LeaFi: Data Series Indexes on Steroids with Learned Filters

QITONG WANG, LIPADE, Université Paris Cité, France
IOANA ILEANA, LIPADE, Université Paris Cité, France
THEMIS PALPANAS, LIPADE, Université Paris Cité & French University Institute (IUF), France

The ever-growing collections of data series create a pressing need for efficient similarity search, which serves as the backbone for various analytics pipelines. Recent studies have shown that tree-based series indexes excel in many scenarios. However, we observe a significant waste of effort during search, due to suboptimal pruning. To address this issue, we introduce LeaFi, a novel framework that uses machine learning models to boost pruning effectiveness of tree-based data series indexes. These models act as learned filters, which predict tight node-wise distance lower bounds that are used to make pruning decisions, thus, improving pruning effectiveness. We describe the LeaFi-enhanced index building algorithm, which selects leaf nodes and generates training data to insert and train machine learning models, as well as the LeaFi-enhanced search algorithm, which calibrates learned filters at query time to support the user-defined quality target of each query. Our experimental evaluation, using two different tree-based series indexes and five diverse datasets, demonstrates the advantages of the proposed approach. LeaFi-enhanced data-series indexes improve pruning ratio by up to 20x and search time by up to 32x, while maintaining a target recall of 99%.

CCS Concepts: • **Information systems** → **Data management systems**; • **Computing methodologies** → **Machine learning**.

Additional Key Words and Phrases: Data Series, Similarity Search, Index, Machine Learning

ACM Reference Format:
Qitong Wang, Ioana Ileana, and Themis Palpanas. 2025. LeaFi: Data Series Indexes on Steroids with Learned Filters. *Proc. ACM Manag. Data* 3, N1 (SIGMOD), Article 51 (February 2025), 27 pages. https://doi.org/10.1145/3709701

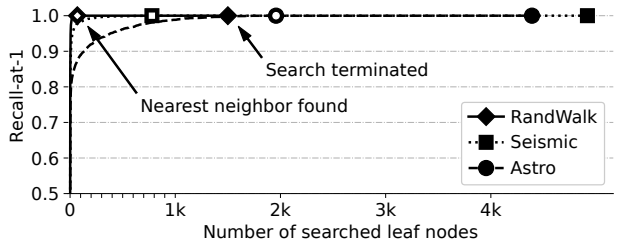
1 Introduction

Background. With the rapid advancements and implementations of modern sensors, there is a significant rise in the generation, collection, and analysis of large datasets consisting of data series across various scientific fields [43]. Common techniques employed in the analysis of data series include classification [19], clustering [45], pattern mining [10], anomaly detection [46], visualization [23], etc. However, as the data series collection grows largely in scale, it becomes crucial to incorporate *similarity search* methods to maintain their efficiency and effectiveness [42]. Data series similarity search is the technique used to identify the most similar series (usually the nearest neighbors) in a dataset, given a query series and a specific similarity measure (usually Euclidean distance [11]).

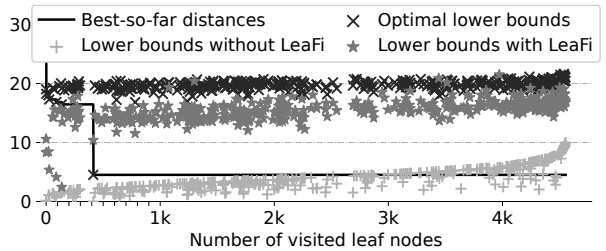
Indexes are widely employed to speed up series similarity search. Among various series indexing techniques, tree-based indexes have demonstrated state-of-the-art (SOTA) performance in numerous

Authors' Contact Information: Qitong Wang, LIPADE, Université Paris Cité, Paris, France, qitong.wang@u-paris.fr; Ioana Ileana, LIPADE, Université Paris Cité, Paris, France, ioana.ileana@parisdescartes.fr; Themis Palpanas, LIPADE, Université Paris Cité & French University Institute (IUF), Paris, France, themis@mi.parisdescartes.fr.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM 2836-6573/2025/2-ART51
https://doi.org/10.1145/3709701



(a) Average recall-at-1 compared to the number of searched DSTree leaf nodes. The search algorithm continues after finding the nearest neighbors.



(b) Using the LeaFi predictions (★) instead of current lower bounds (+) boosts the pruning ratio for an Astro query from 23% to 89.4% (examples above the best-so-far curve can be pruned). Particularly, the pruning enhancement brought by LeaFi is observed both before and after the nearest neighbors are found. The LeaFi filters are trained to predict the optimal lower bounds (×), which can prune 99% leaf nodes.

Fig. 1. A waste of data series search time, caused by insufficient pruning, is observed across various datasets. Employing the LeaFi predictions for the optimal lower bounds, instead of the current summarization-based lower bounds, improves the pruning ratios significantly.

scenarios [15, 16], including in the context of hybrid solutions [4, 66, 69]. A tree-based index is built using lower-dimensional summarizations of data series [55, 65]. Its index structure consists of internal nodes, which determine the order of visiting nodes, and leaf nodes, which store the original series values. Each node uses an aggregated summarized representation of the series it contains, in order to calculate a distance lower bound, which is compared with the best-so-far result to determine if this node can be pruned during query answering.

Motivation. Tree-based indexes greatly enhance the efficiency of series similarity search. However, we argue that there are still significant opportunities for further acceleration. This potential is highlighted in Figure 1a by the large gap in search time[1], between when the nearest neighbor results are found and when they are actually returned [22, 33]. The search algorithm spends this significant amount of extra time trying to verify that there is no other better answer. We attribute

[1]We use the number of searched leaf nodes as a hardware-agnostic surrogate for search time [3]. In our context, *searching* a leaf node means calculating the distances for all series it contains. *Visiting* a node, on the other hand, refers to checking its lower bound to determine whether it can be pruned. A leaf node may be visited during a query, but not searched.

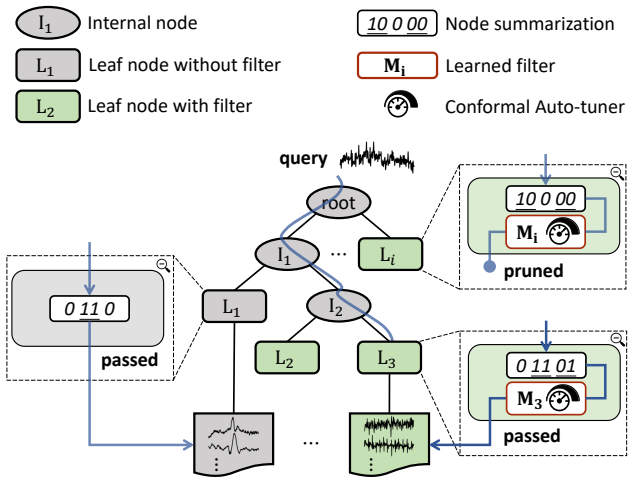


Fig. 2. An illustration of a LeaFi-enhanced tree-based index structure, along with an example of its search procedure.

this inefficiency to the inability of existing tree-based series indexes to provide tight distance lower bounds for effective pruning. Figure 1b reveals that only 23% of the summarization-based lower bounds (cf. Figure 1b, Lower bounds without LeaFi) exceed the best-so-far distances (cf. Figure 1b, black line), resulting in a 23% pruning ratio for a query on the Astro dataset, which is a rather poor performance.

Our key insight to this problem focuses on the introduction of the optimal lower bound for a leaf node, which is the smallest distance between a query and all the series that the leaf node contains; we call this distance the *node-wise nearest neighbor distance*. By leveraging these node-wise distances, 99% of the leaf nodes (up from 23%) can be pruned (cf. Figure 1b, Optimal lower bounds). However, directly calculating these distances by searching each leaf node is impractical. To overcome this challenge, we opt to employ machine learning models to predict the node-wise nearest neighbor distances. These models act as Learned Filters (LeaFi), improving the pruning effectiveness of tree-based series indexes.

***Our solution:* LeaFi.** In this paper, we introduce LeaFi, a novel general framework that introduces machine learning models into tree-based series indexes to improve their pruning capability for search acceleration. We name these machine learning models *learned filters*, and the index that uses them *LeaFi-enhanced index*. In essence, LeaFi carefully places learned filters in a selected subset of leaf nodes, with each filter dedicated to one leaf node. These learned filters are trained to predict the node-wise nearest neighbor distances for a given query, which serve as lower bounds for comparison against the best-so-far distance. Since the predictions are much tighter than the original node summarization-based lower bounds, the leaf node pruning ratios are largely improved. In our example, we go from 23% pruning to 90% pruning (cf. Figure 1b, Lower bounds with LeaFi). Though, this comes at the expense of a slight reduction in accuracy, which is controlled by the user and can be determined at query time (independently for each query). Note that a LeaFi-enhanced index can always provide exact results (guaranteed 100% recall) for a specific query, simply by disabling the filter-based pruning strategy at query time. Our experimental evaluation shows that LeaFi-enhanced indexes achieve a remarkable improvement in pruning ratio (up to 20x more) and search time (up to 32x faster), while maintaining almost perfect accuracy (i.e., 99% recall).

Figure 2 illustrates the structure of a LeaFi-enhanced index. For the selected leaf nodes L_2, L_3 and L_i, learned filters are paired with their node summarizations. When these nodes are visited by a query, a cascade of summarization-based lower bounds and filter predictions are computed to determine whether these nodes should be pruned. To the best of our knowledge, LeaFi is the first framework that incorporates machine learning models into data series indexes for improving pruning during similarity search.

Technical challenges. Incorporating learned filters into tree-based series indexes is not straightforward. The LeaFi workflow unfolds in three main stages. It starts by selecting the subset of leaf nodes for filter insertion. Next, it prepares the training data needed to train these filters. The last step is to calibrate the filters' predictions to better serve as lower bounds. We outline these steps along with their challenges and our solutions as follows.

First, effectively inserting learned filters requires identifying the subset of leaf nodes that offer the most significant reduction in search time. As modern machine learning models are accelerated using Graphics Processing Devices (GPU) [40], indiscriminately adding one learned filter to every leaf node could overwhelm the GPU memory. Typical number of leaf node in tree-based series indexes can reach the range of 100K [67]). Moreover, applying learned filters might not always improve search time. For example, directly searching a small leaf node of 100 series can be faster than predicting a lower bound using a Multilayer Perceptron (MLP) [1] model. To address these issues systematically, we design a general formalization that treats leaf node selection as a knapsack problem [27]. In this analogy, adding a learned filter to a leaf node is considered adding an "item", where its "value" is the expected reduction of this node's search time. Given the constraint of available GPU memory, our goal in this setup is to identify which leaf nodes will yield the highest expected search time savings, when equipped with learned filters. We further show that under certain assumptions, this general formalization can be simplified and solved by a greedy algorithm.

