network routing model, a collection of expandable buckets for vector storage, and a hash map. Each vector in the index is associated with a

unique global identifier (OID) provided by the user during insertion, which enables point deletion and retrieval operations independent of the vector’s position within the index structure.

### 2.2.1 Expandable Buckets

An expandable bucket is a variable-size container that stores vectors in insertion order, without internal sorting or spatial organization. Unlike traditional partitioning schemes that enforce balanced bucket sizes, expandable buckets have no maximum size limit to accommodate skewed data distributions.

Each bucket supports three primary operations: (i) insertion of a vector and its OID, (ii) exhaustive search for  $k$ -nearest neighbors, and (iii) deletion of a vector by its OID.

The unstructured design allows appending new vectors in constant time (not taking reallocation costs into account) regardless of bucket size, but search and deletion scale linearly with bucket occupancy.

### 2.2.2 Routing Model

The routing model is a fully connected feedforward neural network that maps  $d$ -dimensional input vectors to bucket assignments. The architecture consists of an input layer with  $d$  neurons (matching the dataset dimensionality), a single hidden layer with 128 neurons using ReLU activation, and an output layer with  $b$  neurons (one per bucket) using softmax activation.

Given an input vector  $x \in \mathbb{R}^d$ , the network produces a probability distribution  $P$  over buckets. The predicted relevance of bucket  $i$  for  $x$  is given by:

$$P(\text{bucket}_i \mid x) = \text{softmax}(h(x))_i \quad (2.1)$$

where  $h : \mathbb{R}^d \rightarrow \mathbb{R}^b$  represents the routing model, and  $h(x)$  is the output of the model given input  $x$ .

The purpose of the model is to guide the insert and search operations by predicting the probability (relevance) of each bucket for a given input vector. The mechanism of this routing for insert is described in Section 2.3 and for search in Section 2.5

### 2.2.3 OID to Bucket Mapping

The index maintains an auxiliary hash map that maps each vector OID to its assigned bucket. This structure provides expected  $O(1)$  lookup time to determine whether a given OID exists in the index and, if so, which bucket contains it. Without this hash map, deleting a vector by OID would require scanning all buckets, resulting in  $O(n)$  complexity where  $n$  is the total number of indexed vectors.

## 2.3 Insert

The INSERT operation handles three scenarios: (i) *initial insertion* into a newly initialized index, (ii) *naive insertion* into a partially filled index with an existing model, and (iii) *reinsertion* into an emptied index.

This overloading is a design choice that simplifies the interface while allowing implementation flexibility. By having this logic within the interface, we keep the framework general while allowing the naive insert and reinsert to be tailored to the specific underlying index. For example, some static indexes may already support incremental retraining or more sophisticated naive insertions. Rather than requiring the framework to be aware of such variations, the interface allows each index to leverage its own capabilities.

Without making assumptions about the underlying index beyond the standard capabilities of a static learned index, the three scenarios can be implemented as follows. *Initial insertion* builds the index from the batch of data as it would in a static scenario. *Naive insert* uses the same mechanism as search. The routing model predicts bucket relevance for the incoming vector and inserts it into the most relevant bucket. For *reinsertion*, the existing model is removed, and the initial insertion procedure is reused.

Figure 2.2 illustrates the control flow of the insert operation for LMI. The reinsertion variant updates the training configuration and then performs the same steps as initial insertion, but with the new configuration. We refer to this process as *retraining* and discuss it in depth in Section 3.3.2. The following sections describe the LMI training procedure and vector organization for the initial insert. The pseudocode encompassing all three insert scenarios is included in Algorithm 4.

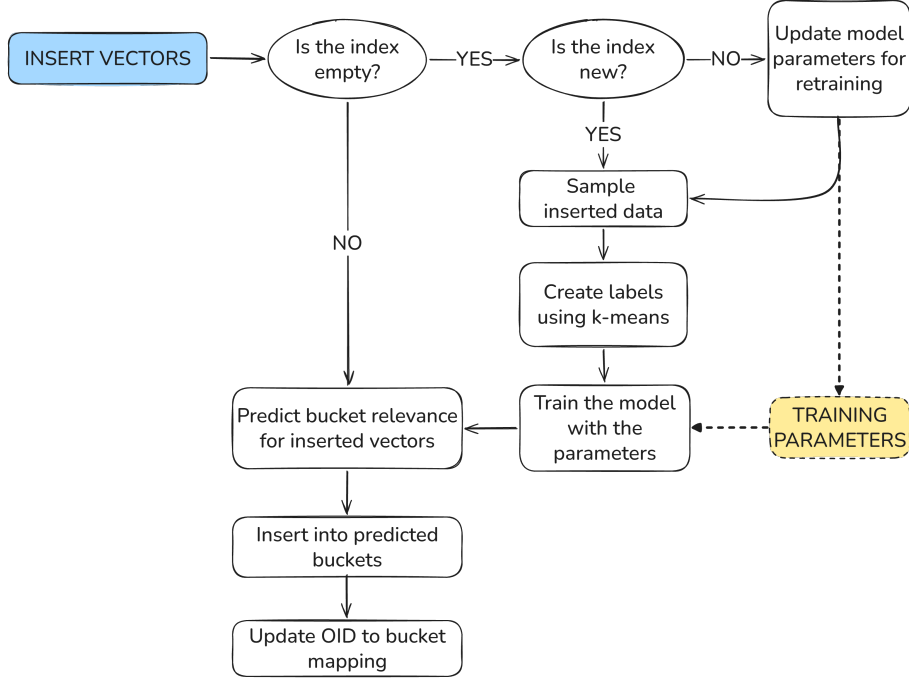


**Figure 2.2:** Control Flow of the static LMI INSERT operation. The procedure varies based on the index state. For a new index, initial insertion trains the model and organizes vectors into buckets. For naive insertion into a partially filled index, training is skipped, and vectors are routed into the most relevant bucket. For an empty index, the training configuration is updated, and the initial insertion procedure is reused.

Since INSERT involves training, its input has to be a reasonably sized batch of vectors. This is ensured by the framework, as discussed in Section 3.3.

### 2.3.1 Training Phase

Training consists of sampling, k-means clustering, optional label balancing and validation splitting, and supervised learning of the neural network. All training parameters are configurable and specified in Table A.3, and the full procedure is depicted in Algorithm 5.

### Sampling

Training on the entire inserted dataset can be computationally expensive. Therefore, a subset of size  $n_{\text{sample}}$  is sampled uniformly at random, without replacement, from the incoming vectors. The sample size is constrained by:

$$40 \cdot n_{\text{buckets}} \leq n_{\text{sample}} \leq m \quad (2.2)$$

Where  $m$  is the total number of vectors in the insertion batch and  $n_{\text{buckets}}$  is the number of buckets in the index. The lower bound ensures approximately 40 samples per cluster for reliable k-means convergence [60]. When early stopping [61] is enabled in the training configuration, the sample size is increased by 20% to accommodate a train-validation split without reducing the training set size.

### K-means Clustering

The sampled subset is partitioned into  $k = n_{\text{buckets}}$  clusters using the Faiss k-means implementation [18]. Each vector  $x_j$  in the sample is assigned to a cluster  $y_j \in \{1, \dots, n_{\text{buckets}}\}$ . These assignments serve as the labels for the neural network training.

### Label Balancing and Validation Split

K-means does not guarantee balanced cluster sizes, potentially producing skewed cluster sizes. Training on imbalanced data can bias the neural network toward predicting frequent classes, degrading routing quality for underrepresented regions.

To address this, the training set is resampled using PyTorch’s WeightedRandomSampler [62], so the sampling probabilities are inversely proportional to cluster frequencies. This ensures each cluster contributes equally to gradient updates during training.

If early stopping is enabled, the balanced dataset is split into training and validation subsets using an 80/20 ratio. Both balancing and validation splitting are optional and controlled by the training configuration (see Table A.3).

## Model Training

The neural network is trained via supervised learning to predict k-means cluster assignments. Training follows a standard supervised process consisting of forward passes, loss computation, backpropagation, and weight updates. The parameters and pseudocode for the training loop are provided in Algorithm 5.

### 2.3.2 Vector Organization

After training, all vectors in the inserted batch are organized into buckets. Each vector  $x$  is assigned to the bucket with the highest predicted probability:

$$\text{bucketAssignment}(x) = \operatorname{argmax}_{i \in \{1, \dots, n_{\text{buckets}}\}} P(\text{bucket}_i \mid x). \quad (2.3)$$

The vector and its identifier are appended to the selected bucket, and the hash map is updated accordingly.

## 2.4 Delete

The `DELETE` operation removes a vector from the index given its OID. The hash map provides an expected  $O(1)$  lookup to identify which bucket contains the vector. If the identifier is not found, the operation returns `false`. Otherwise, the bucket is scanned linearly to locate and remove the vector, and the corresponding hash map entry is deleted. The operation returns `true` upon successful removal.

The deletion physically removes the vector from the bucket, shrinking the bucket's storage. This hard delete approach avoids the data accumulation and query degradation associated with soft deletes.

The overall time complexity is  $O(n_b)$ , where  $n_b$  is the size of the bucket containing the deleted vector. For the full procedure, see Algorithm 1.

## 2.5 Search

While the required interface only includes the `PREDICT` operation, we describe the complete `SEARCH` procedure as it is essential for both

evaluating index quality and designing the dynamic framework’s search strategy.

Given the query vector  $q$ , the index first evaluates the routing model to obtain a probability distribution over buckets (see Algorithm 2), then sorts buckets in decreasing order of predicted probability. Buckets are visited in this order until a stopping criterion is met. For each visited bucket, all its vectors are compared against the query using the distance function, and the top  $k$  candidates from each bucket are identified.

The SEARCH operation receives a candidate budget  $n_{\text{candidates}}$ . However, once a bucket is selected, all its vectors must be examined, so the actual number of evaluated vectors may exceed this budget. The search visits buckets  $B_1, B_2, \dots$  until their cumulative size meets (or exceeds)  $n_{\text{candidates}}$  at bucket  $B_j$ , meaning the actual number of processed candidates is  $\sum_{i=1}^j |B_i|$ .

After visiting the selected buckets, the top  $k$  candidates from all buckets are merged, and the  $k$  vectors with the smallest distances to  $q$  are returned as the approximate nearest neighbors. Algorithm 3 captures the details of this operation.

## 2.6 Performance Evaluation

Learned indexes are not designed to handle dynamic updates. Even though this chapter describes the implementation of naive insert and delete operations for the static LMI, these operations lack mechanisms to maintain search quality after the updates. Therefore, after a series of updates, the search quality will inevitably degrade.

We conducted preliminary experiments evaluating search quality under four different insertion scenarios. In the *Static Baseline* scenario, LMI was trained on the entire dataset. The three insert scenarios trained LMI on different initial subsets: *Uniform Insert* used a random sample, *Slight Shift Insert* used three distinct clusters, and *Extreme Shift Insert* used only one cluster. In all scenarios, the remaining vectors were inserted using naive insert. The detailed experimental setup is explained in Section 4.2.

Figure 2.3 shows recall curves for all of the aforementioned scenarios. Both distribution shift scenarios show significant recall degrada-

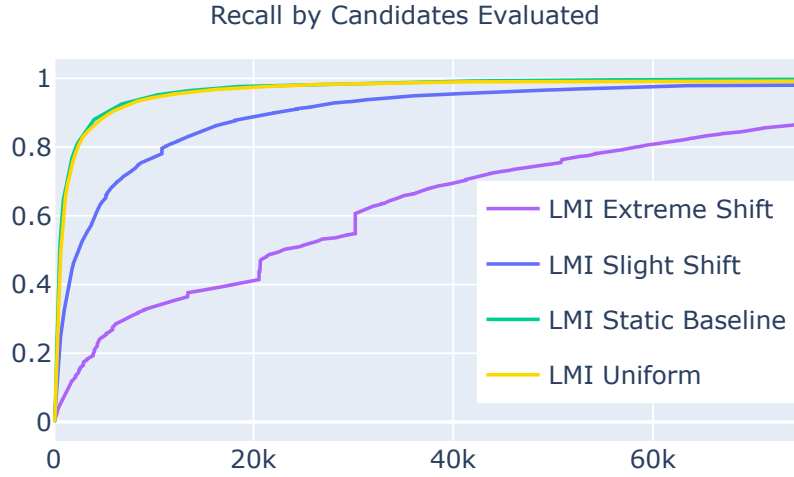


**Figure 2.3:** Recall of LMI with different subsets of dataset used for training. Static Baseline uses the whole dataset, Uniform uses a random sample. Slight Shift uses three distinct clusters, and Extreme Shift only one cluster. The plot shows recall as a function of candidates evaluated (objects examined during search) on the LAION2B-300K dataset with  $k = 30$ . Uniform Insert exhibits minimal degradation, performing nearly identically to the Static Baseline. Distribution shift scenarios show progressively worse performance, with Extreme Shift Insert having the most significant recall drop.

tion. The *Uniform Insert* shows barely any difference from the baseline, which is because in both cases, the size of the dataset is the same, so the bucket size issue described in Section 1.4.1 is not present. However, in the next section, we show that even this scenario leads to degradation of some aspects of the index.

### 2.6.1 Understanding Performance Degradation

Although LMI’s recall and efficiency have been reported extensively, the individual factors contributing to search quality were not analyzed. Understanding the root causes of recall degradation is essential for maintaining search quality in dynamic settings.

For learned indexes (and other partition-based indexes), data is organized into buckets. During search, a routing model orders buckets by predicted relevance. The candidate budget determines how many buckets to explore, with each selected bucket searched exhaustively by comparing all its vectors to the query.

Consider an index with 100 buckets of 1000 vectors each. If the ground truth neighbors for a query are scattered across five buckets and  $n_{\text{candidates}} = 3000$ , the search evaluates only three buckets, never reaching perfect recall regardless of routing quality. Conversely, if all ground truth neighbors reside in a single bucket but the routing model ranks it poorly, e.g., as the last bucket to be visited, the search also cannot find the true neighbors.

These examples showcase that low recall can stem from either poor selection of buckets (*routing failure*) or suboptimal data distributions (*partitioning failure*). To address this, we propose the following metrics: (i) *Ground Truth Buckets* and (ii) *Ground Truth Bucket Candidates* that tackle partitioning quality, and (iii) *Wasted Effort*, which targets routing quality.

These metrics serve as diagnostic tools for understanding the behavior of the index and guiding the design of the dynamic framework.

### Ground Truth Buckets

The number of buckets containing ground truth neighbors (GT) indicates how these neighbors are distributed across partitions for a given query. When ground truth neighbors scatter across many buckets, achieving high recall requires exploring more partitions (and processing more candidates), even if the model was routing optimally. For a given query  $q$ , we define the Ground Truth Bucket Count (GT Buckets) as:

$$\text{GT\_Buckets} = |\{B_i : i \in \{1, \dots, n_{\text{buckets}}\} \wedge B_i \cap \text{GT}(q) \neq \emptyset\}|, \quad (2.4)$$

where  $\text{GT}(q)$  is the set of ground truth neighbors for  $q$ , and  $B_i$  is the set of vectors in  $i^{\text{th}}$  bucket. This metric captures the minimal number

of buckets that must be explored to find all ground truth neighbors for a given query, providing insight into partitioning quality.<sup>1</sup>

Additionally, we observe the size distribution of ground-truth-containing buckets. A configuration where one disproportionately large bucket contains most ground truth neighbors can achieve high recall if routing succeeds, but imposes a high computational cost due to exhaustive within-bucket search. For this reason, we define the Ground Truth Candidates metric (GT Candidates), which counts the number of candidates in the buckets containing GT neighbors:

$$\text{GT\_Candidates} = \sum_{\substack{i \in \{1, \dots, n_{\text{buckets}}\} \\ \wedge B_i \cap \text{GT}(q) \neq \emptyset}} |B_i| \quad (2.5)$$

Figure 2.4 shows the partitioning diagnostics for LMI insertion scenarios. The GT Buckets distribution, shown in Figure 2.4a, reveals only slight differences, with Uniform Insert appearing nearly identical to the Static Baseline. However, the GT Candidates distribution in Figure 2.4b shows that even Uniform Insert requires processing more candidates to find GT neighbors. For the distribution shift scenarios, this effect is amplified. The Extreme Shift scenario has queries that would require processing nearly 10% of the dataset to find all GT neighbors.

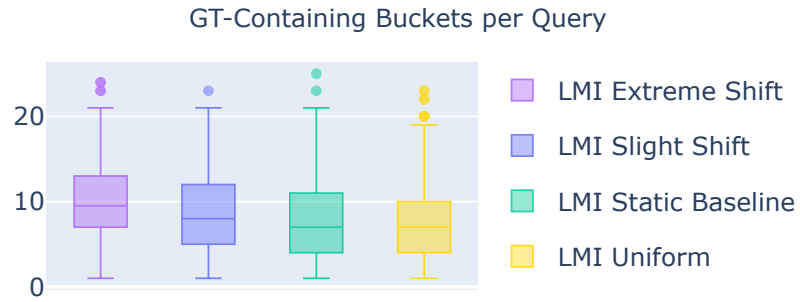
### Wasted Effort

Given a fixed partitioning, we need to quantify how well the routing model directs queries to relevant buckets. For this, we propose the Wasted Effort metric, which measures how much more work is required to reach a given recall level compared to ideal routing. It is defined as the ratio of candidates actually processed versus the theoretical minimum with perfect routing (buckets sorted by GT neighbor count):

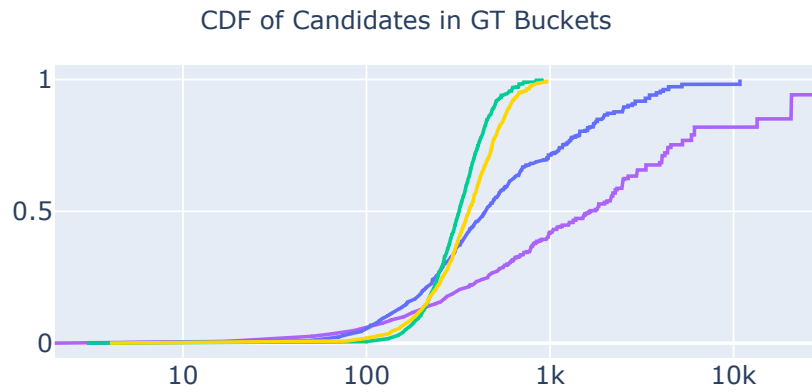
$$\text{WE@}r = \frac{n_{\text{candidates}}(r, \text{actual})}{n_{\text{candidates}}(r, \text{optimal})} - 1, \quad (2.6)$$

---

1. To compare indexes with different bucket counts, we can report the percentage of buckets containing ground truth neighbors rather than the absolute count.



(a) LMI GT Buckets



(b) LMI GT Candidates

**Figure 2.4:** LMI partitioning quality in different distribution shift scenarios. The GT Buckets remain relatively stable. GT Candidates degrade slightly in the Uniform Insert scenario, but show significant degradation with more aggressive distribution shifts.

where  $r$  is the target recall level,  $n_{\text{candidates}}(r, \text{actual})$  is the number of candidates the routing model requires to achieve recall  $r$ , and  $n_{\text{candidates}}(r, \text{optimal})$  is the minimum with optimal bucket ordering.