Second, training the inserted filters requires generating appropriate training data. A key challenge is that the majority of the node-wise nearest neighbor distances fall out of the value range of global nearest neighbor distances. Figure 1b, shows that node-wise nearest neighbor distances lie around the value of 20, while the global nearest neighbor distance is 4.9. Traditional training data generation methods, which typically involve random sampling combined with Gaussian noise [13], tend to bias training towards these larger, node-wise distances, neglecting the global distances [63]. To resolve this issue, we propose a novel twofold strategy for training data generation. It consists of generating global training queries that are derived from the entire series collection and applicable to all leaf nodes, alongside node-wise training queries that are derived from and specific to each leaf node. Our twofold approach ensures that the training data encompasses both local (node-wise) and global contexts, facilitating unbiased filter training across all necessary value ranges.

Lastly, learned filters, being machine learning models, cannot ensure consistent prediction quality [32]. This compromises the exactness of the search result in original series indexes [62]. To mitigate the uncertainty of the search quality, we propose *conformal auto-tuners*, a method inspired by conformal regressions [2] to enable support for user-requested search quality (e.g., recall, tightness, etc.) targets. Our conformal auto-tuners calibrate the filter prediction using a learned offset, determined by a certain quality target that can be set at query time, independently for each query. We employ a separate calibration set [44] to simulate search under different offsets and collect the result qualities. The conformal auto-tuners learn the mapping between observed result qualities and corresponding calibration offsets, such that the calibration offset can be dynamically obtained for any given quality target.

Contributions. Our contributions can be summarized as follows.

(1) We introduce the first approach that uses learned filters to improve the pruning effectiveness of tree-based series indexes, and hence accelerate data series similarity search.

(2) We design LeaFi, a novel general framework that effectively integrates learned filters into (different) tree-based series indexes. LeaFi carefully selects an optimal subset of leaf nodes for filter insertion, and generates appropriate training data for filter training.

(3) We propose conformal auto-tuners to mitigate the uncertainty in the results of machine learning models. Conformal auto-tuners calibrate learned filters at query time to support the user-defined quality target, independently for each query (the user may also choose to disable the learned filters).

(4) Our experimental evaluation, using two diverse tree-based series indexes and five diverse datasets, demonstrates the benefits of the proposed approach, and its advantages when compared to alternatives. LeaFi-enhanced series indexes improve pruning ratio by up to 20x and search time by up to 32x, while maintaining a target recall of 99%. Codes and datasets are available online[2].

2 Related Work

Data series indexes. The most prominent data series indexing techniques can be categorized into graph-based indexes [38], quantization [21] and inverted indexes [5], locality-sensitive hashes [24], and tree-based indexes [55, 65]. Recent studies [15, 16] have demonstrated that tree-based indexes [47] achieve SOTA performance under several conditions (e.g., large-scale dataset).

iSAX [55] and DSTree [65] are two SOTA tree-based indexes for series similarity search of different strengths [15, 16]. iSAX is based on Symbolic aggregate approximation (SAX) [55], a discretized series summarization based on piecewise aggregate approximation (PAA) [28]. PAA first transforms the data series into l real values, and then SAX quantizes each PAA value using discrete symbols. iSAX (indexable SAX) [55] enables the comparison of SAXs of different cardinalities, that makes SAX indexable through a prefix trie [7]. DSTree [65] is a dynamic splitting tree based on the adaptive piecewise constant approximation (EAPCA). Furthermore, ADS+ [74] makes iSAX continuously adaptive to queries, ULISSE[36, 37] supports variable-length queries, Coconut [30] delivers a sortable iSAX variant, DPiSAX [71] and Odyssey [8] make iSAX distributed, ParIS [49], MESSI [47, 48] and SING [50] bring in modern hardware, FreSh [18] adds lock-freedom, Dumpy [67] and DumpyOS [68] introduce a data-adaptive multi-ary structure, while Hercules [13] and Elpis [3] combine the iSAX and EAPCA [65] summarizations.

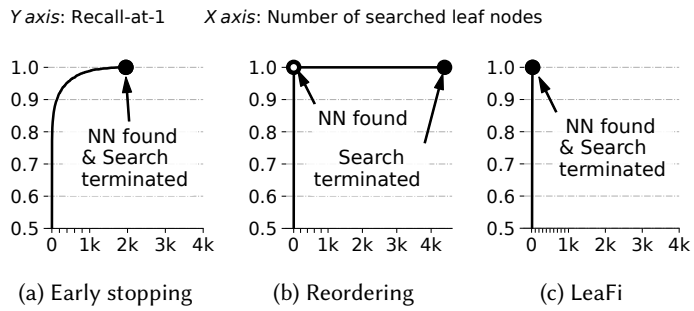
The proposed LeaFi framework is index agnostic. We instantiate and evaluate it on both MESSI [48] and DSTree [65], making its improvements translatable to most tree-based indexes.

Machine learning applications in series indexes. Machine learning techniques have proven effective in enhancing various components of databases [34, 54], such as indexes [12, 31, 35, 39, 62], cardinality estimators [29, 58, 70], etc. A few existing works are also motivated by the fact that there is waste in search time [14, 22]. These works can be divided into two categories, early stopping approaches [14, 16, 22] and leaf node reordering approaches [26].

In the context of data series similarity search, ϵ-search identifies heuristic stopping criteria when best-so-far results are in the ϵ neighborhood of nearest neighbor results [16]. $\delta\epsilon$-search extends ϵ-search by supporting a confidence level δ, based on estimated pairwise distance distribution [16]. Progressive Search (ProS) incorporates machine learning models to estimate when the exact results are retrieved, using the query and best-so-far distances [14, 22]. Learned Reordering (LR) determines the visiting order of the leaf nodes by predicting their probabilities of containing the nearest neighbor results for a given query [26].

Figure 3 illustrates the improvement potential for DSTree index on Astro dataset, as in Figure 1a, of early stopping approaches (ϵ-search, $\delta\epsilon$-search, ProS and FLT), leaf node reordering approaches (LR), and our proposed learned filter approach (LeaFi). These optimal performance is simulated by

[2]https://github.com/qtwang/LeaFi



(a) Early stopping (b) Reordering (c) LeaFi

Fig. 3. The optimal search time that can be possibly achieved by early-stopping approaches [14, 16, 22], leaf node reordering approaches [26] and LeaFi for DSTree index on Astro dataset. The axes are the same as Figure 1b (x-axis in Figure 3c has a different scale).

assuming all machine learning models make no mistake. Leaf node reordering approaches help series indexes find the nearest neighbor results in the first node being visited and employ the tightest best-so-far distance. However, as shown in Figure 1b, a tight best-so-far distance provide marginal help in pruning more leaf nodes - most of the summarization-based lower bounds are still smaller than that. Early stopping approaches can terminate search right after the nearest neighbor results are retrieved, but it cannot reduce the search time before the retrieval.

On the other hand, LeaFi helps series indexes to search only those leaf nodes that can update the best-so-far distances, as the tightest lower bounds are employed in pruning decision, hence attaining a significant and consistent improvement potential. Moreover, as trained using the node-wise distance information instead of index-wise leaf node searching information, LeaFi can be efficiently trained using 2k examples (0.002% of the collection), compared to 100k to 1m examples (0.1% of the collection) in the literature [26, 33]. As far as we are aware, LeaFi represents the first framework that incorporates learned filters in data series indexes and provides substantial improvement in pruning ratio and search time.

FAISS Learned Termination (FLT) is an early-termination technique proposed for vector similarity search using kNN graphs [33]. Its stopping criterion is predicted by a nontrivial expert-crafted feature set, which cannot be directly applied to tree-based series indexes, and, in contrast to LeaFi, it does not offer a mechanism to set search quality targets.

Conformal regressions. Conformal regression is a statistical approach that enhances existing regression methods by providing predictive intervals with a guarantee on their coverage probability [61]. It involves fitting a regression model to a dataset and then generating predictions that include an interval which, with a specified level of confidence (e.g., 95%), is expected to cover the true target values. This process relies on the computation of nonconformity scores that measure how unusual new observations are compared to training data [2, 44].

However, the direct application of existing conformal regression techniques cannot help LeaFi support the user-requested search quality. This is because the conformal prediction intervals are derived independently for each model, whereas achieving an expected target of a LeaFi-enhanced search result, e.g., 99% recall, requires tuning all learned filters collaboratively at the same time. Hence, in LeaFi, we further design our auto-tuning approach based on the conformal regression framework. To the best of our knowledge, LeaFi demonstrates the first effort to introduce conformal regression techniques into the domain of learned indexes.

3 Preliminaries

Data series. A *data series*, $S = \{p_1, ..., p_m\}$, is a sequence of points, where each point $p_i = (v_i, t_i)$, $1 \leq i \leq m$ is associated to a real value v_i and a position t_i. We call m the *length* of the series. $\mathcal{S}$ denotes a collection of data series, i.e., $\mathcal{S} = \{S_1, ..., S_n\}$. We call n the *size* of the series collection.

A *summarization* $E = \{e_1, ..., e_l\}$ of a series S is its lower-dimensional representation, that preserves some desired properties of $\mathcal{S}$. For example, SAX in MESSI [55] and APCA in DSTree [65] are two popular series summarizations. In series similarity search, summarizations can be utilized to calculate lower bounds between a series or a set of series and a query.

Similarity search. Given a query series S_q, a series collection $\mathcal{S}$ of size n, a distance measure d, *similarity search* targets to identify the series $S_c \in \mathcal{S}$ whose distance to S_q is the smallest, i.e., $\forall S_o \in \mathcal{S}, S_o \neq S_c, d(S_c, S_q) \leq d(S_o, S_q)$.

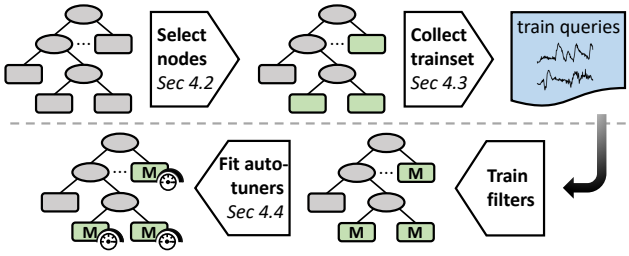
The LeaFi framework works for any distance measure supported by the backbone index, including Euclidean and Dynamic Time Warping (DTW), two popular distances for series similarity search [11].

Tree-based indexes. Tree-based series indexes, including DSTree [65] and MESSI [55], are constituted by *internal nodes* I_is and *leaf nodes* L_is, as shown in Figure 2. We use N_i to denote a node when there is no need to distinguish between it being an internal node or leaf node. Only leaf nodes store the raw series. An internal node routes a series to one of its child node that this series should be inserted into. Both types of nodes contain a *node summarization* E_i^N that aggregates the series summarizations of all series it contains. Node summarizations are used to calculate distance lower bounds for search routing and leaf node pruning.