The metric has a lower bound of 0 (perfect routing), and an upper bound determined by the dataset. A value of  $\text{WE}@r = 0$  indicates optimal routing for that recall level, while higher values indicate routing inefficiency. For example, if achieving 0.95 recall requires processing 5000 candidates but perfect routing would achieve it with 2000, then  $\text{WE}@95 = 1.5$ , meaning the search had to process 1.5 times more candidates than necessary, due to a routing failure.

Figure 2.5 shows Wasted Effort for the LMI insertion scenarios. The Uniform Insert scenario curve remains close to the Static Baseline, indicating that the previously observed recall differences stem primarily from partitioning quality. For both Slight and Extreme shifts, the curves spike at low recall levels, then decrease toward 0 but stop around 1, suggesting the model identifies relevant buckets but explores many irrelevant ones first. Their final values of  $\text{WE}@95} \approx 1$  also show the model processed twice as many candidates as necessary, because it visited irrelevant buckets in addition to incorrectly ordering the relevant ones.

The naive solution to performance degradation in learned indexes is periodic rebuilding [15]. However, rebuilding the index requires retraining the routing model and reorganizing all vectors, imposing substantial computational overhead that grows with dataset size. For applications with continuous updates, frequent rebuilds become prohibitively expensive. In the next chapter, we propose a temporal decomposition framework that maintains search quality under dynamic workloads without requiring full index reconstruction.

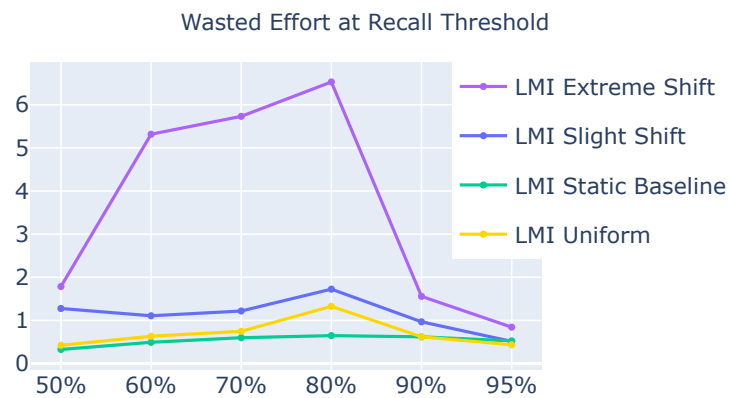


**Figure 2.5:** Wasted Effort as a function of recall threshold for different insertion scenarios. Static Baseline and Uniform Insert maintain low wasted effort ( $WE < 1$ ) across all recall levels, indicating efficient routing. Distribution shift scenarios exhibit higher wasted effort that peaks at mid-range recall thresholds. At high recall (95%), stabilize around 1, indicating that while the routing model eventually explores the correct buckets, it also directs search toward many irrelevant ones.

### 3 Dynamic Learned Index

Maintaining a learned index under continuous updates requires that retraining occurs frequently enough to prevent degradation, but full index reconstruction scales with dataset size and becomes more expensive as data accumulates. We propose a solution to this problem through a dynamic framework based on temporal decomposition.

The key insight driving the decomposition approach is that building a small new index for a batch of incoming data is significantly cheaper than merging the batch with the existing dataset and retraining on the combined data. Moreover, a new index handles distribution shifts effectively, whereas inserting into an existing index would lead to degradation observed in Section 2.6.

As illustrated in Figure 3.1, the framework organizes data across multiple static indexes of geometrically increasing sizes. New data enters small indexes, and when they fill, their contents are merged into progressively larger ones. The largest and most expensive indexes are rebuilt least frequently, amortizing reconstruction costs across updates.

The standard Bentley-Saxe decomposition [16] rebuilds a static index whenever data is merged into it. From the insights in Section 2.6, we know that a learned index can absorb a certain amount of insertions if it does not cause a significant distribution shift. Based on this, our framework allows partial inserts into the static indexes, further delaying rebuild costs.

As mentioned, the proposed framework is general and applicable to any index implementing the interface from Chapter 2. We demonstrate the framework by instantiating it with LMI, creating the Dynamic Learned Index (DLI). The full summary of DLI’s configurable parameters is provided in Table A.1. The rest of this chapter describes the DLI architecture and functionality, while Chapter 4 evaluates the various design choices and performance of DLI compared to other indexes.

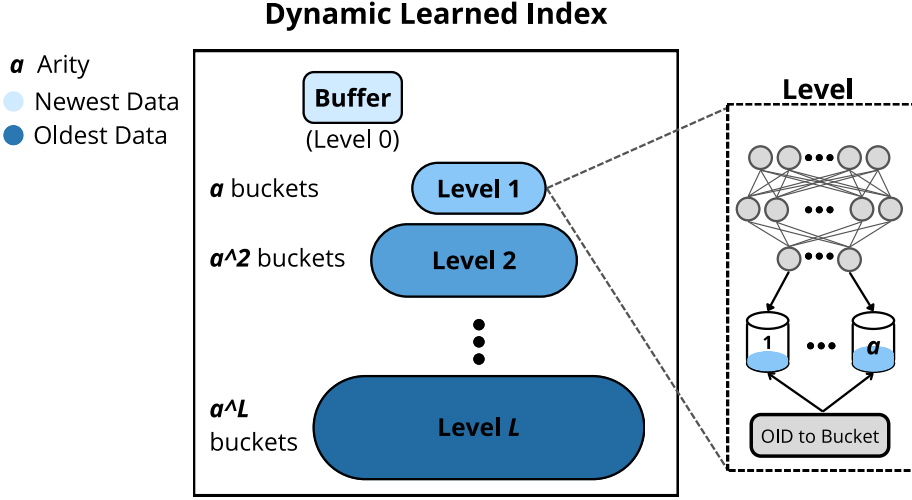


**Figure 3.1:** Dynamic Learned Index structure showing the buffer and multiple levels with geometrically increasing sizes. Each level contains  $a^i$  buckets, where  $a$  is the arity and  $i$  is the level's index. The buffer (level 0) holds the newest data while lower levels contain progressively older data. The detail view (right) shows a level's internal structure with routing model, buckets, and OID to bucket mapping.

### 3.1 Components

As shown in Figure 3.1, DLI consists of multiple levels and a buffer. We identify the levels using index  $i \in \{1, \dots, L\}$ , where  $L$  is the total number of levels.

The number of levels grows dynamically with dataset size and has an upper bound of  $L \leq \log_a(n)$ , where  $n$  is the total number of indexed vectors and  $a$  is the arity determined by the index configuration.

Level 1 is referred to as the highest level, while levels with larger indices are called lower, as visualized in Figure 3.1.

The buffer is referred to as level 0, though it lacks the predict operation since it contains no routing model. This consistent numbering simplifies the description of cross-level operations.

### 3.1.1 Buffer

The buffer is a storage unit for inserted vectors and their identifiers (OIDs). It provides a staging area for new data before integration into the levels, which require larger batches for training their routing models. This design allows the DLI to handle single-vector inserts efficiently without disrupting the underlying multi-level architecture.

The buffer has the same interface as the expandable buckets described in Section 2.2.1, but with a fixed capacity defined by the configuration parameter *buffer size*  $b$ .

### 3.1.2 Levels

Each level is an instance of LMI, satisfying the interface from Section 2.1. The number of buckets on a level  $n_b(i)$  is determined by the *arity*  $a$  and the level index  $i$ :

$$n_b(i) = a^i. \quad (3.1)$$

This exponential growth ensures that upper levels, which are frequently updated, remain small and efficient to manage, while lower levels accommodate larger volumes of data with less frequent updates.

Each level has a maximum capacity derived from the buffer size and arity:

$$C_{\max}(i) = b \cdot a^i = b \cdot n_b(i) \quad (3.2)$$

and a minimum capacity defined by the size of the previous level:

$$C_{\min}(i) = b \cdot a^{i-1} = C_{\max}(i-1). \quad (3.3)$$

These constraints are enforced during dynamic updates as described in Sections 3.3 and 3.4.

## 3.2 Functionality

DLI offers the following operations:

- **INITIALIZE(*config*)**: Initializes an empty index with the specified configuration parameters.

- **INSERT**( $x, oid$ ): Inserts a single vector  $x \in \mathbb{R}^d$  with identifier  $oid$  into the buffer. Buffer overflow is handled internally without user intervention by a process called *compaction*, described in Section 3.3.1.
- **DELETE**( $oid$ ): Removes the vector with identifier  $oid$  from the index. If the vector cannot be located in the index, the operation returns false as explained in Section 2.4.
- **SEARCH**( $q, k, n_{\text{candidates}}$ ): Returns the  $k$  approximate nearest neighbors of query vector  $q \in \mathbb{R}^d$  constrained by candidate budget  $n_{\text{candidates}}$ . The search procedure is described in Section 2.5.

The full list of configurable parameters for DLI is in Table A.1.

### 3.3 Insert

The external interface for **INSERT** is designed to be simple: users insert individual vectors without managing batch sizes or compaction triggers. Internally, DLI accumulates vectors in the buffer and triggers compaction when the buffer reaches capacity. This process is depicted in Algorithm 6.

Most inserts are  $O(1)$  buffer appends. Every  $b$ -th insert triggers a compaction procedure whose cost depends on the *compaction strategy* and the current index size.

#### 3.3.1 Compaction

When the buffer reaches its capacity, its data is moved into the levels through compaction. Data always moves downward through the hierarchy, maintaining temporal organization (older data on the lower levels, newer data on the upper ones). The exponential growth of levels ensures that upper levels, which are updated frequently, remain small and efficient. This design allows the index to adapt to distribution shifts because the upper levels can quickly learn to route new data, while lower levels provide stability because they are trained on a broader representation of the embedding space.

At initialization, the index contains only the empty buffer. When the buffer fills to its capacity  $b$  for the first time, level 1 is created,

and the data is moved into it, freeing the buffer. Since  $C_{\max}(1) = b \cdot a$  and each buffer batch contains  $b$  vectors, level 1 can accommodate the first  $a$  buffer batches before reaching its capacity, requiring further compaction.

In subsequent compactions with multiple existing levels, a compaction strategy determines how data moves downward through the structure. We propose two strategies: (i) *Leveling*, which moves data one level at a time, and (ii) *Bentley-Saxe*, which merges data from several levels in a single operation.

### Leveling Compaction

The Leveling strategy [63] identifies the highest level  $i$  with sufficient free capacity to accommodate data from level  $i - 1$ . If no such level exists, a new level is created. Data then moves downward in a sequence: first, level  $i - 1$  inserts its data into level  $i$ ; then level  $i - 2$  inserts into the emptied level  $i - 1$ ; continuing upward through the levels until level 1 inserts into level 2, and finally the buffer is emptied into level 1. Each insertion uses the LMI insert operation. A more precise depiction of the process can be found in Algorithm 8.

### Bentley-Saxe Compaction

The Bentley-Saxe strategy [16] identifies the highest level  $i$  with sufficient free capacity to accommodate all objects in levels  $0, 1, \dots, i - 1$ . If no such level exists, a new level is created. All data from levels 0 through  $i - 1$  is then merged and inserted into level  $i$  in a single insert operation. The procedure is shown in Algorithm 7.

#### 3.3.2 Level Retraining

The LMI's default behavior for insertion into an emptied index is to discard the existing model and retrain from scratch on the new data, as discussed in Section 2.3. This ensures the model is trained on the current data distribution. However, as shown in Section 2.6, when new data follows a similar distribution, the existing routing model performs well. We designed several *retraining strategies* that reuse the existing model to varying degrees, potentially reducing training time while

maintaining search quality. Specifically, we propose: (i) *No Retrain*, which skips training entirely, (ii) *Retrain From Scratch*, the LMI default behavior, (iii) *Full Retrain*, which preserves model weights, and (iv) *Partial Retrain*, which preserves model weights and also modifies the training configuration.

The effectiveness of these strategies depends on the similarity between the old and new data distributions, and their performance is evaluated in Section 4.5.1.

### **No Retrain**

Skips model training entirely, reusing the existing model weights. This minimizes training time but, without a reliable way to detect distribution shift, risks severe search quality degradation if the new data distribution differs significantly from the original.

### **Retrain From Scratch**

Discards the previous model and re-trains on the new data as if it were a newly initialized index. This is the baseline LMI behavior since it is most reliable but also most computationally expensive.

### **Full Retrain**

Full retraining uses the same training configuration as initial training but does not reset the model weights. This approach may be most effective in combination with early stopping. If the new data distribution is similar to the one used for the initial training, the model is already near convergence, and training terminates quickly. If the distribution differs, the model requires more training epochs to adapt. However, for significantly different distributions, this strategy may degrade search quality compared to retraining from scratch, as the model must unlearn the old distribution before learning the new one.

### **Partial Retrain**

Partial retraining preserves previous model weights but modifies the training configuration by reducing the number of epochs or adjusting the learning rate to encourage adaptation to the new distribution

rather than fine-tuning. Specific parameter modifications are detailed in Section 4.5.1.

### 3.4 Delete

The `DELETE` operation performs a hard delete, completely removing the vector from the index rather than marking it as deleted. Starting from the largest level, DLI attempts to execute the LMI delete operation on each level. If the delete succeeds on a particular level, the process terminates. Otherwise, it continues to the next level until the object is found and deleted or all levels are exhausted. The pseudocode for this procedure is included in Algorithm 9.

Searching from the largest levels first is an optimization based on the assumption that recently inserted objects (which reside in small upper levels) are less likely to be deleted than older objects. Even if the deletes are random, large levels have a bigger probability of containing the vector we want to delete.

The expected time complexity is  $O(L + b)$ :  $O(L)$  for hash map lookups (excluding worst-case scenarios) across levels to locate the vector, and  $O(b)$  for the linear scan within the bucket to remove it.

#### 3.4.1 Level Underutilization

Frequent deletes can cause levels to become underutilized, containing fewer than  $C_{\min}(i)$  vectors. This creates two problems: unreliable routing and insufficient retraining.

First, the routing model becomes unreliable because it was trained on data that no longer represents the actual distribution in the level.

Second, underutilization breaks the assumption that enables partial inserts. Partial inserts work because new data does not overwhelm existing data in the level, allowing the old model to handle minor distribution shifts without retraining. However, in an underutilized level, the incoming data is no longer small relative to the sparse existing data. The new data dominates the level, rendering the old model stale.

To prevent this, after each delete, the index checks for *underutilization* in the level from which the object was deleted. We consider a level  $i$  to be underutilized if the number of objects it contains is less than its

minimum capacity, as defined in Section 3.1.2. If underutilization is detected, we trigger a *rebuild* procedure, defined in Algorithm 10.

### 3.4.2 Rebuild

When a level becomes underutilized, we merge its data with data from one or more upper levels and insert them into a chosen *target level*. If the target level is empty, this also triggers retraining. The process preserves temporal ordering: data from level  $i$  can only move to level  $j > i$  if all intermediate levels are empty or are also being merged into the target.

#### Target Level Selection

Given an underutilized level  $i$ , the target level could be level  $i$  itself, some lower level  $j > i$ , or the immediate upper level  $i - 1$ , depending on the occupancy of surrounding levels.

The target selection aims to create a smaller number of larger, fuller levels rather than many sparsely populated levels, since larger levels are more stable and provide better search performance. The target level is selected in order of preference (once a condition is satisfied, remaining options are not considered):

1. The closest non-empty lower level  $j > i$ , if it has free capacity exceeding the number of objects in level  $i$ .
2. Level  $i$  itself, if there exists a non-empty upper level  $j < i$ .
3. Level  $i - 1$ . From the previous condition, we know that level  $i - 1$  is empty, and combined with the underutilization condition, this implies that  $i - 1$  has sufficient capacity for all objects in level  $i$ .

The selection process is precisely described in Algorithm 11.

Which levels are merged into the target level depends on the rebuild strategy. We designed two strategies: (i) *Basic* and (ii) *Greedy*. For experimental comparison, we also include (iii) *No Rebuild*, a baseline that skips rebuilding entirely.

**Basic Rebuild Strategy**

The basic strategy merges data from exactly two levels: the target level and one source level, where one of them is the underutilized level  $i$ . This minimizes the immediate cost of rebuilding. If the target level is not  $i$ , then level  $i$  is the source. Otherwise, the source is the nearest non-empty upper level.

**Greedy Rebuild Strategy**

The greedy strategy consolidates multiple upper levels into the target level in a single rebuild operation. Starting from the target level's free capacity, the heuristic moves upward through the hierarchy, accumulating all levels that fit within the remaining capacity. This preemptively consolidates data to prevent future frequent rebuilds and improves search performance by creating larger, more stable levels, but incurs a higher immediate cost due to increased data movement.

**3.5 Search**

The SEARCH operation proceeds in three stages. First, the PREDICT operation is performed on each non-empty level to obtain probability distributions over all buckets in the DLI. Then, buckets from all levels are globally ordered using a *search strategy*, and the top buckets that meet or exceed the candidate budget are selected. Finally, the selected buckets are visited by performing an exhaustive KNN search within each bucket, and the results are merged into the global result set returned to the user. The procedure is captured in Algorithm 12.

**3.5.1 Search strategies**

In the static LMI, the probability distribution  $P$ , created by the routing model, is used to order buckets. In DLI, we must handle multiple probability distributions from models on different levels, plus the buffer, which has no model and thus no assigned probability distribution. The search strategy defines how to order buckets across levels and incorporate the buffer into the ordering.

Two factors complicate cross-level bucket ordering. First, models on different levels have exponentially increasing numbers of output neurons. Although the softmax function bounds probabilities to  $[0, 1]$ , absolute values are not comparable across levels. A 0.5 probability on a level with 2 buckets is less significant than 0.5 on a level with 100 buckets. Second, for uniform data and query distributions, lower levels have a higher probability of containing true nearest neighbors (because they contain more data), but also a higher probability of incorrect bucket assignments due to the large number of outputs.

We propose the following strategies for cross-level bucket ordering: (i) *KNN Driven*, which moves through the levels in a fixed order; (ii) *Model Driven*, which orders buckets directly by their model probabilities; and (iii) *Model Driven Normalized*, which normalizes probabilities to account for level differences.

### **KNN Driven Strategy**

The KNN strategy (KNN\_SS) organizes buckets on each level by their model probabilities, then constructs a conceptual tree structure for traversal. The buffer serves as the root. Buckets on each level are partitioned into groups of  $a$  buckets, with each group assigned as “children” of a bucket on the previous level. This creates a tree where each non-leaf node has  $a$  children.

Traversal follows a depth-first approach. Starting from the buffer, descend to the highest-probability bucket on level 1, then to its highest-probability child on level 2, and so on to the deepest level. After visiting all buckets at the deepest level in the current branch, backtrack and explore the next highest-probability branch. This interleaves levels while respecting probability ordering within each level, without explicit cross-level probability comparisons.

The precise description of the process is included in Algorithm 13.

### **Model Driven Strategy**

This Model-Driven strategy (MD\_SS) orders buckets directly by their assigned probabilities, without any form of normalization. It is simple and efficient but suffers from the cross-level comparability issues described in Section 3.5.1, leading to suboptimal search quality.

**Model Driven Normalized Strategy**

The Model Driven Normalized strategy (MDN\_SS) addresses cross-level comparability through normalization. Consider a model on level  $i$  that is equally confident in all its buckets. For all buckets  $b_{i,k}, k \in 1 \dots n_b(i)$ , the softmax probability  $P(b_{i,k}|q) \approx 1/a^i$ . If the same holds for a model on a lower level  $j$  where  $j > i$ , then the softmax probabilities are  $P(b_{j,k}|q) \approx 1/a^j < 1/a^i$ . Without normalization, this biases search toward upper levels (which have higher average probabilities per bucket), even when models are equally confident. Worse, under complete uncertainty, lower levels should be searched preferentially since they contain more data and thus a higher probability of containing true neighbors.

The normalized strategy orders buckets by a score:

$$\text{score}(b_{i,k}) = P(b_{i,k}|q) \cdot a^i. \quad (3.4)$$

With this normalization, if a model is equally confident in all its buckets,  $\text{score}(b_{i,k}) \approx 1$  regardless of level.

The buffer is treated as level 0 with a normalized score

$$\text{score}(\text{buffer}) = 1.0. \quad (3.5)$$

## 4 Experiments

The experimental evaluation is divided into three parts: validation of the decomposition approach, evaluation of DLI parameters, and comparison of DLI performance across multiple datasets and against other indexes.

We begin by comparing LMI degradation under dynamic updates (especially with distribution shifts) to the degradation caused by distributing the indexed dataset across multiple levels. This confirms that the multilevel structure does not introduce unreasonable routing errors and is therefore a viable approach for the dynamization of learned indexes.

After validating the decomposition approach, we explore the DLI design space by evaluating all the attributes in Table A.1 to identify the optimal DLI configuration. To avoid a full grid search over the parameters, we split this evaluation into three rounds, each targeting a specific subset of attributes: (i) **structural experiments**: we fix the search strategy, compaction strategy, arity, and buffer size, using these values in subsequent rounds; (ii) **delete experiments**: we evaluate rebuild strategies under workloads with deletions; (iii) **training experiments**: we evaluate training settings and retrain strategies under distribution shifts.

Once we establish the optimal DLI configuration, we test it across multiple datasets to validate that findings generalize beyond the initial experimental dataset. Finally, we compare DLI to other indexes under dynamic workloads and distribution shifts to assess its advantages and limitations.

Before describing the experiments, we detail the experimental design and datasets used in the evaluation.

### 4.1 Experiment Methodology

Because this thesis focuses on dynamic indexes, each experiment simulates continuous updates through repeated cycles of insertions, deletions, and search evaluation.

An experiment consists of multiple *batches*. In each batch, a specified number of insert and delete operations are executed, followed

by search evaluation. The operations are performed sequentially (e.g., inserting one vector at a time) to simulate a realistic usage rather than bulk operations.

In most of the experiments, we use 20 batches of uniform sizes determined by the full dataset size. In experiments with deletes, an *initial batch* (10% of the dataset) populates the index before the insert-delete cycle begins.

Each experiment scenario is defined by (i) *workload*, (ii) *delete method*, (iii) *data distribution*, and (iv) *query distribution*. The following sections describe the purpose and available options for each of these parameters.

### Workload

The workload parameter defines the ratio of inserts to deletes in each batch:

1. **Only-insert (OI):** For scenarios without deletes (1:0 ratio).
2. **Heavy-insert (HI):** For each  $x$  insertions, delete  $x/10$  vectors (10:1 ratio, index grows aggressively).
3. **Medium-insert (MI):** For each  $x$  insertions, delete  $x/2$  vectors (2:1 ratio, more stable growth).
4. **Constant-insert (CI):** For each  $x$  insertions, delete  $x$  vectors (1:1 ratio, constant index size).

### Delete Method

The delete method defines which vectors are selected for deletion:

1. **Random delete (RD):** Uniformly random selection across all indexed vectors.
2. **Oldest delete (OD):** Delete the oldest inserted vectors, simulating a sliding time window.

### Data Distribution

Data distribution parameter defines how the data changes from one batch to the next:

1. **Uniform distribution (UD):** Data is uniformly distributed across the embedding space.
2. **Shifted distribution (SD):** Batches are drawn from sequentially ordered clusters, cutting across cluster boundaries. This creates a gradual distribution shift as the indexed data transitions from one cluster to the next over time.

The LAION2B [64] dataset, used in most experiments, has a uniform distribution, with queries also uniformly distributed across the same embedding space. To evaluate DLI performance under distribution shift, we create a clustered version of the LAION2B 10M subset using k-means clustering with 10 clusters. The clusters are ordered by proximity. We select a random starting cluster, then iteratively append the cluster nearest to the most recently added cluster. This sequential ordering enables a gradual distribution shift, simulating a more realistic drift than ordering the clusters randomly.

### Query Distribution

The query distribution parameter defines the relationship between queries and indexed data:

1. **In-distribution (ID) queries:** Queries are drawn from the same distribution as the inserted data and remain constant across all batches.
2. **ID-sampled queries (IDS):** Queries are sampled from a pool of vectors set aside during insertion.<sup>1</sup> Under shifting data distributions, these queries remain in-distribution relative to the current dataset, but the query set changes across batches.

---

1. Vectors in the query pool are never inserted into the index, ensuring no overlap between indexed and query vectors.

**Table 4.1:** Experimental scenarios.

| Scenario Name | Workload  | Data Distr. | Query Distr. | Batches | Delete Method |
|---------------|-----------|-------------|--------------|---------|---------------|
| Insert-only   | OI        | UD          | ID           | 1 + 20  | —             |
| Heavy-R       | HI (10:1) | UD          | ID           | 1 + 20  | Random        |
| Medium-R      | MI (2:1)  | UD          | ID           | 1 + 20  | Random        |
| Constant-R    | CI (1:1)  | UD          | ID           | 1 + 20  | Random        |
| Heavy-O       | HI (10:1) | UD          | ID           | 1 + 20  | Oldest        |
| Medium-O      | MI (2:1)  | UD          | ID           | 1 + 20  | Oldest        |
| Constant-O    | CI (1:1)  | UD          | ID           | 1 + 20  | Oldest        |
| Shift-ID      | OI        | SD          | IDS          | 20      | —             |
| Shift-OOD     | OI        | SD          | OODQ         | 20      | —             |
| Shift-M       | OI        | SD          | MQ           | 20      | —             |

3. **Out-of-distribution (OOD) queries:** Queries are sampled from a different distribution than the inserted data.
4. **Mixed queries (MQ):** Queries remain constant throughout the experiment, but the data distribution shifts over time. As a result, queries that are initially out-of-distribution gradually become in-distribution as the indexed data’s distribution migrates toward the query distribution.

We distinguish between ID and IDS queries because IDS requires setting aside a portion of the dataset as a query pool, reducing the amount of indexed data compared to the ID scenario. This may affect search quality. Similarly, the OOD scenario requires setting aside an entire cluster for queries. However, we primarily compare configurations within each scenario rather than across scenarios, so these differences in dataset size should not affect our conclusions.

Table 4.1 summarizes the parameter combinations used during experiments, referred to as scenarios. For the number of batches, value 1 + 20 indicates one initial batch, without deletion, and 20 standard batches.

**Table 4.2:** Datasets used in experiments.

| Dataset           | Dimension | Size | Distance Function |
|-------------------|-----------|------|-------------------|
| LAION2B-300K [64] | 768       | 300K | Cosine            |
| LAION2B-10M [64]  | 768       | 10M  | Cosine            |
| LAION2B-100M [64] | 768       | 100M | Cosine            |
| DEEP1B-10M [65]   | 96        | 10M  | L2                |
| VIBE ImageNet [7] | 640       | 250K | Cosine            |
| VIBE Llama [7]    | 128       | 1,2M | Inner Product     |

#### 4.1.1 Datasets

Unless stated otherwise, experiments use the LAION2B-10M dataset. Final evaluation of the optimal DLI configuration is performed on all datasets in Table 4.2, which lists their sizes, dimensionality, and distance functions.

#### 4.1.2 Hardware

All the experiments were conducted on Metacentrum to ensure consistent hardware for efficiency measurements (for more details, see Appendix A.1). The only exception is Section 4.5.1, which involved only initial experiments for pruning of retrain strategies based on recall, not efficiency.

#### 4.1.3 Implementation

The DLI was implemented in Python. We used PyTorch [62] for efficient handling of vectors and Faiss [18] for k-means and KNN computations.

The implementations of indexes we use for comparison in Section 4.7 often only support bulk insertion or use different approximation parameters for search. The next sections describe the specific settings and adjustments made to evaluate these indexes in the same experiments as DLI.

## HNSW

For HNSW we used the Faiss Library [18] implementation with standard parameters:  $M=16$ ,  $efConstruction=100$ ,  $efSearch=100$ .

Because Faiss HNSW does not natively support even soft deletion, we have extended the index with soft delete functionality by keeping track of the vectors that would be deleted. During the search, we pass this list to the HNSW implementation, which is able to filter them out from the results, so we do not have to implement additional filtering.

## IVFFlat

For IVFFlat, we also used the Faiss Library [18]. The number of partitions ( $nlist = \sqrt{first\ batch}$ ) was chosen based on the recommendations in [66]. This index uses the number of buckets ( $nprobe$ ) as the approximation parameter. During search, we set it to imitate the  $n_{candidates} = 50K$  parameter in DLI to  $nprobe = 50K / (Avg\ bucket\ size)$ . This means  $nprobe$  decreases as more data is inserted into the index.

## Periodically Rebuilt LMI

For best comparability, we use the same extended LMI version as in the DLI implementation. The  $n_{buckets} = 11110$  was set to match the total number of buckets in the compared DLI index, once the full 10M of vectors are inserted. Additionally, we set the number of epochs to 15, and the number of hidden neurons to 512, based on the recommendations in [14]. The rebuild period was set arbitrarily to 4 batches.

### 4.1.4 Memory measurements

Section 4.7.1 compares the memory overhead of each index structure, excluding vector storage. For LMI and DLI, memory usage is measured directly using PyTorch’s `element_size` function to account for routing model parameters and vector identifiers. For IVFFlat and HNSW, memory overhead is estimated based on their internal structure documentation, as the Faiss library does not expose these measurements. Details are provided in Appendix A.4.2.

#### 4.1.5 Search parameters

All experiments performed search with  $k = 30$ . The experiments in Section 4.2 had an unlimited candidate budget to allow computation of search quality metrics at all recall thresholds.

For all the other experiments, with DLI or LMI  $n\_candidates$  was set to 50K. IVFFlat and HNSW had separate search parameters, based on their implementation specifics described in the previous section.

### 4.2 DLI comparison to LMI

In the first part of the experiments, we validate that temporal decomposition is a viable approach for dynamizing learned indexes. The construction cost savings of this approach are established by the Bentley-Saxe decomposition and demonstrated by LSM trees. However, LSM trees assume exact search on each level, so distributing the data into levels does not degrade search quality. For ANN search, this raises the question of whether multilevel navigation degrades index performance.

Because this initial experiment serves as proof of concept for the multilevel architecture, it does not follow the batch-oriented design from Section 4.1. We use the clustered LAION2B-300K dataset with ID queries, varying which portion is used for training to simulate distribution shift across the following scenarios: *LMI Static Baseline* (trained on the full dataset), *LMI Slight Shift* (trained on three clusters), *LMI Extreme Shift* (trained on one cluster), and *DLI Snapshot*. DLI Snapshot distributes data evenly across 7 levels with arity  $a = 3$  (270 vectors per bucket, empty buffer). While a distribution shift occurs between levels, each level is trained on all its data, ensuring balanced distribution within the level. This idealized setup isolates the multilevel architecture’s impact from, e.g., compaction and retraining strategies evaluated later.

All LMI variants use 1092 buckets, matching DLI’s total across all levels, ensuring bucket count does not affect partitioning quality differences. Both LMI and DLI use the default training configuration (Table A.3), with DLI using the MDN\_SS search strategy.

Routing and partitioning quality of the LMI variants was evaluated in Section 2.6. Applying the same metrics to DLI Snapshot (detailed

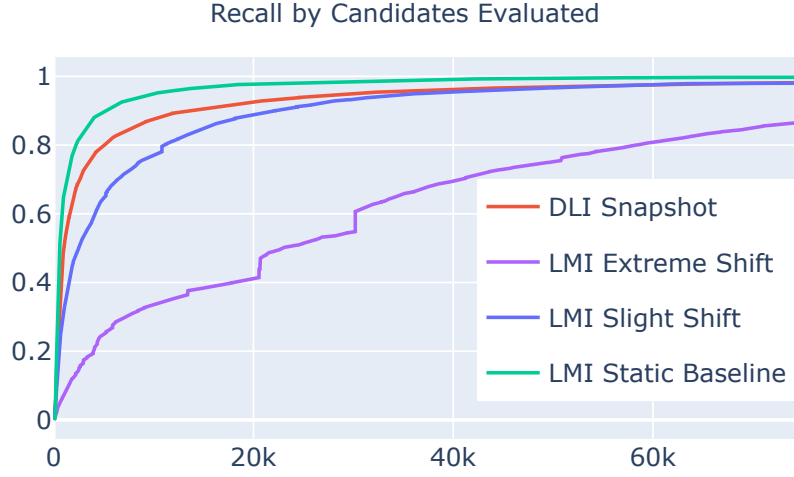


**Figure 4.1:** Recall curves demonstrating DLI’s effectiveness compared to LMI. DLI outperforms the shift scenarios at recall levels below 0.95 (e.g., 15K vs 50K candidates for 0.9 recall). LMI starts showing an advantage in the Slight Shift scenario after 60K candidates, which is an unrealistic candidate budget for practical applications.

in Appendix A.5) reveals two key findings. First, DLI significantly improves GT Candidates, achieving results nearly identical to the LMI Static Baseline. Second, DLI exhibits slightly worse routing quality, visiting more irrelevant buckets overall. However, the next section demonstrates that strong partitioning quality compensates for this routing inefficiency, maintaining competitive overall performance.

#### 4.2.1 DLI vs LMI Overall Search Quality

Figure 4.1 shows the recall curves. DLI achieves higher recall at lower candidate counts than both Extreme and Slight Shift scenarios, confirming that although the multilevel architecture introduces routing inefficiency, the partitioning quality improvement outweighs this drawback.

The routing degradation only becomes visible at very high recall thresholds (above 95%), where Slight Shift LMI begins to outper-

form DLI. However, realistic search scenarios would not operate at these thresholds: reaching 95% recall in this experiment requires approximately 60K candidates, which is 20% of the entire dataset. At practical recall levels, DLI maintains superior performance. Because DLI buckets are more balanced, visiting an irrelevant bucket in DLI is less costly than incorrectly visiting the overpopulated buckets present in distribution shift scenarios.

Based on these results, we conclude that DLI is a viable solution to the LMI degradation problem. The following sections describe the three rounds of experiments aimed at identifying the optimal DLI configuration.

### 4.3 Structural Experiments

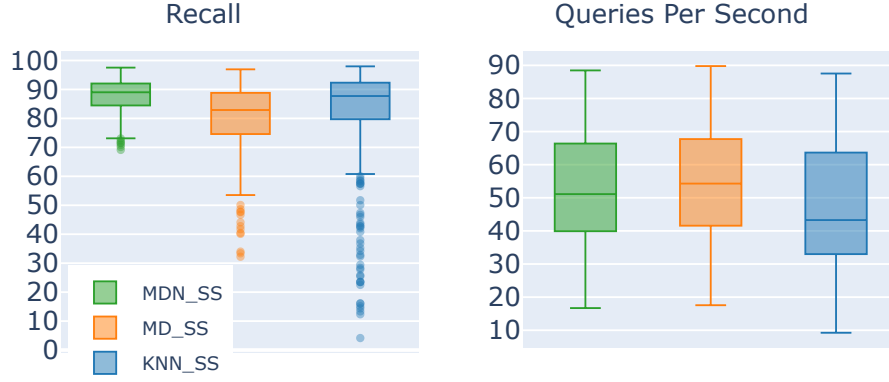
The first round of experiments focuses on the structural parameters and evaluates them on the Insert-only scenario. This allows us to isolate the impact of these parameters on index structure without the additional effects of distribution shifts.

We run the Insert-only experiment on all the combinations of arity, buffer size, compaction strategy, and search strategy from Table A.1, obtaining statistics about inserts and search operations for all batches.

#### 4.3.1 Search Strategy

We begin by evaluating the search strategy, as it has the greatest impact on recall, which is one of our primary metrics. Additionally, the search strategy decides bucket ordering during search, influencing routing quality but not partitioning quality. The other parameters (arity, buffer size, compaction strategy) affect partitioning quality. This independence allows us to evaluate the search strategy first and fix it for subsequent experiments.

Figure 4.2 shows recall and queries per second (QPS) grouped by search strategy across all experiments. MDN\_SS outperforms the other strategies on both metrics. For search effectiveness, MDN\_SS has the least variance in recall, avoiding extreme outliers below 50% present in both KNN\_SS and MD\_SS. These outliers occur when the strategy explores buckets on the wrong level, most frequently in KNN\_SS



(a) Recall by Search Strategy

(b) QPS by Search Strategy

**Figure 4.2:** Search strategy performance across all configurations. MDN\_SS (Model Driven Normalized) achieves the best recall with median 90% and tight distribution, avoiding the catastrophic outliers below 50% present in MD\_SS and KNN\_SS. While MD\_SS has marginally higher throughput (55 vs 52 QPS), MDN\_SS’s superior recall stability makes it the clear winner. KNN\_SS performs worst on both metrics.

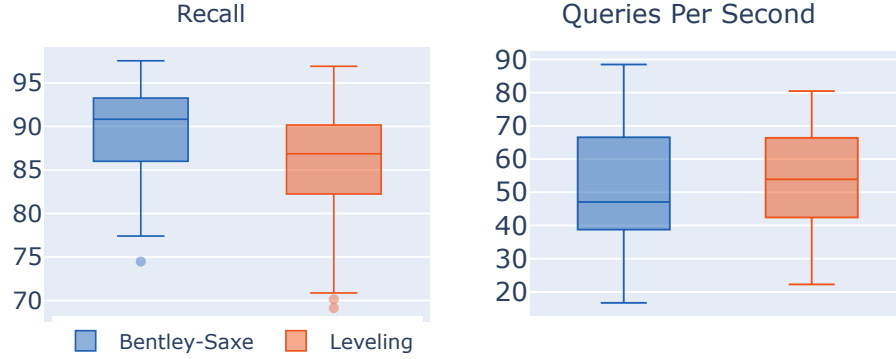
because it changes levels arbitrarily without considering the routing model predictions.

In terms of efficiency, KNN\_SS performs worst due to its more complicated bucket ordering process. MD\_SS has slightly higher average throughput than MDN\_SS (55 vs 52 QPS), but this difference is negligible compared to MDN\_SS’s recall improvement.

Based on these observations, we use MDN\_SS as the search strategy in all subsequent experiments.

#### 4.3.2 Compaction Strategy

With the search strategy fixed, we compare the two compaction strategies across all combinations of arity and buffer size (see Table A.1 for all tested values). Although the compaction strategy does not directly impact search, it affects partitioning quality, which in turn affects search quality.



(a) Recall by Compaction Strategy (b) QPS by Compaction Strategy

**Figure 4.3:** Compaction strategy performance across all configurations. BS\_C achieves higher recall, averaging 90%, while L\_C improves search efficiency (53 QPS compared to 47 for BS\_C).