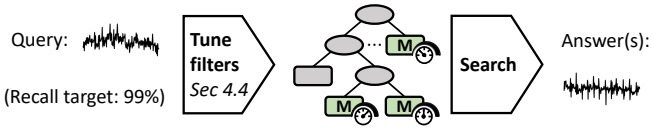
During query answering, internal nodes navigate the query series S_q to visit leaf nodes according to their lower bounds to the query. Only the leaf nodes whose subtree cannot be pruned are visited. We maintain a best-so-far result $d^{\text{bsf}}(S_q, N_i)$, i.e., the smallest distance before visiting a node N_i. Given a node N_i, we first calculate the node summarization-based lower bound $d^{\text{lb}}(S_q, N_i)$ for the distances between S_q and all series N_i contains. When the context is clear, we remove inputs $(\cdot)$ or subscripts $_i$ in the equations for clarity. We then compare d^{lb} with d^{bsf}. $d^{\text{lb}} > d^{\text{bsf}}$ indicates all series in a leaf node L_i or a subtree $I_i$have larger distances to S_q, thus L_i or $I_i$can be safely pruned. Otherwise, there might be a series that has smaller distance to S_q. For L_i, we have to search L_i to get its node-wise nearest neighbor distance $d^{\text{L}}(S_q, L_i) = \min_k d(S_q, S_k), S_k \in L_i$, and check whether d^{L} can update d^{bsf}. For I_i, we repeat the pruning checking for the nodes in its subtree. After all nodes are either visited or pruned, d^{bsf} is returned as the true nearest neighbor result.

Conformal regression. Conformal regression utilizes posterior statistics to auto-tune machine learning predictions to target at a certain confidence level [2]. Suppose we collect a training set $\{(x_1, y_1), (x_2, y_2), ..., (x_l, y_l)\}$ of l examples. In our case, x_i is a training query (i.e., S_q) and y_i is the node-wise nearest neighbor distance d^{N}. Given a confidence level $1 - \delta$, *conformal regression* [2] predicts an interval, instead of a single value, such that the true y_{l+1} of a new example x_{l+1} will be covered by this interval with a $1 - \delta$ confidence.

In LeaFi, we embrace the core designs from the classic *inductive conformal regression* [44] as a proof-of-concept study. Inductive conformal regression splits the training data into two subsets: the *proper training set* $\{(x_1, y_1), (x_2, y_2), ..., (x_m, y_m)\}$ and the *calibration set* $\{(x_{m+1}, y_{m+1}), (x_{m+2}, y_{m+2}), ..., (x_l, y_l)\}$. After training a machine learning model on the proper training set, we calculate and sort the *non-conformity measures* on the calibration set: $\alpha_i := |y_{m+1} - \hat{y}_{m+i}|, i = 1, 2, ..., l - m$, where $\hat{y}_{m+i}$ is the prediction of a sample x_{m+i}. We use $[\alpha_{(1)}, \alpha_{(2)}, ..., \alpha_{(l-m)}]$ to denote the descending-ordered αs on the calibration set. Let $j_s = |\{\alpha_i : \alpha_i \geq \alpha_{(s)}\}|, s = 1, 2, ..., l - m$ be the number of αs that are at least as large as $\alpha_{(s)}$. Given the confidence level $1 - \delta$ and a new example x_{l+1}, the *predictive region* is derived as $(\hat{y}_{l+1} - \alpha_{(s)}, \hat{y}_{l+1} + \alpha_{(s)})$, by choosing the j_s such that $\delta = j_s/(l - m + 1)$.



(a) The LeaFi index building workflow.



(b) The LeaFi query answering workflow. The quality target can be auto-tuned for each individual query.

Fig. 4. The index building and query answering workflow of a LeaFi-enhanced series index.

In Section 4.4, we further discuss how to establish the mapping between αs and search result quality, and then identify the corresponding αs to auto-tune the filter predictions for the support of search quality targets.

4 The LeaFi Framework

In this section, we present the algorithm details of enhancing data series indexes with learned filters. We start with developing the notations for a LeaFi-enhanced index structure. Then, in Section 4.1, we outline the LeaFi-enhanced series index building and search workflow, illustrated in Figure 4. The details of leaf node selection, training data generation and filter conformal auto-tuning, are presented in Section 4.2, Section 4.3 and Section 4.4, respectively.

As illustrated in Figure 2, a LeaFi-enhanced tree-based series index contains learned filters in a selected subset of its leaf nodes. A learned filter M_i in a node L_i is a trained machine learning model, which takes in a query S_q and predicts d^{L}. We denote the prediction as $d^{\mathrm{f}}(S_q, M_i)$, and the conformal adjusted prediction as $d^{\mathrm{F}}(S_q, M_i)$. Hence, the learned filter-based pruning decision can be made by comparing the best-so-far distance d^{bsf} with the predicted lower bound d^{F}. If $d^{\mathrm{F}} > d^{\mathrm{bsf}}$, we prune L_i, otherwise we search L_i.

4.1 LeaFi-enhanced Series Index Workflow

We modify both the index building workflow and the search workflow of original tree-based series indexes to enable learned filters, as illustrated in Figure 4. We outline these procedures as follows, and present the details in later sections.

4.1.1 LeaFi-enhanced Index Building Overview. For index building, we add three main steps after the original index is built, illustrated in Figure 4a. These steps are (a) select leaf nodes for filter insertion, (b) generate training data from both global (index-wise) context and local (node-wise) context, and train the inserted filters, and (c) collect conformal training data and fit the conformal auto-tuners.

Algorithm 1 LeaFi-enhanced Index Building

Input: a series collection $\mathcal{S}$ of size n, a tree-based index I, the training data size $n_q = n_{q(g)} + n_{q(l)}$, and the available GPU memory c^{M}
Output: the LeaFi-enhanced index I^{F}
1: $\{N_i^{\mathrm{F}}\}$ = SelectLeafNode(I, c^{M})
2: $\mathcal{S}_{q(g)}$ = GenerateGlobalQueries($\mathcal{S}$)
3: $T_g := \{(d^{bsf}, d^{\mathrm{L}})\}$ = SearchGlobalQueries(I, $n_{q(g)}$, $\{N_i^{\mathrm{F}}\}$)
4: $\mathcal{S}_{q(l)} = \emptyset$, $T_l = \emptyset$
5: **for all** $N_i^{\mathrm{F}} \in \{N_i^{\mathrm{F}}\}$ **do**
6: $\quad \mathcal{S}_{q(l),i}$ = GenerateLocalQueries(N_i^{F}, $n_{q(l)}$, T_g)
7: $\quad T_{l,i} := \{d^{\mathrm{L}}\}$ = SearchLocalQueries(N_i^{F}, $n_{q(l)}$,)
8: $\quad \mathcal{S}_{q(l)} = \mathcal{S}_{q(l)} \cup \mathcal{S}_{q(g),i}$, $T_l = T_l \cup T_{l,i}$
9: I^{f} = TrainFilters($\{N_i^{\mathrm{F}}\}$, $\mathcal{S}_{q(g)}$, T_g, $\mathcal{S}_{q(l)}$, T_l)
10: I^{F} = FitAutoTuners(I^{f}, $\mathcal{S}_{q(g)}$, T_g)
11: **return** I^{F}

Select leaf nodes (Section 4.2). We first establish a general framework that selects the optimal subset of leaf nodes that maximize the benefit of learned filter insertion, using knapsack solvers [27]. We show that, under certain assumptions, we can simplify the framework and select leaf nodes using a greedy algorithm. In brief, LeaFi calculates a leaf node size threshold th, based on the number of distance calculations that execute in the same time as one filter inference. We then select the leaf nodes according to their sizes until th is met, or the GPU memory is fully occupied.
Generate training data (Section 4.3). We generate training data $\mathcal{S}_q$ from both global and local contexts. Index-wise training data $\mathcal{S}_{q(g)}$ is sampled from the whole dataset and shared among all selected leaf nodes, while node-wise training data $\mathcal{S}_{q(l)}$ is sampled individually from each node and is only available to their corresponding node. There is no additional expense by splitting $\mathcal{S}_q$ into $\mathcal{S}_{q(g)} \cup \mathcal{S}_{q(l)}$. The collection time for $\mathcal{S}_{q(g)}$ and $\mathcal{S}_{q(l)}$ node-wise data is the same as for $n_q = n_{q(g)} + n_{q(l)}$ index-wise training data.
Fit conformal auto-tuners (Section 4.4). After inserted filters fit the training set, we fit the conformal auto-tuners to enable the support for user-request search quality targets. The absolute prediction errors $\{\alpha_i\}$, i.e., the non-conformity scores in the conformal regression context [2], are employed as offsets to auto-tune the predictions. One auto-tuner is added to each learned filter.
Index building pseudocode. We describe the LeaFi-enhanced index building pseudocode for a general tree-based index building algorithm in Algorithm 1. The sub-functions SelectLeafNode is defined later in Algorithm 3 and FitAutoTuners in Algorithm 4.

Given a series collection $\mathcal{S}$, a tree-based index I, the training data size $n_q = n_{q(g)} + n_{q(l)}$, and the available GPU memory c^{M}, we first select the subset of leaf nodes $\{N_i^{\mathrm{F}}\}$. We then generate the collect the index-wise training data from line 2 to line 3. From line 5 to line 8, we iterate over each selected leaf node N_i^{F} and generate its corresponding node-wise training data. Line 9 trains all the inserted filters using the both index-wise and node-wise training data. We finalize the LeaFi-enhanced index building workflow by fitting the conformal auto-tuners in line 10.

4.1.2 LeaFi-enhanced Search Overview. There are two new operations in LeaFi-enhanced search workflow, illustrated in Figure 4b, compared to the original search workflow. Before searching for a query, we first auto-tune the filters according to the user-request search quality target. Then, we

Algorithm 2 LeaFi-enhanced Search

Input: a query S_q, a LeaFi-enhanced index I^{F}, the result quality target Q, and the number of nearest neighbors k
Output: the search results $\mathcal{S}_r$
1: I^{F} = AutoTuneFilters(I^{F}, Q)
2: d^{bsf} = INF, $\mathcal{S}_r$ = PriorityQueue($\emptyset$, k)
3: **for all** $N_i \in [N_j]$, ordered by I^{F} **do**
4: **if** $d^{\mathrm{lb}} > d^{\mathrm{bsf}}$ **then**
5: **continue**
6: **if** $M_i \in N_i$ & $d^{\mathrm{F}} > d^{\mathrm{bsf}}$ **then**
7: **continue**
8: $S_{r,i}$, d^{L} = SearchNode(N_i, d^{bsf})
9: **if** $d^{\mathrm{L}} <= d^{\mathrm{bsf}}$ **then**
10: $d^{\mathrm{bsf}} = d^{\mathrm{L}}$, $\mathcal{S}_r = \mathcal{S}_r \oplus S_{r,i}$
11: **return** $\mathcal{S}_r$

visit each leaf node, and check whether it can be pruned using a cascade of summarization-based strategy and learned filter-based strategy as in Figure 2.

Auto-tune learned filters (Section 4.4). Given a user-specified search quality target q_j, we use the learned mapping f^c to calculate the corresponding adjusting offset $o_{i,j} = f^c(q_j)$ for each filter M_i. Then, we use the left boundary $d_i^f - o_{i,j}$ of the adjusted interval as the predicted lower bound $d_{i,j}^F$, to be compared with d^{bsf} for the pruning decision of node N_i.

Search pseudocode. We describe the LeaFi-enhanced search pseudocode for a general tree-based index search algorithm in Algorithm 2.

Given a query S_q, a LeaFi-enhanced index I^{F}, the result quality target Q, and the number of nearest neighbors k, we first auto-tune the learned filters based on the target Q. A hash table is employed to avoid redundant computation for the same Qs, which we omitted in Algorithm 2 for simplicity. We then iterate across the leaf nodes according to the order suggested by I^{F}. When visiting a node N_i, we first check whether it can be pruned using d^{lb} in line 4. Otherwise, we check whether N_i can be pruned using d^{F}, if a learned filter M_i resides in N_i in line 6. If neither d^{lb} nor d^{F} can prune N_i, we search N_i for S_q to obtain its node-wise nearest neighbor results $\mathcal{S}_{r,i}$ and update $\mathcal{S}_r$ from line 8 to line 10.

4.2 Leaf Node Selection

In this section, we present how the problem of leaf node selection can be formalized as a knapsack problem [27] in Section 4.2.1. We show that this general formalization can be simplified under certain assumptions and solved efficiently by a greedy algorithm.

Briefly speaking, our greedy leaf node selection algorithm works as follows. First, we sort all leaf nodes $\{N_i\}$ by their sizes $\{|N_i|\}$ in a non-increasing order. Then, we select those nodes whose sizes exceed a threshold th to insert filters, until all available GPU memory is occupied. We describe how th is derived from a simplified knapsack formalization and how it is determined in Section 4.2.2.

4.2.1 A general solution framework as a knapsack problem. We start with a brief review of the knapsack problem, and then built the analogy between the knapsack problem and leaf node selection.

The knapsack problem is a classic example of combinatorial optimization [27]. It is a type of resource allocation problem where the objective is to maximize the total value of items placed in a knapsack without exceeding its capacity. Given a set of n items, each with a weight w_i and a value v_i, and a knapsack with a weight capacity W, the goal is to determine the subset of items to include in the knapsack such that the total weight does not exceed W and the total value is maximized. This can be represented by the following optimization problem: maximize $\sum_{i=1}^{n} v_i x_i$, subjective to $x_i \in \{0, 1\}$ and $\sum_{i=1}^{n} w_i x_i \leq W$. Here, x_i is a binary decision variable that indicates whether item i is included ($x_i = 1$) or excluded ($x_i = 0$) in the knapsack. There are several variants of the knapsack problem, among which we only consider 0/1 knapsack problem in this paper.

The analogy between a knapsack problem is built as the following. For n^N leaf nodes, we consider that there are n^N filters, each of which is corresponding to one specific node. In this case, a filter M_i can be considered as an item. Its value is the expected reduction of search time, denoted by b_i, induced by inserting M_i into its corresponding leaf node N_i. The weight of an item M_i is its GPU memory footprint, and the weight capacity of the knapsack is the volume of available GPU memory c^{M}. Hence, by solving the mapped knapsack problem, we identify the subset of leaf nodes that obtain the optimal search time improvement by learned filter insertion. We formalize the leaf node selection as a knapsack problem in Equation 1:

$$\begin{aligned} \max \quad & \sum_i b_i x_i \\ \text{s.t.} \quad & \sum_i w_i x_i \leq c^{\mathrm{M}} \\ & x_i \in \{0, 1\}, \forall i = 1, ..., n^{\mathrm{N}}. \end{aligned} \tag{1}$$

where w_i and c^{M} can be calculated analytically [20] or measured empirically using GPU profiling tools [41]. In this work, we only consider the case where all learned filters share the same neural network architecture. As a result, they also share the same memory footprint, i.e., $w_i = w, \forall i = 1, ..., n^{\mathrm{N}}$. This general framework can also tackle the case where different filters have different network architectures by extending it to a multiple-choice knapsack problem [27]. We will study this case in our future work.

Formulate the expected time reduction b_i**.** To solve Equation 1, we need to estimate the only unknown variable b_i , i.e., the reduced search time by adding a filter M_i to a specific leaf node N_i. We observe that b_i is influenced by the following factors: the node size $|N_i|$, the summarization-based pruning probability p^{lb}, the filter-based pruning probability p^{F}. A larger $|N_i|$ or a smaller p^{lb} indicates the original index I spent more effort on node N_i, and a larger p^{F} hints that filter M_i works well for node N_i. Hence, b_i can be formulated using $|N_i|$, p^{lb} and p^{F} as Equation 2:

$$b_i = (1 - p_i^{\mathrm{lb}}) \times (p_i^{\mathrm{F}} \times t^{\mathrm{S}} \times |N_i| - t^{\mathrm{F}}) \tag{2}$$

where t^{S} denotes the distance calculation time for one series and t_j^{F} denotes the inference time for filter M_i. Note that b_i can be negative in cases where $|N_i|$ is small or t^{F} is large. We describe how we prohibit such insertions by automatically establishing a threshold for $|N_i|$ using t^{S} and t^{F} in Section 4.2.2.

Challenge: estimate the filter-based pruning probability p^F**.** Although we can measure the wall-clock time for t^{S} and t^{F} empirically using trial experiments, there are two other variables in Equation 2 that cannot be directly calculated or measured, i.e., the summarization-based pruning probability p^{lb} and the filter-based pruning probability p^{F}.

In our preliminary studies, we find p^{lb} can be estimated by collecting d^{lb} along with d^{bsf} and d^{L} using trial experiments. However, estimating p^{F} means to accurately estimate the performance of

Algorithm 3 Leaf Node Selection

Input: the leaf nodes $\{N_i\}$ of index I^{F}, the available GPU memory c^{M} and hyperparameter a
Output: a selected subset of leaf nodes $\{N_i^F\}$ for filter insertion

1: w = MeasureFilterGPUMemory(M)
2: t^{F} = MeasureFilterInferenceTime(M)
3: t^{S} = MeasureDistanceCalculationTime($\cdot$)
4: $th = a \times t^{\mathrm{F}}/t^{\mathrm{S}}$
5: $[N_{(i)}]$ = SortBySize($\{N_i\}$)
6: $\mathcal{N}_r = \emptyset$, $w^{\mathrm{M}} = 0$
7: **for all** $N_{(j)} \in [N_{(i)}]$ **do**
8: **if** $|N_{(j)}| \geq th$ & $w^{\mathrm{M}} < c^{\mathrm{M}}$ **then**
9: $\mathcal{N}_r = \mathcal{N}_r \oplus N_{(j)}$, $w^{\mathrm{M}} = w^{\mathrm{M}} + w$
10: **return** $\mathcal{N}_r$

machine learning models without fully fitting their respective training data. This is a nontrivial task, and has been under extensive studies in the automated machine learning (AutoML) literature [53, 72]. Considering the positioning of LeaFi is a proof-of-concept study to demonstrate the potential of embracing learned filters in series indexes, we opt to simplify Equation 2 by assuming that p^{lb} and p^{F} are the same across leaf nodes.

4.2.2 The Greedy Selection Algorithm. To tackle the challenge of accurately estimating the performance of machine learning models, we propose to assume that p^{lb} and p^{F} are the same across leaf nodes in LeaFi. That is, $\forall i = 1, ..., n^{\mathrm{N}}$, both $p_i^{\mathrm{lb}} = p^{\mathrm{lb}}$ and $p_i^{\mathrm{F}} = p^{\mathrm{F}}$ hold. Although this assumption is coarse-grained, we show later in Equation 4 that it can derive a safe greedy algorithm that has no negative effect. Under this assumption, Equation 2 can be simplified into Equation 3:

$$b_i = (1 - p^{\mathrm{lb}}) \times (p^{\mathrm{F}} \times t^{\mathrm{S}} \times |N_i| - t^{\mathrm{F}}) \tag{3}$$

where b_i is only correlated to $|N_i|$ (t^{S} and t^{F} are also the same across leaf nodes). Moreover, the correlation between b_i and $|N_i|$ is positive. That is, a larger node size $|N_i|$ introduces larger search time reduction b_i under this assumption. Hence, we can sort the leaf nodes based on their sizes and select them the larger nodes until consuming all available GPU memory.

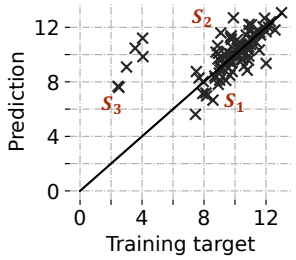
However, as mentioned in Section 4.2, b_i can be negative in cases where $|N_i|$ is small or t^{F} is large, which should be prohibited. To formalize the constraint, we set $b_i > 0$, transform Equation 3 and get a threshold th for leaf node size $|N_i|$ in Equation 4:

$$b_i > 0 \Rightarrow \ |N_i| > \frac{1}{p^{\mathrm{F}}} \times \frac{t^{\mathrm{F}}}{t^{\mathrm{S}}} \Rightarrow \ |N_i| > a\frac{t^{\mathrm{F}}}{t^{\mathrm{S}}} \tag{4}$$

where $a \coloneqq 1/p^{\mathrm{F}}$ is a hyperparameter. $t^{\mathrm{F}}/t^{\mathrm{S}}$ is the number of series that its distance calculation time is equivalent to the inference time of a learned filter. By its definition $1/p^{\mathrm{F}}$, a has an intrinsic interpretation as being the inverse of the filter-based pruning probability p^{F}. Hence, it is intuitive to set a by choosing a lower bound for p^{F} to ensure $b_i > 0$. In our experiments, we set $a = 2$ (i.e., $p^{\mathrm{F}} = 50\%$) and evaluated that it worked well across different index prototypes, datasets, query noise levels and search quality targets.

Leaf node selection pseudocode. Putting all these designs together, we present our leaf node selection approach of the LeaFi index building workflow in Algorithm 3.