Figure 4.3 shows recall and QPS values measured across all batches. Bentley-Saxe compaction (BS\_C) leads to higher recall, averaging 90%, while Leveling (L\_C) improves search efficiency (53 QPS compared to 47 for BS\_C). The difference in QPS may seem unexpected because BS\_C performs more aggressive compaction, which should lead to a smaller number of levels and thus to faster search. To understand this behavior, we need to look at the distribution of search time.

The total search time consists of two main parts: model evaluation time (the query is passed through the model on each non-empty level) and linear search time (each visited bucket is searched exhaustively).

As mentioned, BS\_C is more aggressive, merging multiple levels into one and leaving other levels empty. This means it spends less time on model evaluation than L\_C, where the number of levels only increases.

However, more levels also mean data is more distributed, leading to more granular buckets. This gives L\_C an advantage: smaller buckets prevent “overshooting” the candidate budget in the final exhaustively searched bucket. Figure 4.4 confirms this. While BS\_C has lower model evaluation overhead, its linear search time is much higher than L\_C’s. Because model evaluation overhead is an order of magnitude smaller

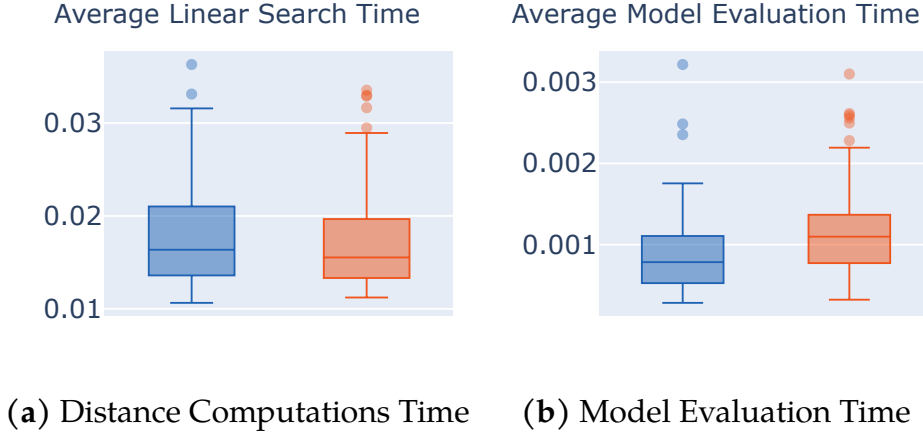


**Figure 4.4:** Effect of compaction strategy on search time components. BS\_C has lower model evaluation overhead, but its linear search time is higher than L\_C’s. Because the search inside buckets dominates over the model evaluation, L\_C is the more efficient strategy.

than linear search time, L\_C’s advantage in linear search outweighs its model evaluation overhead, leading to higher overall QPS.

Finally, we examine insert performance. Figure 4.5 shows that inserts per second (IPS, on logarithmic scale) are generally higher for BS\_C. While lower-end outliers are similar for both strategies, BS\_C achieves much higher upper-end outliers, with some batches reaching up to 20K IPS compared to L\_C’s maximum of 3K IPS.

The similarity in the low range occurs because creating a new level costs approximately the same in both strategies. The main cost component is model training, which scales linearly with the number of training vectors. Since both strategies use the same sample threshold for training, the cost of creating a new level is identical. BS\_C has additional overhead from accumulating data from all higher levels (Leveling merges only one level at a time), but this is marginal compared to the training cost.

However, L\_C moves data across multiple levels during a single compaction. While the first move costs the same as in Bentley-Saxe, subsequent moves, though cheaper because they involve smaller levels, still increase total compaction time, explaining Leveling’s lower maximum IPS.

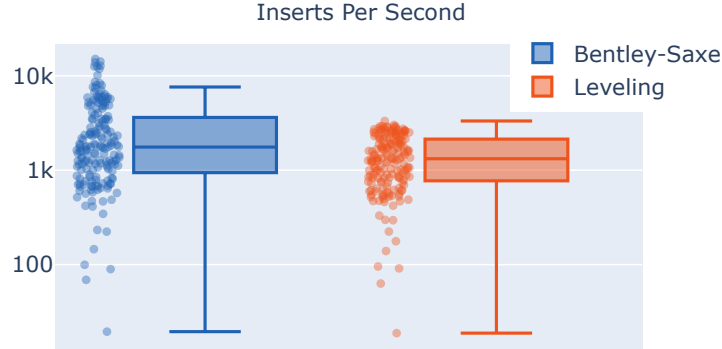


**Figure 4.5:** Insert throughput across all configurations. Both strategies show similar lower-end outliers, but BS\_C has much higher upper-end outliers, overall leading to faster insert performance.

In summary, although L\_C achieves slightly higher QPS, BS\_C provides significantly better recall and insert throughput, making it the superior choice overall. We use the combination of BS\_C and MDN\_SS for the next experiments, which evaluate the impact of arity and buffer size.

### 4.3.3 Arity and Buffer Size

As mentioned earlier, we conducted experiments with all the combinations of arity (3, 5, 10) and buffer size (1K, 5K, 10K). Table 4.3 shows average recall, QPS, IPS, and time cost per batch for each configuration.

For a fixed buffer size, higher arity leads to higher recall but lower QPS and IPS. This occurs because, although higher arity means fewer levels, managing one large level is more costly than managing multiple smaller levels.

For fixed arity, the trend reverses. Higher buffer size leads to lower recall but higher QPS and IPS. This occurs because a higher buffer size allows levels to hold more data, reducing the number of levels and improving efficiency. However, this also means buckets contain more candidates on average, making navigation less precise.

Despite this precision tradeoff, the recall advantage of high arity remains dominant:  $a = 10$  achieves the three highest average recalls.

**Table 4.3:** Arity and buffer size experiments.

| Configuration      | Avg Cost (s) | Avg IPS | Avg QPS | Avg Recall |
|--------------------|--------------|---------|---------|------------|
| $a = 10$ $b = 10k$ | 83.89        | 8035.36 | 68.85   | 88.60      |
| $a = 10$ $b = 5k$  | 366.12       | 5801.18 | 40.58   | 92.18      |
| $a = 10$ $b = 1k$  | 610.60       | 2720.60 | 41.20   | 94.35      |
| $a = 3$ $b = 10k$  | 213.75       | 2894.99 | 69.31   | 84.56      |
| $a = 3$ $b = 5k$   | 304.41       | 1897.14 | 42.39   | 88.64      |
| $a = 3$ $b = 1k$   | 1973.26      | 875.19  | 38.36   | 94.19      |
| $a = 2$ $b = 10k$  | 335.30       | 1581.10 | 65.57   | 83.04      |
| $a = 2$ $b = 5k$   | 454.54       | 1193.57 | 54.82   | 87.81      |
| $a = 2$ $b = 1k$   | 1260.37      | 586.11  | 37.99   | 93.40      |

Based on a more detailed analysis of the configurations across the whole dynamic scenario experiments on the 100M dataset (see Appendix A.6), we conclude that the arity and buffer size should be scaled based on the dataset size, with  $a = 10$ ,  $b = 5k$  being the best general choice.

In the following experiments, we use the MDN\_SS search strategy, BS\_C compaction, and  $a = 10$ ,  $b = 5K$  configuration to evaluate rebuild and training strategies.

#### 4.4 Delete Experiments

For the second round of experiments, we focus on delete functionality, evaluating all delete scenarios from Table 4.1 with three rebuild strategies: Greedy, Basic, and No Rebuild.

Similar to the compaction, rebuild does not directly affect the search process. However, it creates different data distributions across the index, which has an influence on insert and search performance. We therefore report QPS, recall, and IPS metrics alongside delete throughput (DPS).

First, we consider just the scenarios with Oldest Delete. Table 4.4 shows that the rebuild strategy has little to no impact on performance. This is expected: due to DLI’s temporal organization, the oldest data always resides in the lowest level. Because it is the largest level, delet-

**Table 4.4:** Delete experiments — oldest delete method.

| Rebuild   | Scenario   | Avg DPS | Avg QPS | Avg Recall |
|-----------|------------|---------|---------|------------|
| Greedy    | Constant-O | 8906.83 | 63.60   | 95.44      |
| Basic     | Constant-O | 8556.38 | 65.67   | 95.44      |
| NoRebuild | Constant-O | 9134.16 | 104.68  | 95.44      |
| Greedy    | Medium-O   | 7228.61 | 37.37   | 92.60      |
| Basic     | Medium-O   | 7230.09 | 43.29   | 92.51      |
| NoRebuild | Medium-O   | 7490.39 | 78.74   | 92.47      |
| Greedy    | Heavy-O    | 7526.34 | 46.43   | 92.38      |
| Basic     | Heavy-O    | 7966.01 | 51.95   | 92.16      |
| NoRebuild | Heavy-O    | 8294.27 | 64.45   | 91.93      |

**Table 4.5:** Delete experiments — random delete method.

| Rebuild   | Scenario   | Avg DPS | Avg QPS | Avg Recall |
|-----------|------------|---------|---------|------------|
| Greedy    | Constant-R | 8210.63 | 58.19   | 96.41      |
| Basic     | Constant-R | 8345.59 | 61.61   | 95.95      |
| NoRebuild | Constant-R | 9071.15 | 110.58  | 95.87      |
| Greedy    | Medium-R   | 7307.69 | 68.60   | 93.01      |
| Basic     | Medium-R   | 7285.17 | 70.49   | 92.94      |
| NoRebuild | Medium-R   | 7969.14 | 96.56   | 92.73      |
| Greedy    | Heavy-R    | 7406.98 | 55.42   | 92.12      |
| Basic     | Heavy-R    | 7113.59 | 43.62   | 91.97      |
| NoRebuild | Heavy-R    | 7800.00 | 68.81   | 92.01      |

ing from it rarely causes underutilization and triggers a rebuild. When it does, upper levels are typically non-empty, so the rebuild procedure merges one or more upper levels into the lowest level, which is equivalent to standard compaction. This exemplifies the temporal organization advantage: deleting old data is efficient and has minimal impact on search performance.

For the Random Delete variant, in Table 4.5 we see a clear trend — No Rebuild has the highest throughput for all operations, Greedy achieves the highest recall, and Basic generally remains in the middle.

Based on additional observations described in the Appendix A.7, we conclude that Greedy is generally the optimal choice for rebuild strategy, as it prevents index degradation into underutilized levels, has the same computational cost as Basic, and delivers better recall. Insert heavy settings can briefly lead to worse recall with this strategy, but in that case, the compaction process will quickly rectify this. If we could assume frequent inserts that trigger compaction often enough to prevent prolonged suboptimal states, No Rebuild is also a valid option.

## 4.5 Training Experiments

In the final round of configuration experiments, we evaluate the best setup for the routing model by running experiments with different data and query distributions.

### 4.5.1 Retrain Strategy

We evaluate retrain strategies from Section 3.3.2 (From Scratch, Full, No Retrain) as well as Partial strategies that reuse model weights but modify training configuration. All strategies start with the default training configuration defined in Table A.3. The three Partial strategies do the following modifications during retraining:

- **Partial A:** Trains for the full 10 epochs while preserving optimizer state, resulting in less aggressive weight updates.
- **Partial B:** Resets optimizer state but trains for only 20% of normal epochs (2 epochs), resulting in fewer, more aggressive updates.

- **Partial C:** Combines both modifications: resets optimizer state and trains for only 2 epochs.

For initial training configuration, we use the default settings from Section 2.3.1. Due to the large number of strategies, we start with preliminary experiments on the LAION2B-300K subset, evaluating just the Insert-only and Shift-M scenarios.

Table 4.6 shows that Partial strategies generally perform worse than Full and From Scratch, with one exception: Partial A performs comparably in the uniform scenario with ID queries, though this may be anomalous. These results align with the discussion in Section 3.3.2. In scenarios without distribution shift, reusing the model without retraining suffices and provides significant efficiency gains. However, under a distribution shift, recall drops without retraining. Full and From Scratch strategies show nearly identical results for both search quality and insert throughput. Based on additional experiments included in the Appendix A.7.1, we decided to use From Scratch as the main retrain strategy moving forward.

**Table 4.6:** Retrain strategy experiments.

| Retrain             | Scenario    | IPS     | QPS    | Avg Recall   |
|---------------------|-------------|---------|--------|--------------|
| <b>From Scratch</b> | Shift-M     | 1095.72 | 92.39  | <b>74.60</b> |
| Full                | Shift-M     | 1037.74 | 81.25  | 73.06        |
| No Retrain          | Shift-M     | 4077.80 | 154.99 | 66.23        |
| Partial A           | Shift-M     | 1413.61 | 114.34 | 73.80        |
| Partial B           | Shift-M     | 2005.73 | 119.47 | 66.14        |
| Partial C           | Shift-M     | 2067.24 | 129.34 | 66.74        |
| From Scratch        | Insert-only | 1193.81 | 99.29  | 86.75        |
| Full                | Insert-only | 1273.30 | 96.68  | 86.24        |
| No Retrain          | Insert-only | 4025.72 | 124.68 | 86.47        |
| <b>Partial A</b>    | Insert-only | 1178.32 | 85.08  | <b>87.01</b> |
| Partial B           | Insert-only | 1742.11 | 101.03 | 82.92        |
| Partial C           | Insert-only | 2021.07 | 118.04 | 81.81        |

## 4.6 Evaluation on multiple Datasets

Based on the findings from the previous sections, we establish the following optimal DLI configuration:

- **Search strategy:** Model Driven Normalized
- **Compaction strategy:** Bentley-Saxe
- **Rebuild strategy:** Greedy
- **Retraining strategy:** From Scratch
- **Training parameters:** 10 epochs, no early stopping
- **Arity and buffer size:** Dataset-dependent;  $a = 10$ ,  $b = 5K$  for datasets in the tens of millions

We evaluate this configuration across multiple datasets, using the Insert-only scenario (see Table 4.7). Insert throughput scales with arity and buffer size: the 100M dataset achieves the highest IPS using  $a = 10$ ,  $b = 10K$ , as expected from the previous experiments.

Recall is lowest on the 100M dataset due to the fixed 50K candidate budget, which does not scale with dataset size. Given the fact that the index contains 10 times more data than in the 10M variant, and the query throughput is comparable, this recall drop is not unreasonable.

DLI maintains a relatively high recall even VIBE datasets (Llama, ImageNet) with custom out-of-distribution queries.<sup>2</sup> However, we do see a decrease from the >90% recall values on the similarly sized LAION2B and DEEP1B.

Overall, these results demonstrate that DLI is a viable solution for indexing datasets of various sizes while maintaining good recall and adapting to dynamic workloads.

---

2. VIBE OOD queries differ from our simulated OOD queries: VIBE uses queries generated by different embedding models, while ours are a distinct cluster of the same dataset as we insert into the index

**Table 4.7:** Evaluation of DLI on multiple datasets.

| Dataset       | a  | b   | IPS     | QPS    | Avg Recall |
|---------------|----|-----|---------|--------|------------|
| LAION2B-100M  | 10 | 10k | 6436.68 | 57.48  | 80.35      |
| DEEP1B        | 10 | 1k  | 4420.27 | 86.73  | 90.58      |
| VIBE Llama    | 3  | 1k  | 4661.25 | 132.75 | 87.46      |
| LAION2B-10M   | 10 | 5k  | 5801.18 | 40.58  | 92.18      |
| Vibe ImageNet | 2  | 1k  | 1061.49 | 72.87  | 84.96      |

## 4.7 Comparison to Other Indexes

Finally, we compare the optimal DLI configuration to other indexes, namely IVFFlat, HNSW, and a periodically rebuilt LMI, implemented as described in Section 4.1.3.

We evaluated these indexes using the LAION2B-10M dataset. The measured values are summarized in Table 4.8 for distribution shift scenarios and Table 4.9 for delete workloads. We compare them to two DLI configurations: DLI OPT and DLI S. DLI OPT is the optimal configuration reached in the previous sections, with  $a = 10$  and  $b = 5k$ . DLI S is almost identical but uses  $a = 2$  and  $b = 1k$  for a better adaptation to distribution shifts. The following sections describe the results and insights from these experiments.

### 4.7.1 Memory Efficiency

HNSW consumes the most memory, as expected for a graph-based index that maintains explicit neighbor connections. The soft delete furthers this issue because no space is reclaimed, and additional memory is used to keep track of the deleted vectors. IVFFlat and LMI show similar memory usage, while DLI (in both versions) achieves the lowest memory footprint, marginally better than IVFFlat and LMI, but substantially better than HNSW.

**Table 4.8:** Comparison of DLI with other indexes — data and query distributions.

| Index   | Scenario    | Mem/MB | IPS     | QPS    | Avg Recall |
|---------|-------------|--------|---------|--------|------------|
| DLI OPT | Shift-ID    | 31.86  | 4447.30 | 49.67  | 78.25      |
| DLI S   | Shift-ID    | 31.46  | 591.71  | 33.43  | 89.16      |
| HNSW    | Shift-ID    | 696.36 | 64.38   | 411.30 | 86.44      |
| IVFFlat | Shift-ID    | 41.21  | 6800.03 | 13.15  | 86.81      |
| LMI     | Shift-ID    | 42.56  | 3388.18 | 14.42  | 94.40      |
| DLI OPT | Shift-M     | 32.49  | 3475.94 | 34.54  | 68.78      |
| DLI S   | Shift-M     | 31.69  | 529.98  | 22.35  | 83.60      |
| HNSW    | Shift-M     | 710.24 | 64.36   | 524.65 | 84.17      |
| IVFFlat | Shift-M     | 42.14  | 8319.47 | 9.06   | 84.27      |
| LMI     | Shift-M     | 43.04  | 3747.67 | 15.96  | 90.28      |
| DLI OPT | Shift-OOD   | 35.96  | 4353.77 | 41.71  | 53.63      |
| DLI S   | Shift-OOD   | 30.16  | 566.01  | 36.18  | 76.08      |
| hnsw    | Shift-OOD   | 687.86 | 68.01   | 737.53 | 65.76      |
| IVFFlat | Shift-OOD   | 40.76  | 9140.58 | 12.83  | 74.65      |
| LMI     | Shift-OOD   | 42.36  | 3405.60 | 12.24  | 83.94      |
| DLI OPT | Insert-only | 38.44  | 5380.03 | 52.01  | 92.18      |
| DLI S 2 | Insert-only | 31.74  | 495.26  | 26.61  | 93.40      |
| HNSW    | Insert-only | 701.79 | 74.68   | 799.94 | 93.99      |
| IVFFlat | Insert-only | 41.59  | 6615.29 | 60.50  | 91.62      |
| LMI     | Insert-only | 42.79  | 3693.08 | 17.35  | 96.65      |

#### 4.7.2 Throughput Comparison

##### Insert-Only Workloads

DLI OPT provides an excellent balance between insert and query throughput. In contrast, DLI S has lower QPS and significantly lower IPS, as was expected due to its need to manage more levels. However, it is still higher than the insert throughput of HNSW.