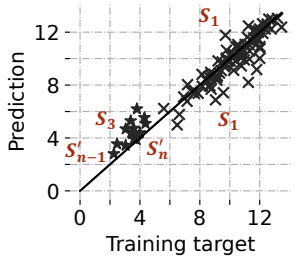
We first collect all necessary runtime statistics, including the GPU memory footprint of a filter in line 1, the filter inference time in line 2 and the series distance calculation time in line 3. Then, we



	Node	d^{L_1}	d^f
S_1	$\mathbb{G}, L_7$	8.4	7.9
S_2	$\mathbb{G}, L_5$	9.8	11.1
S_3	$\mathbb{G}, L_1$	2.3	7.7
...			

☹

(a) Only global training data (× in figure, $\mathbb{G}$ in table) for leaf node L_1. The Node column indicates in which node each series finds its nearest neighbor.



	Node	d^{L_1}	d^f
S_1	$\mathbb{G}, L_7$	8.4	7.5
S_2	$\mathbb{G}, L_5$	9.8	11.9
S_3	$\mathbb{G}, L_1$	2.3	3.8
...			
S'_{n-1}	$\mathbb{L}, L_1$	2.1	2.4
S'_n	$\mathbb{L}, L_1$	3.9	3.9

☺

(b) Both global (×) and local (⋆ in figure, $\mathbb{L}$ in table) training data.

Fig. 5. The value distribution of the training targets (x-axis) for filters. Considering the global examples only overlooks the more important small value ranges, where the global nearest neighbor results are found.

calculate the node size threshold th based on Equation 4 in line 4. In line 5, we sort all the leaf node in a non-increasing order in line 5. From line 6 to line 9, for each leaf node $N_{(j)}$, if its size $|N_{(j)}|$ exceeds th and there is GPU memory available, we add $N_{(j)}$ into the set of selected leaf nodes $\mathcal{N}_r$. After checking all leaf nodes, we return $\mathcal{N}_r$ as in line 10.

4.3 Training Data Generation

After inserting filters into the selected subset of leaf nodes, we need to train these filters using proper training data.

Challenge: imbalanced target value range. However, we observe that even in the presence of a large real workload, preparing training data for the filters is nontrivial. The challenge lies in the fact that the majority of the node-wise nearest neighbor distances fall out of the value range of global nearest neighbor distances. This observation holds in general, because the tree-based series indexes are expected to group similar series into the same leaf nodes. Only a few leaf nodes should have similar series to the query, which have small node-wise nearest neighbor distances close to the global nearest neighbor distances.

Figure 5 further illustrates this observation and demonstrates its pollution in filter training. In Figure 5a, only five example fall into the global nearest neighbor distance range around $[3, 5]$, and only S_3 finds its nearest neighbor in L_i with a distance 2.3. The majority of the node-wise nearest neighbor distances are located in $[7.5, 12.5]$. Thus, the model being trained on targets of $[7.5, 12.5]$, also predicts $d^f(S_3, L_1) = 7.7$, which is much larger than the actual 2.3. This causes the model to generate inaccurate predictions, hence, requiring large adjusting ranges, and leading to small

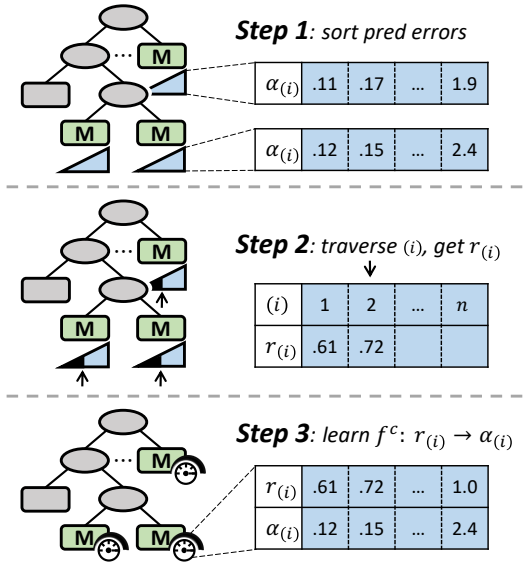


Fig. 6. Collect training examples on calibration set for conformal auto-tuners, and learn the mapping between search result qualities and prediction adjusting offsets.

pruning enhancements. This disparity not only brings harmful target bias on filter training [51, 73], but also invalidates the conformal regression as the absolute prediction errors will possess difference different distributions for the examples in different target ranges [44, 61].

Our solution: generating both global and local data. To resolve this issue, we propose a novel twofold strategy for training data generation. It consists of generating global training queries that are derived from the entire series collection and applicable to all leaf nodes, alongside node-wise training queries that are derived from and specific to each selected leaf node. As illustrated in Figure 5b, our twofold approach ensures that the training data encompasses both global (index-wise) and local (node-wise) contexts, providing unbiased training target value ranges.

In LeaFi, we employ the conventional query generating approach from the data series literature [13, 15, 16] for both index-wise and node-wise training data. We uniformly select $n_{q(g)}$ random series $\mathcal{S}_{q(g)}$ from the entire collection, as well as $n_{q(l)}$ random series $\mathcal{S}_{q(l),i}$ from each select leaf node N_i^F, and then add random Gaussian noise. We then search the index for $\mathcal{S}_{q(g)}$ to collect the training targets $T_g \coloneqq \{(d^{bsf}, d^{\mathrm{L}})\}$, i.e., the node-wise nearest neighbor distances $\{d^{\mathrm{L}}\}$ and the best-so-far distances $\{d^{\mathrm{bsf}}\}$. $\{d^{\mathrm{bsf}}\}$ is utilized later to fit the auto-tuners for the support of user-requested search quality targets. For each set of node-wise training series $\mathcal{S}_{q(l),i}$, we only search its corresponding leaf node N_i^F to collect the training targets $T_{l,i} \coloneqq \{d^{\mathrm{L}}\}$. To train the filter M_i inserted into N_i^F, we use $\mathcal{S}_{q(g)} \cup \mathcal{S}_{q(l),i}$ as the input and $T_g \cup T_{l,i}$ as the targets. In our experiments, we empirically set the split $n_{q(g)}/n_{q(l)} = 3$ to obtain a balanced target values ranges. This process is presented in lines 5-8 of Algorithm 1.

4.4 Filter Conformal Auto-tuning

In this section, we describe how the LeaFi-enhanced indexes can be auto-tuned to support some user-defined qualities of the query answers. The conformal auto-tuners make use of the prediction statistics from a separate calibration set [44] to build a mapping between the search quality and

Algorithm 4 Auto-tuner Learning

Input: a set of learned filters $\{M_i\}$ for index I^{F}, a set of calibration series $\mathcal{S}_{q(c)}$
Output: learned auto-tuners $\{f_i^c\}$

```
1: for all M_i ∈ {M_i} do
2:     E_i = ∅
3:     {α}_i = CalAbsPredErrors(M_i, S_q(c))
4:     [α]_i = Sort({α}_i)
5: for all j ∈ [1, |S_q(c)|] do
6:     for all M_i ∈ {M_i} do
7:         o_j,i = α_i,(j)
8:     for all S_q(c),k ∈ S_q(c) do
9:         S_r,k = SimulateSearch(I^F, {o_i}_j, S_q(c),k)
10:        q_j,k = EvaluateQuality(S_r,k)
11:        for all M_i ∈ {M_i} do
12:            E_i = E_i ⊕ (q_j,k, o_j,i)
13: for all M_i ∈ {M_i} do
14:     f_i^c = FitSplineRegression(E_i)
15: return {f_i^c}
```

prediction adjusting offset for each trained filter. In LeaFi, the calibration set is a subset of the index-wise training data.

Briefly speaking, we employ absolute prediction errors α as the adjusting offsets o, following the common exercise of inductive conformal regression [44]. During LeaFi-enhanced index building, the training data for auto-tuners are the examples of the mapping between offsets o and search result qualities q, simulated and collected using the calibration set. We then fit a spline regression [57] on these examples to obtain a learned mapping $f^c : q \rightarrow o$. In LeaFi-enhanced search, given a user-requested quality target q, we first determine the corresponding adjusting offset $o_i = f_i^c(q)$ for each learned filter M_i. Then, the adjusted filter prediction $d_i^{\mathrm{F}} = d_i^{\mathrm{f}} - o_i$ is calculated to help prune node N_i if necessary.

4.4.1 Conformal Auto-tuner Learning in Index Building. Figure 6 shows how we prepare the training data for the conformal auto-tuners in three steps.

First, we calculate the absolute prediction errors $\{\alpha\}_i$ on the calibration set for each learned filter M_i. $\{\alpha\}_i$ are then employed as the candidate adjusting offsets o_i. To avoid enumerating all possible combinations of candidate offsets for all inserted leaf nodes, we sort o_i (Step 1 in Figure 6), such that we can iterate over the sorted positions (j), instead of enumerating the combinations. Second, we iterate over sorted positions (j) to get the corresponding result quality $q_{(j)}$ on the calibration set. For each (j), we select the corresponding error $o_{i,(j)}$ as the adjusting offset of M_i. We then simulate LeaFi-enhanced search for the calibration queries, and calculate the result quality $q_{(j)}$ (Step 2 in Figure 6). After checking all sorted positions, we collect the examples $\{(q_{(j)}, o_{i,(j)}\}$ of the mapping between the result quality and the adjusting offset for each M_i. Using these examples, we train a spline regression model [57] as the auto-tuner to fit the mapping $f_i^c : r \rightarrow \alpha$ (Step 3 in Figure 6).

Auto-tuner learning pseudocode. We present the auto-tuner training pseudocode in Algorithm 4. In line 1 to line 4, we traverse each inserted filter M_i, evaluate it on the calibration set $\mathcal{S}_{q(c)}$, calculate and sort the absolute prediction errors. We then iterate over all sorted position in line 5. For each

sorted position j, we set the adjusting offset o_{j_i} to be the sorted error $\alpha_{i,(j)}$ for each filter M_i in line 7. We simulate the LeaFi-enhanced search for the calibration queries $\mathcal{S}_{q(c)}$ and calculate their achieved result qualities $\{q_{j,k}\}$ in line 9 to line 10, and add the (adjust offset, result quality) pair $(q_{j,k}, o_{j,i})$ to M_i's auto-tuner training set in line 12. Finally, we train and return the auto-tunes $\{f_i^c\}$.

4.4.2 Filter Prediction Auto-tuning in Search. In LeaFi-enhanced search stage, given a user-requested quality target q, we first determine the corresponding adjusting offset o_i using the learned mapping $f_i^c(q)$ for each learned filter M_i. The filter prediction is adjusted accordingly, i.e., $d_i^{\mathrm{F}} = d_i^{\mathrm{f}} - o_i$, based on which the filter-based pruning for node N_i is triggered.

4.5 Complexity Analysis

In this section, we briefly analyze the time and space complexity for the LeaFi framework. The cost of LeaFi is mainly from LeaFi-enhanced index building. In LeaFi-enhanced search, the overhead of filter inference is expected to be covered by search time reduction through the leaf node selection procedure in Section 4.2.

Among the four steps in LeaFi-enhanced index building, we note that the time bottleneck lies in training data generation and filter learning, rather than leaf node selection and conformal auto-tuner learning. The complexity of training data generation is $O(n_q \times n)$, linear to the training data size n_q and dataset size n. The filter training is proportional to the number of inserted filters n_F as well as the training time of an individual filter t_F, resulting in a complexity of $O(t_F \times n_F)$. Hence, the time complexity of LeaFi enhancement in index building (i.e., besides the original index building time complexity) is $O(n_q n + t_F n_F)$.

The space complexity of LeaFi $O(n_q + s_F n_F)$, although proportional to the training data size n_q and number of inserted filters n_F, is negligible compared to the dataset size n.

We note that with a reasonable training data size (2K series in our experiments) and a proper node size threshold ($th = a \cdot t^{\mathrm{F}}/t^{\mathrm{S}}$ in our experiments), LeaFi can provide considerable benefits when compared to the original indexes.

5 Experiments

In this section, we report our experiment evaluation using two different tree-based series indexes and five diverse datasets. The source code and datasets are available online[2].

5.1 Evaluation Setup

The experiments were carried out on a server equipped with an Intel(R) Xeon(R) Gold 6242R CPU, 520 GB RAM, and an NVIDIA Quadro RTX 6000 with 24 GB GDDR6 memory. The software environments were gcc/9.4.0, cuda/11.2, libtorch/1.13.1 (for MLP), and gsl/2.7.2 (for spline regression).

Datasets. We include a variety of datasets in our evaluation, consisting of one synthetic dataset and four real datasets from different domains. For the synthetic dataset, we choose RandWalk [17], which is generated by accumulating steps following a standard Gaussian distribution $N(0, 1)$. Regarding the real datasets, we select Seismic [59] from seismology, Astro [56] from astronomy, Deep [6] and SIFT [25] from image processing. Note that Deep and SIFT are two popular high-dimensional vector datasets of image descriptors (not data series). The series length is 256, except for Deep (96) and SIFT (128). Each dataset contains 25 million (i.e., 25M) data series in our experiments. In addition, we experimented with datasets of sizes between 10M-100M to test the scalability of LeaFi.

Following recent data series studies [13], we create four different query sets with varying levels of difficulty for every dataset. Each query set consists of 1,000 series, generated by adding 10%, 20%, 30%, and 40% Gaussian noise into uniform random samples. Additionally, we generate one training

set of 2,000 series for each dataset, by adding random levels of Gaussian noise $\in$ [10%, 40%]. The split of training and validation is 4:1.

Tree-based series indexes. We choose MESSI [47] (the SOTA variant of iSAX [55]) and DSTree [65] as the backbone indexes. The split threshold of node size is 10k for both indexes. We use 16 threads in MESSI for index building and query answering.

LeaFi instantiation. We use MLP to instantiate learned filters. Each MLP model have one hidden layer, whose dimension is set the same as the input series. We train these models using the stochastic gradient descent (SGD) [52] algorithm for at most 1,000 epochs. The learning rate, initialized as 0.01, is divided by 10 (until 10^{-5}) when the validation errors plateau.

We leverage multicore processing to accelerate the collecting of training data and the training of inserted filters. For the efficient collecting of training data, we propose a two-pass strategy that works for all tree-based indexes. In the first pass, we detach all the leaf nodes with inserted filters to calculate their node-wise nearest neighbor distances in parallel. Then in the second pass, we search for the global queries and collect the best-so-far distances efficiently by reusing the distances calculated in the first pass. For filter training, we assign one CUDA stream to each thread [40] and use 16 threads to train the models in parallel.

Comparison approaches. The exact search performance of the original indexes is an important baseline for LeaFi. Additionally, we choose ϵ-search, $\delta\epsilon$-search [16], ProS [14, 22] and LT [33] to represent the early-stopping strategy, as well as LR [26] to represent the reordering strategy.

Note that the comparison approaches do not natively support result quality targets. Moreover, LT employs sets of features designed specifically for graph-based indexes [38] and inverted indexes [5], which do not apply to the tree-based data-series indexes used in this study. Hence, we need to adjust these approaches to fit our scenario, and then also pay the cost to fine-tune them. Specifically:

(1) For ϵ-search, we grid-search for the maximal $\epsilon \in [1, 7]$ that provides $\geq$ 99% recall on the validation set [16]. A larger ϵ results in larger acceleration but lower recall. We set ϵ=1 when 99% recall cannot be achieved.

(2) For $\delta\epsilon$-search, we set ϵ=0 and then grid-search for the smallest $\delta \in$ [90%, 99.9%] that provides $\geq$ 99% recall on the validation set [16]. A larger δ results in smaller acceleration but higher recall. We set δ=99.9% when 99% recall cannot be achieved. We estimate the nearest neighbor distances using the node-wise nearest neighbor distances on the validation set.

(3) For ProS, we choose the early-stopping strategy that predicts whether the nearest neighbor results have been found after checking certain number of leaf nodes using the best-so-far distances [14, 22]. We set the early-stopping checking nodes to be [16, 64, 256, 512, 1024, 2048] for DSTree, and also [4096, 8192] for iSAX.

(4) For LT, we design our own features for tree-based series indexes following similar intuitions [33]: (a) the input query, (b) the ratio between the node-wise nearest neighbor distances of the first leaf node and the first [2, 4, 8, 16] leaf nodes for DSTree ([8, 16, 32, 64] for MESSI), (c) the best-so-far distances after examining the first [1, 2, 4, 8, 16] leaf nodes for DSTree ([1, 8, 16, 32, 64] for MESSI), and (d) the ratio between the node-wise nearest neighbor distance of the first leaf node and the best-so-far distance after examining the 16th node for DSTree (or 64th for MESSI). Besides the need for a new feature template, we also need to tune the multiplier, a hyperparameter that expands the predicted number of early-stop nodes. We grid-search for the minimal multiplier $\in [1, 20]$ that can provide $\geq$ 99% recall on the validation set. A larger multiplier results in smaller acceleration but higher recall. We set multiplier=20 when 99% recall cannot be achieved.

(5) For LR [26], we use the optimal reordering, i.e., the nearest neighbor results lie in the first examined leaf node, to provide the largest possible acceleration by any reordering strategies.

Evaluation measures. We report the query time (lower is better), recall-at-1 [33] (higher is better) and series pruning ratio (higher is better) in our experiments.

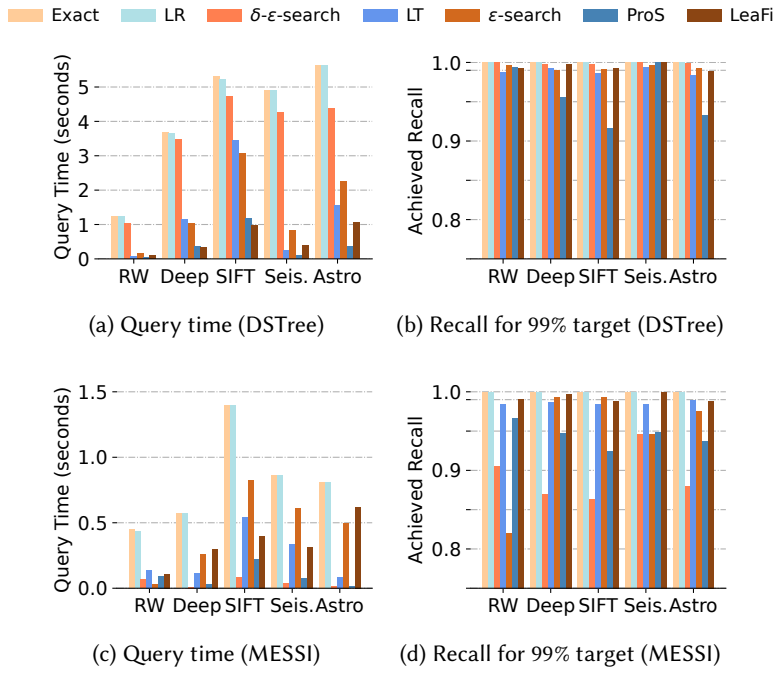


Fig. 7. The query time and achieved recalls at a target 99% recall for enhanced DSTree and MESSI indexes across datasets.

5.2 Main Results

We first report the average query time and actual recalls for a recall target 99% in Figure 7. We also provide a detailed analysis for the enhanced DSTree indexes by reporting the detailed results over different query noise levels, in Figure 8.

Overall, we find that LeaFi is the only solution that can provide a substantial search time improvement while achieving 99% recall in all test cases. LeaFi-enhanced series indexes improved pruning ratio by up to 20x (by DSTree on SIFT dataset with queries of 40% noise) and search time by up to 32x (by DSTree on RandWalk dataset with queries of 20% noise), while maintaining a target recall of 99%. Moreover, LeaFi is the only solution that supports ad-hoc quality targets, chosen at query time, independently for each query.

On the contrary, despite our best efforts to tune the comparison methods, none of the early-stopping strategies ($\delta\epsilon$-search, LT, ϵ-search, and ProS) can consistently achieve 99% recall across four query subsets of different noise levels. The reordering strategy (LR) failed to provide query time improvement. These empirical observations comply to our analysis in Section 2.

5.2.1 Main Results on DSTree. We first discuss the average search time and actual recalls for a recall target 99% on DSTree in Figure 7a and 7b. The split-down results across different query noise levels are reported in Figure 8.