HNSW is optimized for search speed, achieving the highest QPS but at the cost of insert throughput two orders of magnitude lower than other indexes. It is important to note that the extremely low insert performance of HNSW is caused by the fact that HNSW is

**Table 4.9:** Comparison of DLI with other indexes — delete workloads.

| Index   | Exp   | Mem/MB   | IPS     | DPS      | QPS    | Avg Recall |
|---------|-------|----------|---------|----------|--------|------------|
| DLI OPT | HI/RD | 38.70    | 6160.84 | 7406.98  | 55.24  | 92.27      |
| DLI S 2 | HI/DR | 32.50    | 587.45  | 8196.95  | 30.64  | 93.44      |
| HNSW    | HI/RD | 2087.15  | 51.04   | —        | 403.71 | 93.76      |
| IVFFlat | HI/RD | 42.67    | 6091.45 | 403.25   | 46.28  | 93.90      |
| LMI     | HI/RD | 42.85    | 3916.90 | 9089.91  | 17.83  | 96.72      |
| DLI OPT | MI/RD | 29.91    | 7502.17 | 7616.19  | 100.41 | 93.24      |
| DLI S 2 | MI/RD | 24.61    | 686.21  | 7649.50  | 40.12  | 95.07      |
| HNSW    | MI/RD | 7374.34  | 50.11   | —        | 357.86 | 92.81      |
| IVFFlat | MI/RD | 28.99    | 6000.17 | 608.53   | 68.34  | 95.94      |
| LMI     | MI/RD | 36.06    | 3831.83 | 9351.90  | 14.45  | 97.49      |
| DLI OPT | CI/RD | 20.13    | 7566.29 | 8210.63  | 57.75  | 96.35      |
| DLI S 2 | CI/RD | 13.08    | 676.76  | 5211.77  | 43.18  | 97.09      |
| HNSW    | CI/RD | 13983.21 | 50.03   | —        | 411.66 | 84.64      |
| IVFFlat | CI/RD | 12.13    | 6481.12 | 1324.18  | 94.19  | 98.68      |
| LMI     | CI/RD | 27.13    | 4037.74 | 10199.99 | 12.35  | 98.84      |

optimized for bulk updates. Inserting vectors one by one represents the worst-case scenario for this index. Additionally, the IPS falls down significantly in higher batches due to the increasing dataset size. In practice, both of these issues could be partially mitigated by using a buffer and periodically rebuilding the index.

IVFFlat achieves the highest insert throughput among all indexes. On uniform data, it also outperforms DLI in QPS. However, under distribution shift, IVFFlat’s QPS drops dramatically, as it is not designed for dynamic workloads and degrades significantly when data distribution changes.

LMI shows the lowest QPS in all scenarios and lower insert throughput than DLI, but the differences are not as extreme as with HNSW.

### Workloads with Deletions

DLI OPT balances the extreme difference in throughputs for all operations. It achieves the highest IPS values, vastly outperforming HNSW, although the gap with IVFFlat and LMI is smaller. In DPS, it is firmly in the middle between the IVFFlat and LMI. We do not consider HNSW in the DPS comparison due to its soft delete mechanism, which is not comparable to hard deletes. Finally, even though HNSW maintains superior QPS by far, even in this metric, DLI shows a significant improvement over LMI and comparable performance to IVFFlat.

For DLI S, we see the same trends as previously, with significantly reduced performance in all operations compared to DLI OPT.

#### 4.7.3 Search Quality

##### Insert-Only Workloads

DLI’s primary limitation is search quality under a distribution shift. In the uniform scenario, DLI OPT achieves competitive recall, even slightly outperforming IVFFlat. When data distribution shifts, DLI OPT exhibits the lowest recall among all indexes.

LMI maintains the highest recall across all scenarios. Its scheduled rebuilds prevent the degradation observed in DLI, but they also lower the average insert throughput. However, LMI still has significantly more IPS than HNSW, suggesting it is a better option for dynamic workloads with expected distribution shift.

IVFFlat shows second-best recall even under distribution shift, with a high IPS, but this comes with a hidden cost of slow search. Distribution shift causes inserted vectors to concentrate in a small subset of buckets. When queries visit these overpopulated buckets, they find many true neighbors but must also process many candidates, significantly overshooting the candidate budget. In the uniform scenario, the absence of overpopulated buckets allows IVFFlat to maintain both good recall and high search speed.

DLI S shows that by scaling down the arity and buffer size, we can make DLI more adaptable to distribution changes. In this version, DLI shows competitive recall values, consistently exceeded only by LMI.

### Workloads with Deletions

In recall, DLI OPT outperforms only HNSW in the Constant Insert scenario, but with a large margin of 10%. DLI Small again shows better adaptability, with recall similar to LMI.

It is important to note that because all Delete scenarios use uniform data distribution, and the growth of the dataset is limited by the deletes, all indexes maintain very high recall and low variance, with LMI being slightly superior in Heavy and Medium insert.

The only exception to this is HNSW in the Constant scenario. Because this scenario contains the most deletes, which reveals how soft deletes degrade HNWS performance.

## 4.8 Summary

The comparison of DLI to LMI, IVFFlat, and HNSW reveals DLI's position in the design space of dynamic indexes: DLI excels in memory efficiency and provides balanced throughput across insert, delete, and query operations in uniform data scenarios.

However, DLI's recall degrades under distribution shift. We are able to mitigate this degradation by scaling down the levels, but it introduces a significant decline in performance. These findings signal that a more frequent retraining or adaptive compaction strategies may be needed to adapt the DLI to distribution shift without decreasing the arity and buffer size.

In scenarios without distribution shift, DLI proved to be a competitive solution, even surpassing HNSW in some delete workloads.

The comparison highlights that the main advantage of DLI is its versatility. Each of the indexes can outperform DLI in a specific metric or scenario, but they usually have to sacrifice their performance in other areas. DLI doesn't prioritize any operation at the cost of others, making it a strong candidate for unpredictable dynamic workloads.

## 5 Conclusion

This thesis addressed the issue of dynamization of learned indexes through a temporal decomposition approach. We have proposed a general framework, which enables any learned index to accommodate various dynamic workloads, provided it implements a minimal interface. The main contributions of this work include: (i) definition of a static learned index interface, (ii) design of a dynamic framework, with multiple strategies for insert, delete and search operations, (iii) implementations of Dynamic Learned Index — an instantiation of the framework with LMI as the base static index, and (iv) thorough evaluation of the DLI’s design space and its comparison to other indexing techniques.

We have designed evaluation metrics for analyzing learned index degradation, decomposing search quality into partitioning quality (GT Buckets, GT Candidates) and routing quality (Wasted Effort).

For each of the core DLI operations (insert, delete, search), several strategies were presented, namely (i) compaction, (ii) search, (iii) rebuild, and (iv) retrain. We found that Bentley-Saxe compaction and Greedy rebuild were superior in their respective categories because they consolidate more data into a single level at a time, maximizing the data available for model training. For search, we designed the Model Driven Normalized (MDN) strategy, which demonstrated that normalizing model outputs enables effective traversal of the multilevel structure. Finally, for retraining, we found that training the model from scratch produces the most consistent results across diverse scenarios.

We used two DLI configurations to compare DLI against HNSW, IVFFlat, and a periodically rebuilt LMI. This comparison showed that DLI with small levels is able to adapt to distribution shifts, optimizing recall, while DLI with levels scaled based on the dataset size has the ability to balance throughput across all three core operations. This trait is a unique feature of the DLI, compared to other indexes, which usually excel in one functionality at the cost of others. As a result, DLI could be a strong index candidate for settings with unpredictable update patterns.

### 5.0.1 Future Work

Based on the limitations revealed by the experiments, we propose the following directions for future work.

**Detection of distribution shift.** By detecting changes in the incoming data, we could bridge the gap in search quality between DLI and other indexes. This would put DLI into the position of the most versatile index across all dynamic settings.

**Adaptation to OOD queries.** An interesting observation is that DLI appears more sensitive to out-of-distribution queries than other indexes. This finding deserves a future investigation into the source of this sensitivity and potential mitigation strategies.

**Flexible arity and buffer size.** In our experiments, we concluded that arity and buffer size should be chosen based on the approximate size of the dataset. For DLI to be truly dynamic without any limitations, we should propose an extension to the framework that would allow for automatic scaling of arity and buffer size with the data volume.

**Evaluation of rebuild strategies on distribution shift scenarios.** The rebuild strategies were evaluated only on the Insert-only workloads with uniform data distribution. It would be interesting to see how they perform in scenarios with distribution shift, and if the No Rebuild strategy would still be viable.

**Evaluation with other learned indexes.** The proposed dynamic framework was implemented and evaluated only on the LMI as the base index. Before it can be presented as a truly general technique, we need to evaluate it in combination with different learned indexes.

## A An appendix

### A.1 Configuration parameters

The implementations of LMI and DLI are highly configurable. Some parameters are given to the index as an input based on the dataset (e.g., distance function), and others can be defined by the user. In the following sections, we provide details of these configurations.

#### A.1.1 DLI Configuration

Table A.1 summarizes the configurable parameters of the dynamic framework. The default values are marked in bold.

**Table A.1:** DLI configuration parameters.

| Parameter                   | Purpose                                                              | Values                                          | Variable          |
|-----------------------------|----------------------------------------------------------------------|-------------------------------------------------|-------------------|
| Arity                       | Defines the rate of expansion of levels                              | 2, 3, <b>10</b>                                 | <i>a</i>          |
| Buffer Size                 | Maximum number of vectors in the buffer                              | 1K, <b>5K</b> , 10K                             | <i>b</i>          |
| Compaction Strategy         | Method for level reorganization                                      | Leveling, <b>Bentley-Saxe</b>                   | <i>compact_s</i>  |
| Search Strategy             | Method for ordering buckets during search                            | KNN_SS, MD_SS, <b>MDN_SS</b>                    | <i>search_s</i>   |
| Rebuild Strategy            | Method for reorganization during delete                              | No_Rebuild, Basic, <b>Greedy</b>                | <i>rebuild_s</i>  |
| Retraining Strategy         | Method for retraining empty index on new data                        | No_Retrain, <b>From_Scratch</b> , Full, Partial | <i>retrain_s</i>  |
| Learned Index Configuration | Definition of the base static index and all its necessary parameters | <b>LMI</b> (Table A.2)                          | <i>lmi_config</i> |

**Table A.2:** LMI configuration parameters.

| Parameter                     | Purpose                                                       | Values                | Variable             |
|-------------------------------|---------------------------------------------------------------|-----------------------|----------------------|
| <b>Number of buckets</b>      | How many partitions are in the cluster                        | $\mathcal{N}$         | $n_{\text{buckets}}$ |
| <b>Dimension</b>              | Dimensionality of the indexes dataset                         | Dataset dependent     | $d$                  |
| <b>Distance Function</b>      | Function used to compare indexed and query vectors            | Dataset dependent     | $df$                 |
| <b>Hidden neurons</b>         | How many neurons are in the hidden layer of the routing model | <b>128</b> , 256, 512 | $n_{\text{neurons}}$ |
| <b>Training configuration</b> | All the parameters required for the training process          | (Table A.3)           | train_config         |

### A.1.2 LMI Configuration

Table A.2 summarizes the LMI configuration parameters, most of which are given by the currently indexed dataset. When using LMI in the dynamic framework, the  $n_{\text{buckets}}$  is set by the framework, based on the arity and current level. When using LMI as a standalone index in Chapter 3, we set the  $n_{\text{buckets}}$  to match the total number of buckets in the DLI we compare it to.

The  $n_{\text{neurons}} = 128$  value was fixed in the DLI based on a series of preliminary experiments, including scaling the hidden layer with the growing index size, but the results showed no notable improvement.

### A.1.3 Training Configuration

All the parameters used for training the LMI the Table A.3. Parameters that have multiple values are evaluated in the experiment chapter, chosen based on preliminary experiments and recommendations from [14]. In bold, we marked the values of the default configuration used in all the DLI index experiments, excluding the ones targeting the training itself in Section 4.5.

**Table A.3:** Training configuration.

| Parameter                   | Purpose                                                                                                      | Values                                                     | Variable            |
|-----------------------------|--------------------------------------------------------------------------------------------------------------|------------------------------------------------------------|---------------------|
| <b>Learning Rate</b>        | Manage the speed of the learning process                                                                     | <b>0.001</b>                                               | <i>lr</i>           |
| <b>Number of epochs</b>     | Maximum epoch executed during learning                                                                       | <b>10, 20, 30</b>                                          | $n_{\text{epochs}}$ |
| <b>Loss Function</b>        | Computing error of the routing model                                                                         | <b>Cross Entropy</b>                                       | <i>loss</i>         |
| <b>Optimizer</b>            | Adjusts the rate of weight updates in addition to learning rate                                              | <b>Adam</b>                                                | <i>optimizer</i>    |
| <b>Class Balancing</b>      | Resamples data before training to ensure balanced representation of all classes                              | <b>True, False</b>                                         | <i>balancing</i>    |
| <b>Early Stopping</b>       | Stops training when validation loss does not improve                                                         | <b>True, False</b>                                         | <i>early_stop</i>   |
| <b>Patience</b>             | How many epochs of non-decreasing validation loss are acceptable before training stops                       | <b>1, 3, —</b>                                             | <i>patience</i>     |
| <b>Sample threshold</b>     | Number of vectors used for training                                                                          | <b>100K</b> , $40 \cdot n_{\text{buckets}}$ , full dataset | <i>sample_size</i>  |
| <b>Validation split</b>     | If early stopping is enabled, defines the percentage of the training sample that is set aside for validation | <b>20%, —</b>                                              | <i>valid_ratio</i>  |
| <b>Retrain from scratch</b> | Defines if model is reset when the index is emptied                                                          | <b>True, False</b>                                         | <i>from_scratch</i> |

## A.2 Static Learned Metric Index Functionality

The following algorithms describe the LMI implementation of the non-trivial operations from Section 2.1. We will be using  $M : \mathbb{R}^{m \times d} \rightarrow \mathbb{R}^{m \times n_{\text{buckets}}}$  to refer to the LMI's routing model and  $B$  to refer to its collection of buckets, with  $b \in B$  being a set of vectors  $b_X$  and identifiers  $b_I$ . To obtain the model's output for a given input vector or a set of vectors, we will be using the notation  $M(x)$ . Finally, the  $oid\_b$  is used to refer to the hash map containing mappings of OIDs to their vector's buckets.

### A.2.1 Delete

The DELETE procedure uses the hash map to locate bucket containing the OID to be deleted.

---

**Algorithm 1:** Delete( $i$ )

---

**Input:** Vector OID to delete

**Output:** Boolean indicating success

**if**  $id \notin oid\_to\_bucket$  **then**

**return** False;

$b_{index} \leftarrow oid\_to\_bucket[oid]$ ;

Delete  $i$  from  $B[b_{index}]$ ;

Update  $oid\_to\_bucket$  to reflect the changes;

**return** True;

---

### A.2.2 Search

This section contains the SEARCH and associated PREDICT procedure.

---

**Algorithm 2:** PREDICT( $q$ )

---

**Input:** Query vector  $q$

**Output:** Sorted list of buckets and their probabilities

$logits \leftarrow M(q)$ ;

$P \leftarrow \text{softmax}(logits)$ ;

$pairs \leftarrow [(B[1], P[1]), (B[2], P[2]), \dots, (B[n_{\text{buckets}}], P[n_{\text{buckets}}])]$ ;

$sorted\_result \leftarrow \text{sort } pairs \text{ by probability (desc)}$ ;

**return**  $sorted\_result$ ;

---

**Algorithm 3:** SEARCH( $q, k, n_{\text{candidates}}$ )**Input:** Query  $q$ , neighbors  $k$ , candidate budget  $n_{\text{candidates}}$ **Output:** Top- $k$  nearest vectors to  $q$ , and their identifierssorted\_B  $\leftarrow$  PREDICT( $q$ ); $C_I \leftarrow \emptyset, C_X \leftarrow \emptyset, C_D \leftarrow \emptyset$ ;**foreach**  $b, p$  *in* sorted\_B **do**    **if**  $|\mathcal{C}_I| \geq n_{\text{candidates}}$  **then**        **break**;     $b_D \leftarrow$  distances to  $q$  for each vectors in  $b_X$ ;     $C_I \leftarrow C_I \cup b_I, C_X \leftarrow C_X \cup b_I, C_D \leftarrow C_D \cup b_D$ ; $I, X \leftarrow$  select top  $k$  results from  $C_X, C_I$  based on  $C_D$ ;**return**  $X, I$ ;**A.2.3 Insert**

LMI's insert handles batches of vectors large enough for model training through two steps: training the routing model and organizing vectors into buckets. The training procedure implements From Scratch retraining by default; other strategies are implemented by disabling the *from\_scratch* flag and modifying the training configuration.

**Algorithm 4:** Insert( $X, I$ )**Input:** List of vectors  $X = \{x_1, x_2, \dots, x_n\}$  and corresponding ids $I = \{i_1, i_2, \dots, i_n\}$ **Output:** Index contains the inserted vectors

// Naive inserts skip training;

**if** *model is not initialized* **or** *index is empty* **then**

// From Scratch retrain strategy;

**if** *model is initialized* **and** *from\_scratch* **then**        Reset  $M$  weights and optimizer;        Mark  $M$  as not initialized;    TrainIndex( $\mathcal{X}$ );**foreach**  $i \in 1 \dots |\mathcal{X}|$  **do**     $b, p \leftarrow$  PREDICT( $X[i]$ );    Insert  $X[i], I[i]$  into  $b$ ;    Update *oid\_to\_bucket* to reflect the changes;

**Algorithm 5:** TrainIndex( $X$ )**Input:** List of vectors  $X = \{x_1, x_2, \dots, x_n\}$ **Output:** Trained neural network model

// Could be true for the No Retrain strategy;

**if**  $n_{epochs} = 0$  **then**└ **return**;

// Sampling;

 $sample \leftarrow \max(sample\_size, 40 \cdot n_{buckets});$ **if**  $early\_stop$  **then**└  $sample \leftarrow sample \cdot (1 + valid\_ratio)$  $sample \leftarrow \min(sample, |X|);$  $X_{sample} \leftarrow$  random sample from  $X$  of size  $sample$  ;

// Obtaining labels;

 $centroids \leftarrow \text{KMeans}(X_{sample}, n_{buckets});$  $y \leftarrow$  bucket labels based on the closest centroids for  $x \in X$ ;**if**  $balance$  **then**└  $X_{sample}, y \leftarrow$  resampled  $X_{sample}$ ;**if**  $early\_stop$  **then**└  $X_{sample}, y, X_{val}, y_{val} \leftarrow$  Split  $X_{sample}, y$  with the  $valid\_ratio$ ;

// Model training

 $patience\_counter, loss_{best} \leftarrow 0, \infty;$ **for**  $e \leftarrow 1$  **to**  $n_{epochs}$  **do**└ **foreach** batch  $(X_b, y_b)$  from  $X_{sample}, y$  **do**└└  $loss \leftarrow lf(M(X_b), y_b);$ └└ Backpropagate  $loss$  and update  $M$  weights;└ **if**  $early\_stopping$  enabled **then**└└  $loss_{val} \leftarrow lf(M(X_{val}), y_{val});$ └└ **if**  $loss_{val} < loss_{best}$  **then**└└└  $loss_{best} \leftarrow loss_{val};$ └└└  $patience\_counter \leftarrow 0;$ └└  $patience\_counter \leftarrow patience\_counter + 1;$ └└ **if**  $patience\_counter \geq patience$  **then**└└└ **break**;Mark  $M$  as initialized;

### A.3 DLI functionality

This section contains algorithms describing the DLI operations defined in 3.2. We will be using  $L$  and *buffer* to refer to the index components.