LeaFi-enhanced DSTree obtained 99% recall on all 5 datasets, with an average speedup of 9.2x, and best speedup of 12.7x, when compared to the exact search. This is attributed to the largely improved pruning capability of DSTree leaf nodes. LeaFi achieved an average pruning ratio of 95.6%, compared to 59.8% without any learned filters (cf. Figure 8). ϵ-search and $\delta\epsilon$-search for DSTree

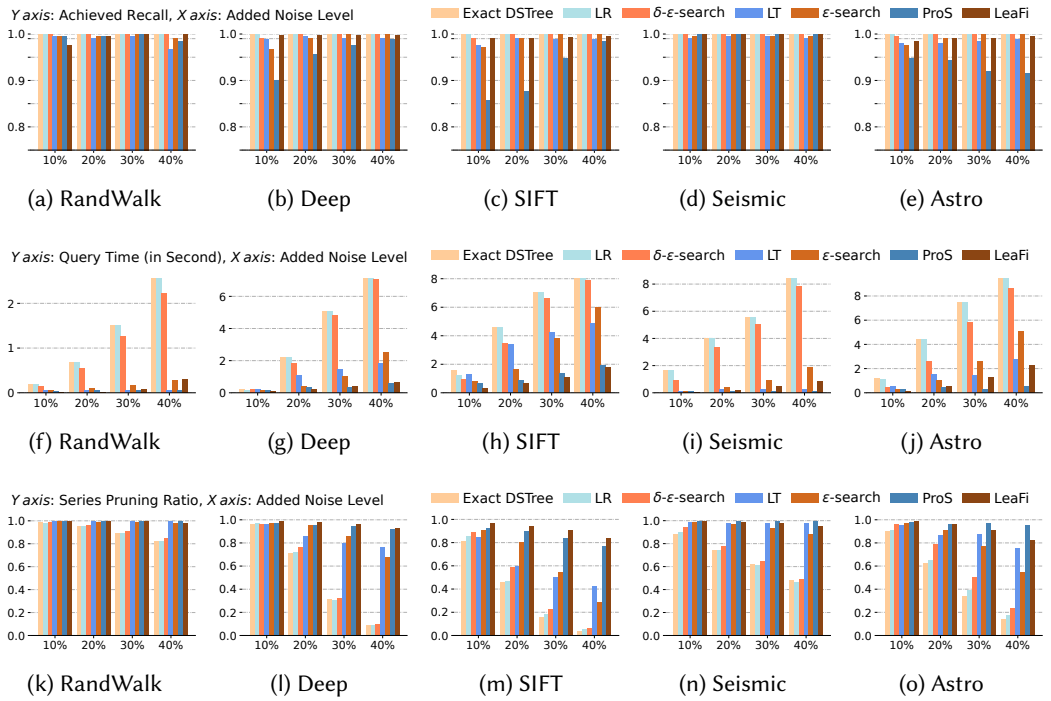


Fig. 8. The actual recall (8a to 8e), average query time (8f to 8j) and average series pruning ratio (8k to 8o) for enhanced DSTree indexes, targeting at 99% recall.

indexes also provided 99% recall on 5 datasets. However, LeaFi-enhanced DSTree are much more efficient, i.e., 2.4x faster than ϵ-search and 8.1x faster than $\delta\epsilon$-search on average. This observation indicates that although heuristic early-stopping approaches can effectively support quality targets, machine learning-based enhancements can provide larger query acceleration. We also observe that $\delta\epsilon$-search requires the accurate estimation of the nearest neighbor distances [16], which is extremely hard.

ProS- and LT-enhanced DSTree achieved 99% recall on only 2 datasets, which are RandWalk and Seismic for ProS, and Deep and Seismic for LT. The average recall of ProS is 96%, largely lagging behind the 99% target. Although the average recall of LT is 98.9%, LeaFi-enhanced DSTree are 1.9x faster than LT on average. LT's inability to achieve 99% recall by tuning the multiplier also implies that the model training for LT is much harder than LeaFi. This is because LeaFi fits one model to one leaf node with a small target range $\in[0, 10]$, while LT fits one model to the whole datasets with large target range $[10^2, 10^5]$.

Detailed analysis at different query noise levels. Breaking down the results across different query noise levels, we report the average search time, actual recalls and pruning ratios in Figure 8. LeaFi-enhanced DSTree achieved 99% recall on 17 of 20 (85%) cases, with an average of 99.4% and the lowest of 97.6%. The 3 cases of <99% recall are with queries of small noise levels, 10% on RandWalk, 10% and 20% on Astro. This is because these cases are equipped with much smaller nearest neighbor distances than cases of higher noise, hence challenging LeaFi's conformal auto-tuners. The best speedup of LeaFi-enhanced DSTree is 32.4x, obtained on RandWalk with queries of 20% noise. LeaFi achieved an average pruning ratio of 95.6%, compared to 59.8% without any learned filters.

Similarly to the aggregated results in Figure 8, ϵ-search and $\delta\epsilon$-search perform well in achieving the 99% recall target, but with small time performance improvements. ϵ-search achieved 99% recall on all 20 cases, while $\delta\epsilon$-search achieved 99% recall on 17 of 20 (85%) cases. On the other hand, ProS- and LT-enhanced DSTree struggled in achieving 99% recall target. ProS-enhanced DSTree achieved 99% recall on 7 (35%) cases, while LT-enhanced DSTree achieved 99% recall on 11 (55%) cases. LR, as expected, provided 100% recall with marginal improvement in terms of search time and pruning ratio, compared to LeaFi.

In conclusion, LeaFi is the only solution that can provide a substantial improvement in search time (up to 32.4x faster) and pruning ratio (up to 20x more), while consistently maintaining 99% recall.

5.2.2 Main Results on iSAX. We provide the average search time and actual recalls for a recall target 99% on MESSI in Figure 7c and 7d. Due to the lack of space, we remove the splitting-down figures across query noise levels to an extended version.

MESSI differs from DSTree in the following aspects: (1) MESSI has more leaf nodes, yet with smaller filling factors, than DSTree; (2) MESSI can provide tighter node summarization than DSTree, while DSTree better groups similar series into the same leaf nodes [3]. Moreover, the fact that MESSI utilizes the parallelization capabilities of modern hardware is an additional factor that affects performance. These elements explain the performance differences between enhanced DSTree and MESSI.

LeaFi improved MESSI query time on all 5 datasets while maintaining 99% recall, with an average speedup of 2.7x and the best speedup of 4x. In the splitting-down results across query noise levels, the best speedup is 13x on RandWalk with queries of 20% noise. The search time improvement is less than the pruning ratio improvement, which is improved from 55.2% to 86.2% on average (which is expected to provide a 3.2x speedup). We believe this is due to GPU resource contention as a result of concurrent filter inference requests. We further verified this speculation by varying the leaf node thresholds in Section 5.3.3. These results verified LeaFi can provide a substantial improvement in search time and pruning ratio, while achieving 99% recall for MESSI in a multithread environment.

For the comparison methods, only LT-enhanced MESSI achieved 99% recall on only 2 of the datasets. Other comparison methods cannot achieve 99% recall in any dataset, despite thorough tuning using the validation set. In the detailed results across query noise levels, LeaFi-enhanced MESSI achieved 99% recall on 14 of 20 (70%) cases, while ϵ-search and LT on 6 (30%) cases, $\delta\epsilon$-search and ProS on 1 (5%) case. Moreover, LeaFi-enhanced MESSI is faster than LT-enhanced MESSI on 3 datasets.

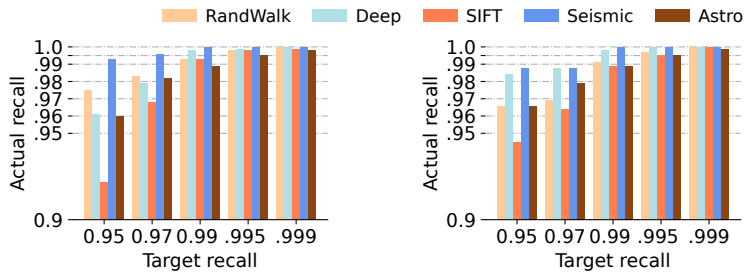
In summary, the results on MESSI are consistent with those on DSTree. LeaFi is the only solution that maintains 99% recall across queries of all noise levels, while providing considerable query speedups (up to 13x) and pruning ratio improvements (up to 76.2%).

5.3 Extended Analysis

In this section, we report the actual recalls under different recall targets in Figure 9, the query time for 100M datasets in Figure 10, and the query time across different leaf node threshold in Figure 11 to better understand the performance of LeaFi-enhanced indexes.

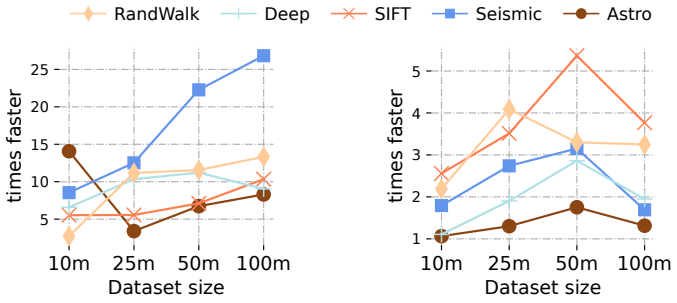
5.3.1 Varying Recall Target. We present the actual recalls of LeaFi-enhanced DSTree and MESSI for user-requested high recalls $\in$[95%, 99.9%] in Figure 9. These results verify that the conformal auto-tuner of LeaFi can effectively adjust the filter predictions to accommodate user quality targets.

Overall, LeaFi-enhanced DSTree achieved the recall targets on 31 of 35 (88.6%) cases and LeaFi-enhanced MESSI achieved the recall targets on 28 of 35 (80%). The recall difference when the target recall cannot be achieved is marginal, i.e., 0.54% for DSTree and 0.35% for MESSI. The largest



(a) DSTree (b) MESSI

Fig. 9. The target recall and actual (achieved) recall for LeaFi-enhanced DSTree and MESSI indexes across datasets.



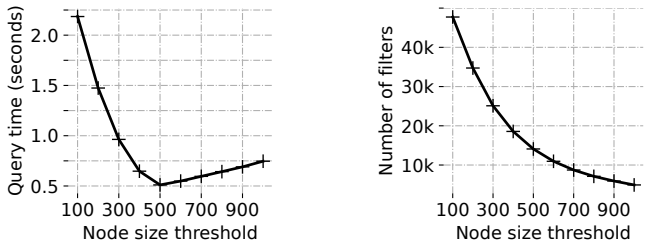
(a) DSTree (b) MESSI

Fig. 10. The query time improvements at a target 99% recall for LeaFi-enhanced DSTree and MESSI when varying the dataset size between 10M-100M.

difference was found by DSTree on the SIFT dataset, with a 95% recall target. We believe that the small number of validation examples (300) resulted in imbalanced statistics, leaving some queries sensitive to the adjusting offset for the 95% recall target. Even though our focus has been on very high recall targets, we note that LeaFi-enhanced indexes are effective for smaller recall targets, as well. As the recall target decreases, LeaFi leads to increased speedups. We omit these results for brevity.

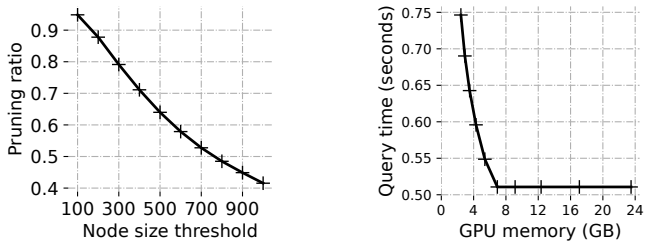
5.3.2 Varying Dataset Size. In Figure 10, we present the query time improvements of LeaFi-enhanced DSTree and MESSI for 99% recall target as we vary the dataset size between 10M-100M. We note that in this experiment, we build a different index structure for each dataset size (i.e., a tree with a different number of leaves and different sets of series in each leaf), which leads to a different set of inserted and trained filters in each case. This explains the non-smooth curves shown in the graphs.