#### A.3.1 Insert

This section contains the main insert functionality and the associated compaction procedures.

---

**Algorithm 6:** INSERT( $x, i$ )

---

**Input:** Vector  $x$  and its OID  $i$

**Output:** Index contains the inserted vector

Insert  $x, i$  into *buffer*;

**if** *buffer* is at full capacity **then**

**if** *search\_s* = Bentley-Saxe **then**

        COMPACT\_BENTLEY-SAXE;

**return**;

    COMPACT\_LEVELING;

---



---

**Algorithm 7:** COMPACT\_BENTLEY-SAXE()

---

**Input:**

**Output:** Buffer is empty

$X_{all}, I_{all} \leftarrow \text{buffer}_X, \text{buffer}_I$ ;

Empty *buffer*;

**foreach**  $L_i \in L$  **do**

    capacity  $\leftarrow$  free space in  $L_i$ ;

**if** capacity  $\geq |X_{all}|$  **then**

$L_i.\text{INSERT}(X_{all}, I_{all})$ ;

**return**;

$X_l, I_l \leftarrow$  vectors and OIDs in level  $L_i$ ;

$X_{all} \leftarrow X_{all} \cup X_l, I_{all} \leftarrow I_{all} \cup I_l$ ;

$L_i.\text{EMPTY}()$ ;

new\_level  $\leftarrow$  new level with  $n_{\text{buckets}} = a^{|L|} \cdot b$ ;

new\_level.INSERT( $X_{all}, I_{all}$ );

Append new\_level to  $L$ ;

---

---

**Algorithm 8:** COMPACT\_LEVELING()

---

**Input:****Output:** Buffer is emptydata\_volume  $\leftarrow |buffer_X|$ ;stop\_level  $\leftarrow$  null;**foreach**  $L_i \in L$  **do**    capacity  $\leftarrow$  free space in  $L_i$ ;    **if** capacity  $\geq$  data\_volume **then**        stop\_level  $\leftarrow i$ ;        **break**;    data\_volume  $\leftarrow |X|$  on from  $L_i$ ;**if** stop\_level is null **then**    new\_level  $\leftarrow$  new level with  $n_{\text{buckets}} = a^{|L|} \cdot b$ ;    Append new\_level to  $L$ ;    stop\_level  $\leftarrow$  new\_level;**for**  $i \in \text{stop\_level}, \text{stop\_level} - 1 \dots 2$  **do**     $X_i, I_i \leftarrow$  vectors and OIDs in  $L_i - 1$ ;     $L_{i-1}.$ EMPTY();     $L_i.$ INSERT( $X_{all}, I_{all}$ );

---

### A.3.2 Delete

This section contains the main `DELETE` operation and rebuild strategy. The target level selection (from Section 3.4.2) is captured in the Algorithm 11.

---

**Algorithm 9:** `DELETE(oid)`

---

**Input:** OID of the vector to delete

**Output:** Boolean indicating success

**if** *buffer* contains *oid* **then**

    Delete vector with *oid* from *buffer*;

**return** True;

$l \leftarrow \text{null}$ ;

**foreach**  $L_i \in L$  **do**

**if**  $L_i.\text{DELETE}(\textit{oid})$  **then**

$l \leftarrow i$ ;

**break**;

**if**  $l$  is null **then**

**return** False;

$\text{occupancy} \leftarrow |X|$  on  $L_l$ ;

**if**  $\text{occupancy} < b \cdot a^{l-1}$  **then**

`REBUILD`( $L_l$ );

**return** True;

---

**Algorithm 10:** REBUILD( $L_l$ )**Input:** Underutilized level  $L_l$ **Output:** All levels in the index are properly utilizedtarget, source  $\leftarrow$  SELECT\_REBUILD\_PARTICIPANTS( $L_l$ );**if** target = -1 **then**    Flush *buffer* into the first level;    **return**;source\_data  $\leftarrow$  vectors and OIDs in source;

source.EMPTY();

**if** rebuild strategy = Greedy **then**    **foreach**  $L_i \in$  source  $\dots L_1$  **do**        source\_data  $\leftarrow$  source\_data  $\cup$  vectors and OIDs in  $L_i$ ;         $L_i$ .EMPTY();

target.INSERT(source\_data);

**return**;**Algorithm 11:** SELECT\_REBUILD\_PARTICIPANTS( $L_l$ )**Input:** Underutilized level  $L_l$ **Output:** Target and source levels for rebuild $L_{lower} \leftarrow$  nearest non-empty level below  $L_l$ ;**if** free space in  $L_l$  + occupancy of  $L_{lower} \geq$  capacity of  $L_l$  **then**    target  $\leftarrow L_{lower}$ ;    **return** target,  $L_l$ ; $L_{upper} \leftarrow$  nearest non-empty level above  $L_l$ ;**if** free space in  $L_l$  + occupancy of  $L_{upper} \geq$  capacity of  $L_l$  **then**    source  $\leftarrow L_{upper}$ ;    **return**  $L_l$ , source;**if**  $l = 1$  **then**    **return**  $L_l$ , -1;**return**  $L_l$ ,  $L_{l-1}$ ;

### A.3.3 Search

This section contains the main `SEARCH` functionality. The KNN Driven search strategy is depicted in a separate Algorithm 13.

---

**Algorithm 12:** `SEARCH( $q, k, n_{\text{candidates}}$ )`


---

**Input:** Query vector  $q$ , number of neighbors  $k$ , candidate budget

**Output:**  $n_{\text{candidates}}$  Top- $k$  nearest vectors to  $q$ , and their identifiers

**if** *search strategy* = `KNN_SS` **then**

$\text{buckets} \leftarrow \text{KNN\_BUCKETS}(q);$

$\text{buckets} \leftarrow [(\text{buffer}, 1)];$

**foreach**  $L_i \in L$  **do**

**if**  $L_i$  is empty **then**

**continue;**

$\text{level\_predictions} \leftarrow L_i.\text{PREDICT}(q);$

**if** *search strategy* = `MDN_SS` **then**

**foreach**  $(\text{bucket}, p)$  in  $\text{level\_predictions}$  **do**

            Append  $(\text{bucket}, p \cdot a^i)$  to  $\text{buckets};$

    Append  $\text{level\_predictions}$  to  $\text{buckets};$

$\text{buckets} \leftarrow \text{sort buckets by probability (desc)};$

$C_I \leftarrow \emptyset, C_X \leftarrow \emptyset, C_D \leftarrow \emptyset;$

**foreach**  $b, p$  in  $\text{sorted\_B}$  **do**

**if**  $|\mathcal{C}_I| \geq n_{\text{candidates}}$  **then**

**break;**

$b_D \leftarrow \text{distances to } q \text{ for each vectors in } b_X;$

$C_I \leftarrow C_I \cup b_I, C_X \leftarrow C_X \cup b_I, C_D \leftarrow C_D \cup b_D;$

$I, X \leftarrow \text{select top } k \text{ results from } C_X, C_I \text{ based on } C_D;$

**return**  $X, I;$

---

---

**Algorithm 13:** KNN\_BUCKETS( $q$ )

---

**Input:** Query vector  $q$ **Output:** Sorted list of buckets and their probabilitiesall\_results  $\leftarrow [[(buffer, 1)]]$ ;**foreach**  $L_i \in L$  **do**

- level\_predictions  $\leftarrow L_i.PREDICT(q)$ ;
- Append level\_predictions to all\_results;

labels  $\leftarrow [[] \text{ for } i \in 0 \dots |L| - 1]$ ;indices  $\leftarrow []$ ;counter  $\leftarrow 0$ ;counter, indices  $\leftarrow \text{DFS\_ORDER}(0, \text{labels}, \text{indices}, \text{counter})$ ;ordered\_buckets  $\leftarrow []$ ;**foreach**  $level\_index, bucket\_index$  in indices **do**

- Append all\_results[ $level\_index$ ][ $bucket\_index$ ] to ordered\_buckets;

Return ordered\_buckets;

---

---

**Algorithm 14:** DFS\_ORDER( $current\_level, labels, indices, counter$ )

---

**Input:** Current level index, list of already assigned labels for each level, list of indices, and a counter for the current position in the indices list**Output:** current value of counter, list of indices in the order they should be visitedcounter  $\leftarrow \text{counter} + 1$ ;last\_idx  $\leftarrow |labels[current\_level]|$ ;Append ( $current\_level, last\_idx$ ) to indices;Append counter to labels[ $current\_level$ ];**if**  $current\_level$  is the last level in the index **then**

- return** counter, indices;

ForEach  $i \in 0 \dots a$  counter, indices  $\leftarrow$ 

- DFS\_ORDER( $current\_level + 1, labels, indices, counter$ );

**return** counter, indices;

---

**Table A.4:** Used Metacentrum hardware.

| Cluster | CPU           | Cores | Allocated Memory |
|---------|---------------|-------|------------------|
| zenon   | AMD EPYC 7351 | 16    | 200GB            |
| elwe    | AMD EPYC 7532 | 32    | 1500GB           |

## A.4 Experiment Specifications

### A.4.1 Environment

All the experiments described in Chapter 4 were conducted on Metacentrum, to ensure consistent hardware for efficiency measurements. The only exception is Section 4.5.1, which involved only initial experiments for pruning of retrain strategies based on recall, not efficiency. For the 100M dataset, we used the elwe cluster and 32 CPUs. Experiments using all the other datasets were run on the zenon cluster with 8 CPUs. Their specifications are in Table A.4. The allocated memory includes all the overhead required to keep the datasets in memory and collect detailed statistics about the index performance.

### A.4.2 Memory Estimation

Because IVFFlat and HNSW do not expose direct memory measurements, we estimate the memory overhead of these indexes using the following calculations.

### A.4.3 HNSW Memory Estimation

We estimate the HNSW index overhead (excluding vector storage) from three components. The graph structure stores neighbor IDs as 32-bit integers, consuming  $|N| \cdot 4$  bytes, where  $|N|$  is the size of the neighbors array exposed by Faiss. Offset arrays for indexing neighbor lists use 64-bit integers, requiring  $|O| \cdot 8$  bytes for the offsets array ( $O$  is also an accessible parameter). The soft-delete implementation adds a set of deleted IDs as 64-bit integers ( $|D| \cdot 8$  bytes).

$$\text{Total overhead: } M_{\text{HNSW}} = |N| \cdot 4 + |O| \cdot 8 + |D| \cdot 8.$$

#### A.4.4 IVFFlat Memory Estimation

We estimate the IVFFlat index overhead (excluding vector storage) from two components. The quantizer stores  $n_{\text{lists}}$  centroid vectors of dimension  $d$  as 32-bit floats, requiring  $n_{\text{lists}} \cdot d \cdot 4$  bytes. Vector identifiers are stored as 64-bit integers, requiring  $n \cdot 8$  bytes for  $n$  indexed vectors. These values are obtained from the index configuration (`nlist`) and index state (`ntotal`).

$$\text{Total overhead: } M_{\text{IVFFlat}} = n_{\text{lists}} \cdot d \cdot 4 + n \cdot 8.$$

#### A.5 DLI Partitioning and Routing Quality

Continuing the experiments in 2.6, we evaluate partitioning and routing metrics for DLI Snapshot described in Section 4.2

Starting with partitioning quality, Figure A.1 shows that although DLI Snapshot is comparable to the LMI Shift scenarios in the GT Buckets metric, the GT Candidates is almost identical to the Baseline. This is expected behavior because the disjunct levels prevent some similar vectors from being grouped together, but the freshly trained models ensure that buckets are relatively balanced<sup>1</sup>.

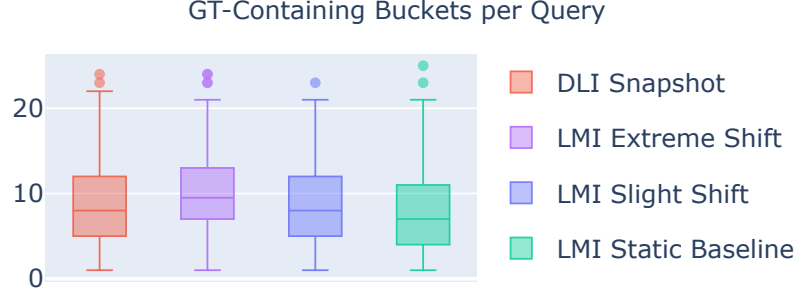
Moving to routing quality, in Figure A.2, we see that DLI performs similarly to Slight Shift, with approximately 0.5 higher wasted effort from 70% recall upward, making it significantly better than Extreme Shift across most of the recall range. Near the 95% threshold, DLI spikes to  $\text{WE} = 1.5$ , which is higher than all other variants. This indicates DLI visits more buckets with no relevant candidates, suggesting issues with ordering buckets from different levels.

#### A.6 Arity and Buffer Size Experiments

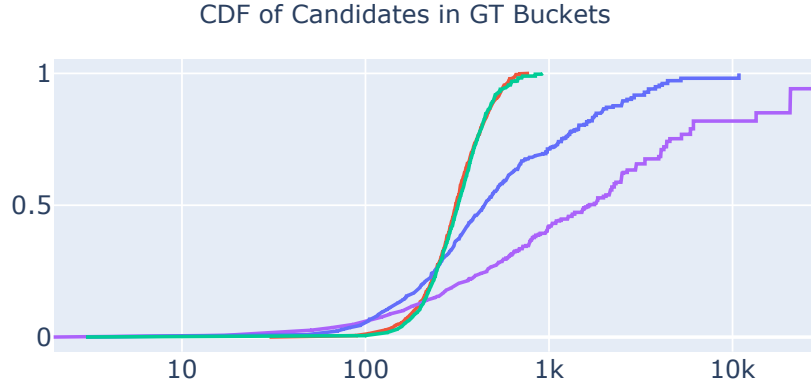
Following the experiments in Section 4.3.3, we perform a detailed analysis of the observed metrics across the whole dynamic experiment.

Figure A.3 shows how recall and total cost per batch changes with increasing dataset size.

1. Because training uses k-means, balancing is not guaranteed



(a) GT Buckets in LMI vs DLI



(b) GT Candidates in LMI vs DLI (log scale)

**Figure A.1:** Partitioning quality metrics comparing LMI and DLI. (a) GT Buckets show similar distribution across all variants, with ground truth neighbors scattered across 7-12 buckets per query. (b) GT Candidates CDF shows DLI Snapshot matches LMI Baseline performance and significantly outperforms both shift scenarios. DLI eliminates bucket overpopulation from distribution shift, requiring fewer candidates to find ground truth neighbors despite a similar bucket spread across levels.

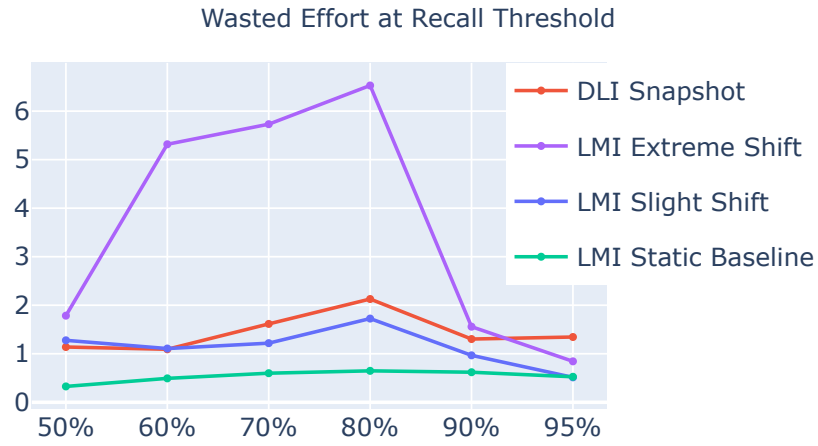
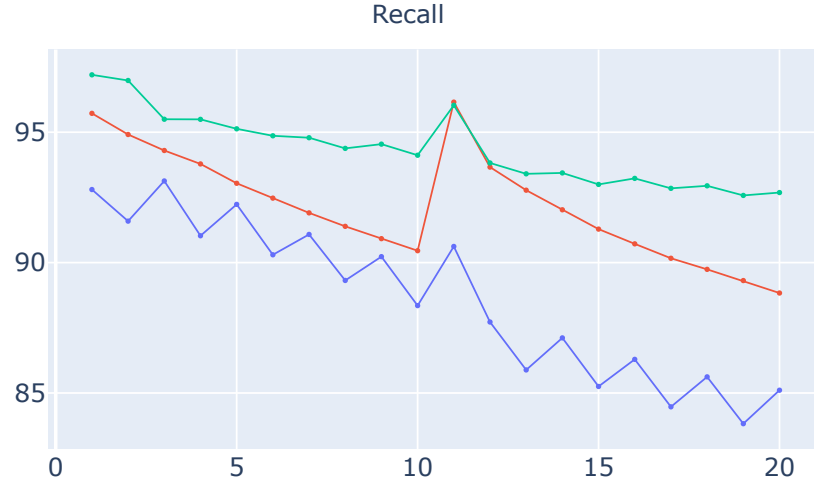
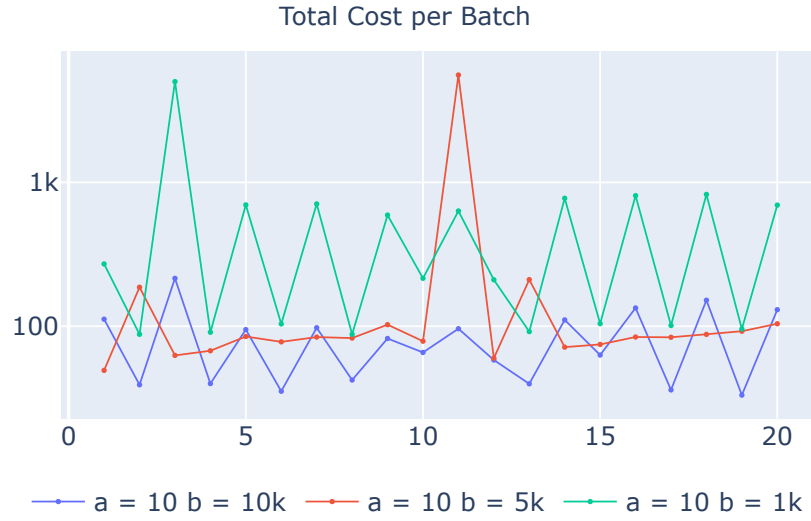


**Figure A.2:** Wasted Effort comparison for LMI and DLI. DLI Snapshot performs similarly to Slight Shift for recall below 90%, maintaining WE within 1.0-2.0. At 95% recall, DLI shows a spike to  $WE = 1.5$ , the highest among all variants, suggesting routing inefficiency from cross-level navigation. Extreme Shift exhibits the worst mid-range performance ( $WE = 6.5$  at 70%), while LMI Baseline maintains near-optimal routing throughout.



(a) Recall



(b) Cost of batch in seconds (logarithmic scale)

**Figure A.3:** Recall and Cost across batches for  $a = 10$  configurations. The  $a = 10$   $b = 1k$  configuration achieves the highest recall but at a much higher cost, while  $a = 10$   $b = 10k$  has the lowest cost but also the lowest recall. The  $a = 10$   $b = 5k$  configuration provides a good balance between recall and cost, maintaining high recall with a reasonable cost across all batches.

The  $b = 10K$  configuration not only starts with the lowest recall but also shows the steepest decreasing trend, due to infrequent compactions.

The  $b = 1K$  configuration adapts best to increasing dataset size in terms of recall, due to the frequent retraining of smaller levels. This is reflected in the total cost spikes, which indicate batches with expensive retraining, with higher spikes corresponding to retraining at larger levels.

The  $b = 5K$  configuration shows more stable costs, with most batches remaining in the lower range of  $b = 1K$ , while maintaining higher recall with less variance. The only outlier is batch 11, which has a cost comparable to  $b = 1K$  due to the creation of a new level. However, this cost is justified by a significant recall improvement that propagates into subsequent batches.

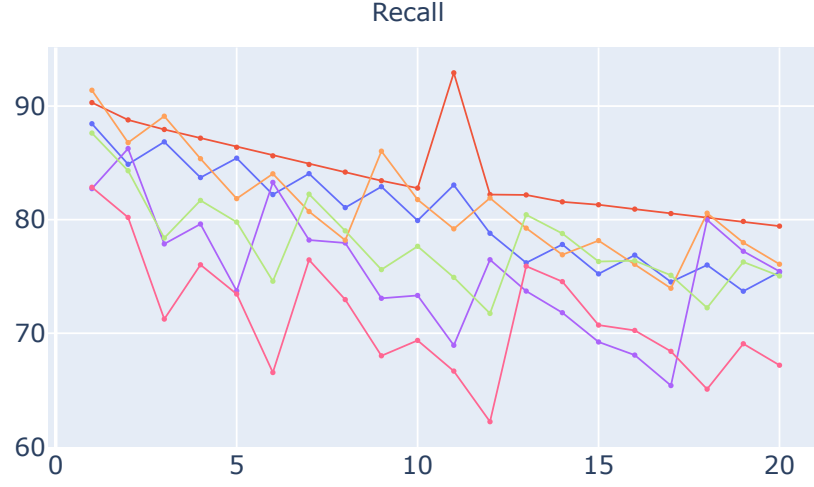
Overall, the  $a = 10, b = 5K$  configuration provides the best balance between recall and cost.

### 100M Dataset

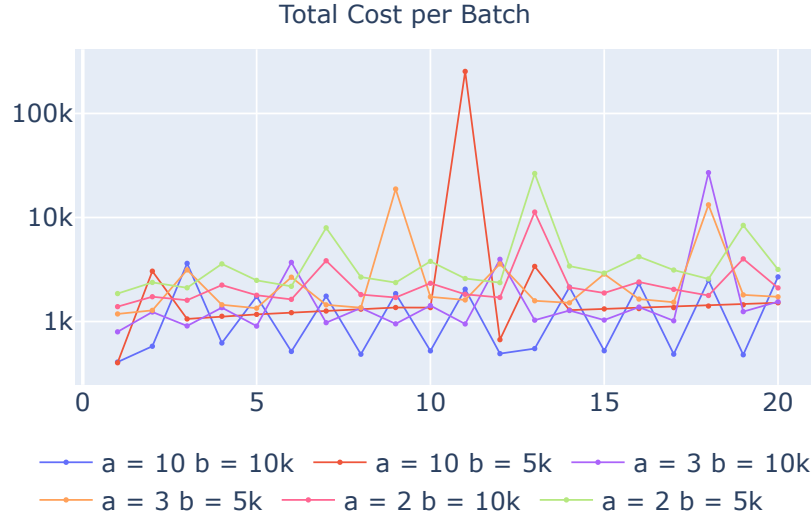
We also evaluated arity and buffer size combinations on the larger LAION2B-100M dataset, excluding the  $b = 1K$  setting due to computational cost. Table A.5 shows the general trends from the 10M dataset hold. However, the high cost of  $a = 10, b = 5K$  does not translate to sustained recall improvements. Although in Figure A.4 we observe a significant recall increase to approximately 92% in the batch with the highest cost (batch 11), recall drops back to the previous range in subsequent batches.

In contrast,  $a = 10, b = 10K$  shows recall within 1-5% of  $b = 5K$  while generally maintaining lower costs throughout. This suggests that for larger datasets,  $a = 10, b = 10K$  provides a better balance between recall and cost, while  $a = 10, b = 5K$  becomes too costly to justify its recall improvement.

To summarize, we conclude that the Model Driven Normalized (MDN\_SS) search strategy and Bentley-Saxe (BS\_C) compaction strategy are the optimal choices for DLI. For arity and buffer size,  $a = 10, b = 5K$  provides good recall across most scenarios. However, if we can predict the expected range of the index size, for datasets exceeding



(a) Recall



(b) Cost of batch in seconds (logarithmic scale)

**Figure A.4:** Recall and cost evolution across batches for  $a = 10$  configurations on the 100M dataset. The  $b = 5K$  configuration achieves the highest peak recall but with unsustainable cost and unstable performance. The  $b = 10K$  configuration maintains comparable recall with the lowest cost among all configurations, making it the optimal choice for large-scale datasets.

**Table A.5:** Arity and buffer size experiments — 100M dataset.

| ID             | Avg Cost/<br>Batch (s) | Avg IPS | Avg QPS | Avg Recall |
|----------------|------------------------|---------|---------|------------|
| a = 10 b = 10k | 1315.59                | 6436.68 | 57.48   | 80.35      |
| a = 10 b = 5k  | 14035.15               | 4057.29 | 47.42   | 84.13      |
| a = 3 b = 10k  | 2700.82                | 4007.90 | 64.09   | 75.62      |
| a = 3 b = 5k   | 3272.63                | 2707.51 | 50.51   | 81.27      |
| a = 2 b = 10k  | 2560.05                | 2447.33 | 56.33   | 71.36      |
| a = 2 b = 5k   | 4529.50                | 1656.54 | 45.63   | 77.91      |

100M vectors, the buffer size should be scaled to  $b = 10K$  for better efficiency.

## A.7 Delete Experiments

From the delete experiments summarised in Table 4.5, we see some general trends across the strategies. However, the variance of the metrics is too small to choose the best strategy based on the average values. By analyzing how the key metrics change across the batches, we can make three key observations.

First, **Rebuild cost is always justified by improvement in recall.** This can be clearly seen in the Figure A.5, where the stable dataset size makes it easier to isolate patterns in performance. Each batch where DPS drops down also shows an upward spike in recall, which stays equal to or higher than No Rebuild across the whole experiment. Although this finding seems obvious, it is important to confirm that rebuild is a beneficial operation. However, we need to note that the inserts seem to be also affected by the rebuild, which poses a question of how much throughput we are able to sacrifice for an approximately 1% recall improvement.

Second, **Rebuild can disrupt the compaction process**, especially with frequent inserts. This is most evident in Figure A.7, where we can clearly see how both rebuild strategies delayed the compaction. In the insert graph A.7b, we see that the insert throughput for No Rebuild drops sharply at batch 10, indicating a compaction event with retraining, while Greedy and Basic maintain high insert throughput until batch 12, when the delete graph A.7c shows a sharp drop in delete throughput, indicating retraining there. The recall A.7a reflects this as well, with No Rebuild spiking up at batch 10, while Greedy and Basic only show a significant recall improvement at batch 12. Interestingly, after batch 12, the recall for Greedy stabilizes at around 1% higher than No Rebuild, which could mean a slight delay in compaction may lead to better results overall.

Third, **Greedy has the same cost as Basic but achieves higher recall.** This is clearest in Heavy Insert (Figure A.7, batch 12), where DPS dips and recall spikes in both, but Greedy achieves 2% higher recall than Basic. This mirrors the compaction findings in Section 4.3.2: merging more levels simultaneously does not increase training cost (because the number of training vectors is capped by the sample threshold) but results in greater structural improvement, reflected in higher recall.

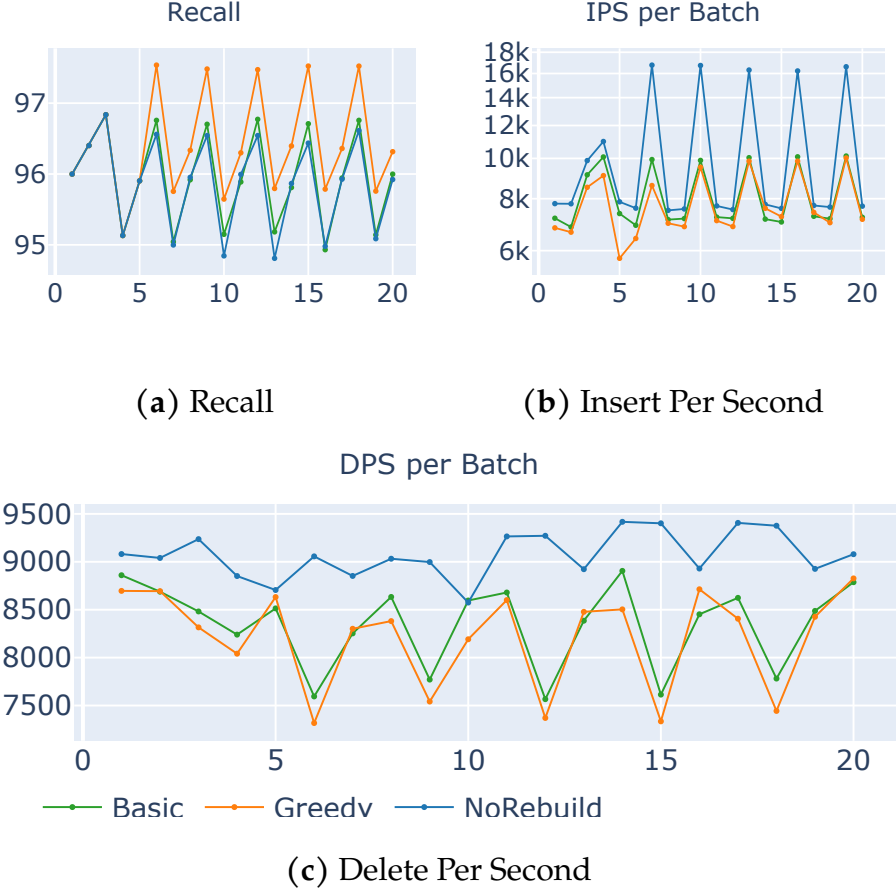


**Figure A.5:** Performance metrics for Constant Insert workload with Random Delete. (a) Recall remains high (95-97%) across all rebuild strategies, but Greedy is generally 1% better. (b) Insert throughput fluctuates between 8K-18K IPS, with No Rebuild achieving the highest peaks. (c) Delete throughput remains consistently high (8.5-9K DPS) for all strategies, with minimal variation, and the drops in Basic and Greedy strategies indicate when rebuild triggers retraining.



**Figure A.6:** Performance metrics for Medium Insert with Random Delete. (a) Recall shows a gradual decline from 96% to 90%, with a spike near batch 20. (b) Insert throughput varies between 5K-15K IPS, with sharp drops indicating that the batch included expensive compaction and retraining. (c) Delete throughput remains stable around 5K-7K DPS through batch 17, then drops sharply for Basic and Greedy, suggesting they triggered retraining, but No Rebuild stayed high. In general, the strategies show the same trends in all metrics, with the curves being shifted on the x-axis when Greedy or Basic trigger retraining sooner than No Rebuild.



**Figure A.7:** Performance metrics for Heavy Insert with Random Delete. (a) Recall curves are nearly identical across all strategies up to batch 10, then diverge when No Rebuild experiences compaction sooner. However, the delayed increase in recall for Greedy propagates better into subsequent batches. (b) The IPS shows the same trend, with all strategies being nearly identical up to batch 11. (c) Delete throughput remains stable around 5K-8K DPS, with a dip at batch 12, to around 75 DPS, where Greedy and Basic trigger rebuild. The Greedy rebuild results in 2% higher recall improvement compared to Basic despite identical computational cost.

### A.7.1 Early Stopping and Epochs

In the subsequent experiments, we continue with Full and From Scratch retraining while varying the number of epochs (10, 20, 30) and enabling early stopping for the 20- and 30-epoch variants. This experiment compares From Scratch and Full retraining in a more flexible training setting to determine whether reusing a pretrained routing model provides benefits.

Table A.6 shows that for the uniform scenario, recall is identical across all variants, as expected from previous experiments. However, IPS decreases with more training epochs, indicating that early stopping does not work effectively.

This is confirmed in the OODQ variant, where From Scratch with 30 epochs achieves lower recall than 20 epochs, indicating overfitting. Interestingly, Full with 30 epochs outperforms all other configurations in the OODQ setting, a result not replicated in any other scenario.

Based on these results, we conclude that the default training configuration with From Scratch retraining is optimal in most scenarios. The scenarios with MQ and OOD queries could point to future research directions, but since we cannot know the query distribution in advance, we prioritize the configuration that performs best for the ID scenarios.

**Table A.6:** From Scratch and Full Retrain experiments.

| ID                | DD | QD   | IPS     | QPS   | Avg Recall |
|-------------------|----|------|---------|-------|------------|
| From Scratch - 10 | SD | IDQ  | 4447.30 | 49.67 | 78.25      |
| From Scratch - 20 | SD | IDQ  | 2940.30 | 30.89 | 78.22      |
| From Scratch - 30 | SD | IDQ  | 3123.56 | 39.79 | 78.17      |
| Full - 20         | SD | IDQ  | 2879.84 | 29.44 | 77.67      |
| Full - 30         | SD | IDQ  | 4080.27 | 48.69 | 77.67      |
| From Scratch - 10 | SD | MQ   | 3475.94 | 34.54 | 68.78      |
| From Scratch - 20 | SD | MQ   | 2794.09 | 23.99 | 69.66      |
| From Scratch - 30 | SD | MQ   | 3662.44 | 45.25 | 71.59      |
| Full - 20         | SD | MQ   | 3887.26 | 52.27 | 70.20      |
| Full - 30         | SD | MQ   | 3466.82 | 51.23 | 71.02      |
| From Scratch - 10 | SD | OODQ | 4353.77 | 41.71 | 53.63      |
| From Scratch - 20 | SD | OODQ | 3588.41 | 41.28 | 57.58      |
| From Scratch - 30 | SD | OODQ | 2840.75 | 32.09 | 53.29      |
| Full - 20         | SD | OODQ | 3490.97 | 38.28 | 53.65      |
| Full - 30         | SD | OODQ | 2940.95 | 28.09 | 59.78      |
| From Scratch - 10 | UD | IDQ  | 5380.03 | 52.01 | 92.18      |
| From Scratch - 20 | UD | IDQ  | 4511.24 | 58.99 | 92.18      |
| From Scratch - 30 | UD | IDQ  | 3830.27 | 56.45 | 92.26      |
| Full - 20         | UD | IDQ  | 4929.91 | 59.79 | 92.23      |
| Full - 30         | UD | IDQ  | 3996.75 | 41.10 | 92.21      |

## Bibliography

1. SMEULDERS, A.W.M.; WORRING, M.; SANTINI, S.; GUPTA, A.; JAIN, R. Content-based image retrieval at the end of the early years. *IEEE Transactions on Pattern Analysis and Machine Intelligence*. 2000, vol. 22, no. 12, pp. 1349–1380. Available from doi: 10.1109/34.895972.
2. SARWAR, Badrul; KARYPIS, George; KONSTAN, Joseph; RIEDL, John. Item-based collaborative filtering recommendation algorithms. In: *Proceedings of the 10th International Conference on World Wide Web*. Hong Kong, Hong Kong: Association for Computing Machinery, 2001, pp. 285–295. WWW '01. ISBN 1581133480. Available from doi: 10.1145/371920.372071.
3. PROCHÁZKA, David; SLANINÁKOVÁ, Terézia; OLHA, Jaroslav; ROŠINEC, Adrián; GREŠOVÁ, Katarína; JÁNOŠOVÁ, Miriama; ČILLÍK, Jakub; PORUBSKÁ, Jana; SVOBODOVÁ, Radka; DOHNAL, Vlastislav; ANTOL, Matej. AlphaFind: discover structure similarity across the proteome in AlphaFold DB. *Nucleic Acids Research*. 2024, vol. 52, no. W1, W182–W186. ISSN 0305-1048. Available from doi: 10.1093/nar/gkae397.
4. SLANINÁKOVÁ, Terézia; ROŠINEC, Adrián; ČILLÍK, Jakub; KŘENEK, Aleš; GREŠOVÁ, Katarína; PORUBSKÁ, Jana; MARŠÁLKOVÁ, Eva; OLHA, Jaroslav; PROCHÁZKA, David; HEJTMÁNEK, Lukáš; DOHNAL, Vlastislav; BERKA, Karel; SVOBODOVÁ, Radka; ANTOL, Matej. *AlphaFind v2: Similarity Search in AlphaFold DB and TED Domains across Structural Contexts - data*. 2026. Available from doi: <https://doi.org/10.58074/mv1wy227>.
5. LEWIS, Patrick; PEREZ, Ethan; PIKTUS, Aleksandra; PETRONI, Fabio; KARPUKHIN, Vladimir; GOYAL, Naman; KÜTTLER, Heinrich; LEWIS, Mike; YIH, Wen-tau; ROCKTÄSCHEL, Tim; RIEDEL, Sebastian; KIELA, Douwe. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. *CoRR*. 2020, vol. abs/2005.11401. Available from arXiv: 2005.11401.

6. AUMÜLLER, Martin; BERNHARDSSON, Erik; FAITHFULL, Alexander. ANN-Benchmarks: A Benchmarking Tool for Approximate Nearest Neighbor Algorithms. In: BEECKS, Christian; BORUTTA, Felix; KRÖGER, Peer; SEIDL, Thomas (eds.). *Similarity Search and Applications*. Cham: Springer International Publishing, 2017, pp. 34–49. ISBN 978-3-319-68474-1.
7. JÄÄSAARI, Elias; HYVÖNEN, Ville; CECCARELLO, Matteo; ROOS, Teemu; AUMÜLLER, Martin. VIBE: Vector Index Benchmark for Embeddings. *ArXiv*. 2025, vol. abs/2505.17810. Available also from: <https://api.semanticscholar.org/CorpusID:278886246>.
8. ANTOL, Matej; OL’HA, Jaroslav; SLANINÁKOVÁ, Terézia; DOHNAL, Vlastislav. Learned Metric Index — Proposition of learned indexing for unstructured data. *Information Systems*. 2021, vol. 100, p. 101774. ISSN 0306-4379. Available from DOI: <https://doi.org/10.1016/j.is.2021.101774>.
9. SLANINÁKOVÁ, Terézia; ANTOL, Matej; OUNDEFINEDHA, Jaroslav; KAŇA, Vojtěch; DOHNAL, Vlastislav. Data-Driven Learned Metric Index: An Unsupervised Approach. In: *Similarity Search and Applications: 14th International Conference, SISAP 2021, Dortmund, Germany, September 29 – October 1, 2021, Proceedings*. Dortmund, Germany: Springer-Verlag, 2021, pp. 81–94. ISBN 978-3-030-89656-0. Available from DOI: 10.1007/978-3-030-89657-7\_7.
10. GUPTA, Gaurav; MEDINI, Tharun; SHRIVASTAVA, Anshumali; SMOLA, Alexander J. BLISS: A Billion scale Index using Iterative Re-partitioning. In: *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. Washington DC, USA: Association for Computing Machinery, 2022, pp. 486–495. KDD ’22. ISBN 9781450393850. Available from DOI: 10.1145/3534678.3539414.
11. LI, Lingli; HAN, Ao; CUI, Xiaotong; WU, Baohua. FLEX: A fast and light-weight learned index for kNN search in high-dimensional space. *Information Sciences*. 2024, vol. 669, p. 120546. ISSN 0020-0255. Available from DOI: <https://doi.org/10.1016/j.ins.2024.120546>.

12. LI, Wuchao; FENG, Chao; LIAN, Defu; XIE, Yuxin; LIU, Haifeng; GE, Yong; CHEN, Enhong. Learning Balanced Tree Indexes for Large-Scale Vector Retrieval. In: *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. Long Beach, CA, USA: Association for Computing Machinery, 2023, pp. 1353–1362. KDD '23. ISBN 9798400701030. Available from doi: 10.1145/3580305.3599406.
13. DONG, Yihe; INDYK, Piotr; RAZENSHTEYN, Ilya P.; WAGNER, Tal. Learning Sublinear-Time Indexing for Nearest Neighbor Search. *ArXiv*. 2019, vol. abs/1901.08544. Available also from: <https://api.semanticscholar.org/CorpusID:59222824>.
14. PROCHÁZKA, David; SLANINÁKOVÁ, Terézia; ČERŇANSKÝ, Jozef; OLHA, Jaroslav; ANTOL, Matej; DOHNAL, Vlastislav. Scaling Learned Metric Index to 100M Datasets. In: *Similarity Search and Applications: 17th International Conference, SISAP 2024, Providence, RI, USA, November 4–6, 2024, Proceedings*. Providence, RI, USA: Springer-Verlag, 2024, pp. 266–273. ISBN 978-3-031-75822-5. Available from doi: 10.1007/978-3-031-75823-2\_22.
15. SLANINÁKOVÁ, Terézia; OLHA, Jaroslav; PROCHÁZKA, David; ANTOL, Matej; DOHNAL, Vlastislav. *On the Costs and Benefits of Learned Indexing for Dynamic High-Dimensional Data: Extended Version*. 2026. Available from arXiv: 2507.05865 [cs.IR].
16. BENTLEY, Jon Louis; SAXE, James B. Decomposable searching problems I. Static-to-dynamic transformation. *Journal of Algorithms*. 1980, vol. 1, no. 4, pp. 301–358. ISSN 0196-6774. Available from doi: [https://doi.org/10.1016/0196-6774\(80\)90015-2](https://doi.org/10.1016/0196-6774(80)90015-2).
17. MALKOV, Yu A.; YASHUNIN, D. A. Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*. 2020, vol. 42, no. 4, pp. 824–836. Available from doi: 10.1109/TPAMI.2018.2889473.
18. DOUZE, Matthijs; GUZHVA, Alexandr; DENG, Chengqi; JOHNSON, Jeff; SZILVASY, Gergely; MAZARÉ, Pierre-Emmanuel; LOMELI, Maria; HOSSEINI, Lucas; JÉGOU, Hervé. *The Faiss library*. 2025. Available from arXiv: 2401.08281 [cs.LG].

19. HAMBARDE, Kailash A.; PROENÇA, Hugo. Information Retrieval: Recent Advances and Beyond. *IEEE Access*. 2023, vol. 11, pp. 76581–76604. Available from doi: 10.1109/ACCESS.2023.3295776.
20. ZEZULA, Pavel; AMATO, Giuseppe; DOHNAL, Vlastislav; BATKO, Michal. *Similarity Search - The Metric Space Approach*. Vol. 32. 2006. ISBN 978-0-387-29146-8. Available from doi: 10.1007/0-387-29151-2.
21. CHÁVEZ, Edgar; NAVARRO, Gonzalo; BAEZA-YATES, Ricardo; MARROQUÍN, José Luis. Searching in metric spaces. *ACM Comput. Surv.* 2001, vol. 33, no. 3, pp. 273–321. ISSN 0360-0300. Available from doi: 10.1145/502807.502808.
22. KELLEY, John L.; YORK, N.; TORONTO, Cincinnati; MELBOURNE, London. General Topology D. Van Nostrand Company. *Mathematical Gazette*. 1957. Available from doi: 10.2307/3610641.
23. LOWE, D.G. Object recognition from local scale-invariant features. In: *Proceedings of the Seventh IEEE International Conference on Computer Vision*. 1999, vol. 2, 1150–1157 vol.2. Available from doi: 10.1109/ICCV.1999.790410.
24. LEVENSHTAIN, Vladimir I. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet physics. Doklady*. 1965, vol. 10, pp. 707–710. Available also from: <https://api.semanticscholar.org/CorpusID:60827152>.
25. BENGIO, Yoshua; COURVILLE, Aaron; VINCENT, Pascal. Representation Learning: A Review and New Perspectives. *IEEE Trans. Pattern Anal. Mach. Intell.* 2013, vol. 35, no. 8, pp. 1798–1828. ISSN 0162-8828. Available from doi: 10.1109/TPAMI.2013.50.
26. KRIZHEVSKY, Alex; SUTSKEVER, Ilya; HINTON, Geoffrey E. ImageNet classification with deep convolutional neural networks. *Commun. ACM*. 2017, vol. 60, no. 6, pp. 84–90. ISSN 0001-0782. Available from doi: 10.1145/3065386.
27. LECUN, Yann; HINTON, Geoffrey. Deep Learning. *Nature*. 2015, vol. 521, pp. 436–44. Available from doi: 10.1038/nature14539.

28. MIKOLOV, Tomas; SUTSKEVER, Ilya; CHEN, Kai; CORRADO, Greg; DEAN, Jeffrey. Distributed representations of words and phrases and their compositionality. In: *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*. Lake Tahoe, Nevada: Curran Associates Inc., 2013, pp. 3111–3119. NIPS'13.
29. BENTLEY, Jon Louis. Multidimensional binary search trees used for associative searching. *Commun. ACM*. 1975, vol. 18, no. 9, pp. 509–517. ISSN 0001-0782. Available from DOI: 10.1145/361002.361007.
30. GUTTMAN, Antonin. R-trees: a dynamic index structure for spatial searching. In: *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*. Boston, Massachusetts: Association for Computing Machinery, 1984, pp. 47–57. SIGMOD '84. ISBN 0897911288. Available from DOI: 10.1145/602259.602266.
31. SHAFT, Uri; RAMAKRISHNAN, Raghu. Theory of nearest neighbors indexability. 2006, vol. 31, no. 3, pp. 814–838. ISSN 0362-5915. Available from DOI: 10.1145/1166074.1166077.
32. BEYER, Kevin; GOLDSTEIN, Jonathan; RAMAKRISHNAN, Raghu; SHAFT, Uri. When Is “Nearest Neighbor” Meaningful? In: BEERI, Catriel; BUNEMAN, Peter (eds.). *Database Theory — ICDT'99*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 217–235. ISBN 978-3-540-49257-3.
33. INDYK, Piotr; MOTWANI, Rajeev. Approximate nearest neighbors: towards removing the curse of dimensionality. In: *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*. Dallas, Texas, USA: Association for Computing Machinery, 1998, pp. 604–613. STOC '98. ISBN 0897919629. Available from DOI: 10.1145/276698.276876.
34. WANG, Mengzhao; XU, Xiaoliang; YUE, Qiang; WANG, Yuxiang. A comprehensive survey and experimental comparison of graph-based approximate nearest neighbor search. *Proc. VLDB Endow.* 2021, vol. 14, no. 11, pp. 1964–1978. ISSN 2150-8097. Available from DOI: 10.14778/3476249.3476255.

35. CHEN, Lu; GAO, Yunjun; ZHENG, Baihua; JENSEN, Christian S.; YANG, Hanyu; YANG, Keyu. Pivot-based metric indexing. *Proc. VLDB Endow.* 2017, vol. 10, no. 10, pp. 1058–1069. ISSN 2150-8097. Available from DOI: 10.14778/3115404.3115411.
36. WANG, Jun; LIU, Wei; KUMAR, Sanjiv; CHANG, Shih-Fu. Learning to Hash for Indexing Big Data—A Survey. *Proceedings of the IEEE.* 2016, vol. 104, no. 1, pp. 34–57. Available from DOI: 10.1109/JPROC.2015.2487976.
37. ANDONI, Alexandr; RAZENSHTEYN, Ilya. Optimal Data-Dependent Hashing for Approximate Near Neighbors. In: *Proceedings of the Forty-Seventh Annual ACM Symposium on Theory of Computing*. Portland, Oregon, USA: Association for Computing Machinery, 2015, pp. 793–801. STOC '15. ISBN 9781450335362. Available from DOI: 10.1145/2746539.2746553.
38. FU, Cong; XIANG, Chao; WANG, Changxu; CAI, Deng. Fast approximate nearest neighbor search with the navigating spreading-out graph. *Proc. VLDB Endow.* 2019, vol. 12, no. 5, pp. 461–474. ISSN 2150-8097. Available from DOI: 10.14778/3303753.3303754.
39. SUBRAMANYA, Suhas Jayaram; DEVVRIT; KADEKODI, Rohan; KRISHASWAMY, Ravishankar; SIMHADRI, Harsha Vardhan. DiskANN: fast accurate billion-point nearest neighbor search on a single node. In: *Proceedings of the 33rd International Conference on Neural Information Processing Systems*. Red Hook, NY, USA: Curran Associates Inc., 2019.
40. SINGH, Aditi; SUBRAMANYA, Suhas Jayaram; KRISHNASWAMY, Ravishankar; SIMHADRI, Harsha Vardhan. *FreshDiskANN: A Fast and Accurate Graph-Based ANN Index for Streaming Similarity Search*. 2021. Available from arXiv: 2105.09613 [cs.IR].
41. XIAO, Wentao; ZHAN, Yueyang; XI, Rui; HOU, Mengshu; LIAO, Jianming. *Enhancing HNSW Index for Real-Time Updates: Addressing Unreachable Points and Performance Degradation*. 2024. Available from arXiv: 2407.07871 [cs.IR].

42. CIACCIA, Paolo; PATELLA, Marco; ZEZULA, Pavel. M-tree: An Efficient Access Method for Similarity Search in Metric Spaces. In: *Proceedings of the 23rd International Conference on Very Large Data Bases*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997, pp. 426–435. VLDB '97. ISBN 1558604707.
43. DOHNAL, Vlastislav; GENNARO, Claudio; SAVINO, Pasquale; ZEZULA, Pavel. D-Index: Distance Searching Index for Metric Data Sets. *Multimedia Tools Appl.* 2003, vol. 21, no. 1, pp. 9–33. ISSN 1380-7501. Available from DOI: 10.1023/A:1025026030880.
44. NOVAK, David; BATKO, Michal; ZEZULA, Pavel. Metric Index: An efficient and scalable solution for precise and approximate similarity search. *Inf. Syst.* 2011, vol. 36, no. 4, pp. 721–733. ISSN 0306-4379. Available from DOI: 10.1016/j.is.2010.10.002.
45. YIANILOS, Peter N. Data structures and algorithms for nearest neighbor search in general metric spaces. In: *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*. Austin, Texas, USA: Society for Industrial and Applied Mathematics, 1993, pp. 311–321. SODA '93. ISBN 0898713137.
46. DING, Chris; HE, Xiaofeng. K-nearest-neighbor consistency in data clustering: incorporating local information into global optimization. In: *Proceedings of the 2004 ACM Symposium on Applied Computing*. Nicosia, Cyprus: Association for Computing Machinery, 2004, pp. 584–589. SAC '04. ISBN 1581138121. Available from DOI: 10.1145/967900.968021.
47. RUIZ, Guillermo; SANTOYO, Francisco; CHÁVEZ, Edgar; FIGUEROA, Karina; TELLEZ, Eric Sadit. Extreme Pivots for Faster Metric Indexes. In: *Proceedings of the 6th International Conference on Similarity Search and Applications - Volume 8199*. A Coruña, Spain: Springer-Verlag, 2013, pp. 115–126. SISAP 2013. ISBN 9783642410611. Available from DOI: 10.1007/978-3-642-41062-8\_12.
48. MACQUEEN, J. Some methods for classification and analysis of multivariate observations. In: 1967. Available also from: <https://api.semanticscholar.org/CorpusID:6278891>.

49. KRASKA, Tim; BEUTEL, Alex; CHI, Ed H.; DEAN, Jeffrey; POLY-ZOTIS, Neoklis. The Case for Learned Index Structures. In: *Proceedings of the 2018 International Conference on Management of Data*. Houston, TX, USA: Association for Computing Machinery, 2018, pp. 489–504. SIGMOD '18. ISBN 9781450347037. Available from doi: 10.1145/3183713.3196909.
50. AL-MAMUN, Abdullah; WU, Hao; HE, Qiyang; WANG, Jianguo; AREF, Walid G. *A Survey of Learned Indexes for the Multi-dimensional Space*. 2024. Available from arXiv: 2403.06456 [cs.DB].
51. TIAN, Yao; YAN, Tingyun; ZHAO, Xi; HUANG, Kai; ZHOU, Xiaofang. A Learned Index for Exact Similarity Search in Metric Spaces. *IEEE Trans. on Knowl. and Data Eng.* 2023, vol. 35, no. 8, pp. 7624–7638. ISSN 1041-4347. Available from doi: 10.1109/TKDE.2022.3206441.
52. LI, Mingjie; WANG, Yuan-Gen; ZHANG, Peng; WANG, Hanpin; FAN, Lisheng; LI, Enxia; WANG, Wei. Deep Learning for Approximate Nearest Neighbour Search: A Survey and Future Directions. 2023, vol. 35, no. 9, pp. 8997–9018. ISSN 1041-4347. Available from doi: 10.1109/TKDE.2022.3220683.
53. MORENO-TORRES, Jose G.; RAEDER, Troy; ALAIZ-RODRÍGUEZ, Rocío; CHAWLA, Nitesh V.; HERRERA, Francisco. A unifying view on dataset shift in classification. *Pattern Recognition*. 2012, vol. 45, no. 1, pp. 521–530. ISSN 0031-3203. Available from doi: <https://doi.org/10.1016/j.patcog.2011.06.019>.
54. ZEIGHAMI, Sepanta; SHAHABI, Cyrus. Theoretical Analysis of Learned Database Operations under Distribution Shift through Distribution Learnability. *Proceedings of machine learning research*. 2024, vol. 235, pp. 58283–58305. Available also from: <https://api.semanticscholar.org/CorpusID:272330617>.
55. VEN, Gido M. van de; SOURES, Nicholas; KUDITHIPUDI, Dhireesha. Continual learning and catastrophic forgetting. In: *Learning and Memory: A Comprehensive Reference*. Elsevier, 2025, pp. 153–168. ISBN 9780443157554. Available from doi: 10.1016/b978-0-443-15754-7.00073-0.

56. KIRKPATRICK, James; PASCANU, Razvan; RABINOWITZ, Neil; VENESS, Joel; DESJARDINS, Guillaume; RUSU, Andrei A.; MILAN, Kieran; QUAN, John; RAMALHO, Tiago; GRABSKA-BARWINSKA, Agnieszka; HASSABIS, Demis; CLOPATH, Claudia; KUMARAN, Dharshan; HADSELL, Raia. Overcoming catastrophic forgetting in neural networks. *Proceedings of the National Academy of Sciences*. 2017, vol. 114, no. 13, pp. 3521–3526. Available from doi: 10.1073/pnas.1611835114.
57. DING, Jialin; MINHAS, Umar Farooq; YU, Jia; WANG, Chi; DO, Jaeyoung; LI, Yanan; ZHANG, Hantian; CHANDRAMOULI, Badrish; GEHRKE, Johannes; KOSSMANN, Donald; LOMET, David; KRASKA, Tim. ALEX: An Updatable Adaptive Learned Index. In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. Portland, OR, USA: Association for Computing Machinery, 2020, pp. 969–984. SIGMOD '20. ISBN 9781450367356. Available from doi: 10.1145/3318464.3389711.
58. O'NEIL, Patrick; CHENG, Edward; GAWLICK, Dieter; O'NEIL, Elizabeth. The log-structured merge-tree (LSM-tree). *Acta Inf.* 1996, vol. 33, no. 4, pp. 351–385. ISSN 0001-5903. Available from doi: 10.1007/s002360050048.
59. LUO, Chen; CAREY, Michael J. LSM-based storage techniques: a survey. *The VLDB Journal*. 2019, vol. 29, no. 1, pp. 393–418. ISSN 1066-8888. Available from doi: 10.1007/s00778-019-00555-y.
60. FACEBOOK RESEARCH. FAISS FAQ: Questions about training [<https://github.com/facebookresearch/faiss/wiki/FAQ>]. 2024. Accessed: 2024.
61. PRECHELT, Lutz. Early Stopping - But When? In: *Neural Networks: Tricks of the Trade*. Ed. by ORR, Genevieve B.; MÜLLER, Klaus-Robert. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 55–69. ISBN 978-3-540-49430-0. Available from doi: 10.1007/3-540-49430-8\_3.
62. PASZKE, Adam; GROSS, Sam; MASSA, Francisco; LERER, Adam; BRADBURY, James; CHANAN, Gregory; KILLEEN, Trevor; LIN, Zeming; GIMELSHEIN, Natalia; ANTIGA, Luca; DESMAISON, Alban; KÖPF, Andreas; YANG, Edward; DEVITO, Zach; RAISON, Martin; TEJANI, Alykhan; CHILAMKURTHY,

- Sasank; STEINER, Benoit; FANG, Lu; BAI, Junjie; CHINTALA, Soumith. PyTorch: an imperative style, high-performance deep learning library. In: *Proceedings of the 33rd International Conference on Neural Information Processing Systems*. Red Hook, NY, USA: Curran Associates Inc., 2019.
63. SARKAR, Subhadeep; STARATZIS, Dimitris; ZHU, Ziehen; ATHANASSOULIS, Manos. Constructing and analyzing the LSM compaction design space. *Proc. VLDB Endow.* 2021, vol. 14, no. 11, pp. 2216–2229. ISSN 2150-8097. Available from doi: 10.14778/3476249.3476274.
64. SCHUHMANN, Christoph; BEAUMONT, Romain; VENCU, Richard; GORDON, Cade; WIGHTMAN, Ross; CHERTI, Mehdi; COOMBES, Theo; KATTA, Aarush; MULLIS, Clayton; WORTSMAN, Mitchell; SCHRAMOWSKI, Patrick; KUNDURTHY, Srivatsa; CROWSON, Katherine; SCHMIDT, Ludwig; KACZMARCZYK, Robert; JITSEV, Jenia. LAION-5B: an open large-scale dataset for training next generation image-text models. In: *Proceedings of the 36th International Conference on Neural Information Processing Systems*. New Orleans, LA, USA: Curran Associates Inc., 2022. NIPS '22. ISBN 9781713871088.
65. YANDEX, Artem Babenko; LEMPITSKY, Victor. Efficient Indexing of Billion-Scale Datasets of Deep Descriptors. In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016, pp. 2055–2063. Available from doi: 10.1109/CVPR.2016.226.
66. FACEBOOK RESEARCH. *Faiss Indexes: Cell Probe Method with a Flat Index as Coarse Quantizer* [<https://github.com/facebookresearch/faiss/wiki/Faiss-indexes>]. 2024. Accessed: 2026-05-18.