Overall, the results demonstrate that LeaFi-enhanced series indexes consistently improve query answering times across all dataset sizes (while maintaining the 99% target recall). They also show a trend for larger query time improvements as the dataset size grows larger. Delving into the results, we observe that LeaFi-enhanced DSTree obtained an average speedup of 9.2x on the 25M



(a) Query time vs. node size threshold

(b) Number of inserted filters vs. node size threshold



(c) Pruning ratio vs. node size threshold

(d) Query time vs. GPU memory size

Fig. 11. Tradeoff between the available GPU memory and leaf node size threshold th, and the query time, pruning ratio, and number of inserted filters (for LeaFi-enhanced MESSI).

datasets, and a 13.6x average speedup on the 100M datasets. This increase in speedup is attributed to the increased pruning ratio for the 100M dataset (from 94.5% to 97.6%). On the other hand, LeaFi-enhanced MESSI obtained a 2.7x average speedup on the 25M datasets, and a 2.4x speedup on the 100M datasets. This behavior is due to the increased number of inserted filters (from ~10k to ~30k), which overwhelms the GPU access. Exploring techniques for efficient GPU utilization would be a promising direction going forward [64].

5.3.3 Varying Node Size Threshold. Figures 11a-c depict the search time and pruning ratios of LeaFi-enhanced MESSI across a range of leaf node thresholds on the 25M Deep dataset, with queries of 40% noise. With the increase of leaf node size threshold th, the search time first decreased then increased, while the pruning ratios kept decreasing. The decrease of the search time came from both the filter inference time and the multithreading GPU access competition overhead. Empirically, we measured $t^F/t^S \approx 279$ for the Deep dataset. By setting the hyperparameter $a = 2$, LeaFi used $th = 558$ for Deep dataset, which provided near-optimal performance on iSAX. Without the multithreading GPU access competition, LeaFi-enhanced DSTree has a turning point of ~100.

5.3.4 Varying GPU Memory. We present in Figure 11d the search time of LeaFi-enhanced MESSI across a range of GPU memory limits under the same setting as in Figures 11a-c. All reported results targeted at 99% recall, and achieved at least 99.3% actual recall. Overall, the query time starts to decrease and then stabilizes as we increase the available GPU memory. The decreasing phase is bounded by GPU memory, while the stabilization phase is bounded by the number of filters that are expected to improve query time. The point where improvement stops is at 6.9 GB, which

Table 1. Filter inference time (μs) and the node size threshold th derived for Deep (length 96).

Type	Time	th
MLP	46	558
CNN	258	3,129
RNN	11,695	141k
$d(\cdot,\cdot)$	0.16	-

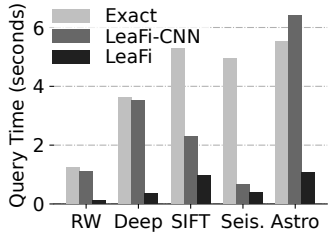


Fig. 12. LeaFi with CNN filters on DSTree.

Table 2. Query time (seconds) and actual recall for LeaFi-enhanced DSTree without (−Local) and with (+Local) local training data, for 99% recall target; +Local corresponds to our proposed LeaFi solution.

	Query time (s)			Actual recall	
Dataset	Exact	−Local	+Local	−Local	+Local
RandWalk	1.2	0.4	**0.1**	0.98 ✗	1.0 ✓
Deep	3.6	0.7	**0.3**	0.96 ✗	0.99 ✓
SIFT	5.3	1.3	**1.0**	0.94 ✗	0.99 ✓
Seismic	4.9	**0.3**	0.4	0.99 ✓	1.0 ✓
Astro	5.5	**1.0**	1.6	0.91 ✗	0.99 ✓

corresponds to inserting filters to the 14K leaf nodes with more than $th = 500$ series. Inserting more filters would mean that these additional filters would end up in leaf nodes with less the 500 series: in such cases, the filter inference time would be larger than the time to scan all series in the leaf, bringing no additional benefit. (The LeaFi-enhanced DSTree results are similar, and omitted for brevity.)

5.3.5 Varying Model Type. Table 1 and Figure 12 illustrate the results on instantiating LeaFi filters with Convolutional Neural Network (CNN) and Recurrent Neural Network (RNN) models. The CNN model has 2 convolutional layers with kernel size 3 and latent channel size equal to the series length. Similarly, the RNN model contains 2 LSTM blocks. Both CNN and RNN are much more computationally expensive than MLP, resulting in 5.6x and 254x larger node size thresholds. These high thresholds render the deployment of RNN filters not suitable in our setting (the RNN inference time would only be justified for leaf nodes of size $\geq 11k/0.16 * 2 = 141K$ series, while our leaves have a maximum capacity of 10K). Having $th = 3k$, the benefit of CNN is also limited, leading to a much smaller number of inserted filters (when compared to MLP), and in general, significantly smaller pruning ratios and query time improvements.

5.3.6 Without Local Training Data. We study the impact of local training data in Table 2. We observe that training LeaFi without local training data can still bring improvements in query time, but it cannot consistently achieve the 99% recall target. This is because the nodewise NN distances of the global queries are larger than the distances of their query results. Removing the local training queries also removes the query results' distance ranges from the training data, making these lower ranges ignored by the filters and conformal auto-tuners. These results confirm that local training data is necessary for LeaFi.

Table 3. Indexing time breakdown (minutes) for LeaFi-enhanced DSTree and MESSI on Seismic 100M.

	DSTree	MESSI
Indexing	28.6	5.3
Collecting data	104.2	67.5
Training	46.4	48.3

Table 4. Additional space overhead (GB) for LeaFi-enhanced DSTree and MESSI on Seismic 100M.

	DSTree	MESSI
Data	100	100
Index structure	1.5	0.3
Filters	4.7	4.8

5.3.7 Training Time. We report the indexing and training time for LeaFi-enhanced indexes in Table 3. LeaFi-enhanced DSTree spends more time in indexing and collecting training data, because it employs a more complex summarization (EAPCA), while LeaFi-enhanced MESSI needs longer training time, because it employes more filters than DSTree (19k vs. 18k). We note that if training time is a critical resource, the users can choose to train fewer filters, yet, still benefit from the improved pruning ratios. As Figure 11a shows, reducing the number of filters from ~14k to ~7k reduces the training time to half, while the query time only increases from ~0.5 to ~0.75 seconds, still outperforming the baseline.

5.3.8 Space Overhead. We report the space overhead for LeaFi-enhanced indexes in Table 3. The additional space needed by the filters is comparable for DSTree and MESSI, and is in both cases a small percentage (<5%) of the data size.

6 Conclusions and Future Work

In this paper, we present LeaFi, a framework that enhances tree-based series indexes with learned filters in order to accelerate data series similarity search, while satisfying a user-defined target recall. LeaFi can improve pruning ratio by up to 20x, and query answering time by up to 32x, while maintaining a target recall of 99%. These results set the foundations for future advancements in employing learned filters for data series similarity search, including the development of algorithms for inserting filters into both internal nodes and leaf nodes, estimating filter-based pruning ratios effectively, choosing among different learned filter models, and supporting updates. As updates may trigger node splitting, incremental learning [60] could play an important role in efficiently training the new filters of the children nodes based on the filter of the parent [9]. It would also be interesting to study the potential of LeaFi to enhance the performance of other index types, such as inverted indexes [5] and Locality-Sensitive Hashing (LSH) [24].

Acknowledgments

Work partially supported by EU Horizon projects AI4Europe (101070000), TwinODIS (101160009), ARMADA (101168951), DataGEMS (101188416) and RECITALS (101168490).

References

[1] Shun-ichi Amari. 1967. A Theory of Adaptive Pattern Classifiers. *IEEE Transactions on Electronic Computers* 16, 3 (1967), 299–307.
[2] Anastasios N. Angelopoulos and Stephen Bates. 2023. Conformal Prediction: A Gentle Introduction. *Found. Trends Mach. Learn.* 16, 4 (2023), 494–591.
[3] Ilias Azizi, Karima Echihabi, and Themis Palpanas. 2023. Elpis: Graph-Based Similarity Search for Scalable Data Science. *PVLDB* 16, 6 (2023), 1548–1559.
[4] Ilias Azizi, Karima Echihabi, and Themis Palpanas. 2025. Graph-Based Vector Search: An Experimental Evaluation of the State-of-the-Art. *PACMMOD* (2025).
[5] Artem Babenko and Victor S. Lempitsky. 2015. The Inverted Multi-Index. *TPAMI* 37, 6 (2015), 1247–1260.

[6] Artem Babenko and Victor S. Lempitsky. 2016. Efficient Indexing of Billion-Scale Datasets of Deep Descriptors. In *CVPR*. 2055–2063.

[7] Rene De La Briandais. 1959. File searching using variable length keys. In *IRE-AIEE-ACM Computer Conference (Western)*. 295–298.

[8] Manos Chatzakis, Panagiota Fatourou, Eleftherios Kosmas, Themis Palpanas, and Botao Peng. 2023. Odyssey: A Journey in the Land of Distributed Data Series Similarity Search. *PVLDB* 16, 5 (2023), 1140–1153.

[9] Tianyi Chen, Jun Gao, Hedui Chen, and Yaofeng Tu. 2023. LOGER: A Learned Optimizer towards Generating Efficient and Robust Query Execution Plans. *PVLDB* 16, 7 (2023).

[10] Bill Yuan-chi Chiu, Eamonn J. Keogh, and Stefano Lonardi. 2003. Probabilistic discovery of time series motifs. In *KDD*. 493–498.

[11] Hui Ding, Goce Trajcevski, Peter Scheuermann, Xiaoyue Wang, and Eamonn J. Keogh. 2008. Querying and mining of time series data: experimental comparison of representations and distance measures. *PVLDB* 1, 2 (2008), 1542–1552.

[12] Jialin Ding, Vikram Nathan, Mohammad Alizadeh, and Tim Kraska. 2020. Tsunami: A Learned Multi-dimensional Index for Correlated Data and Skewed Workloads. *PVLDB* 14, 2 (2020), 74–86.

[13] Karima Echihabi, Panagiota Fatourou, Kostas Zoumpatianos, Themis Palpanas, and Houda Benbrahim. 2022. Hercules Against Data Series Similarity Search. *PVLDB* 15, 10 (2022), 2005–2018.

[14] Karima Echihabi, Theophanis Tsandilas, Anna Gogolou, Anastasia Bezerianos, and Themis Palpanas. 2023. ProS: data series progressive k-NN similarity search and classification with probabilistic quality guarantees. *VLDBJ* 32, 4 (2023), 763–789.

[15] Karima Echihabi, Kostas Zoumpatianos, Themis Palpanas, and Houda Benbrahim. 2018. The Lernaean Hydra of Data Series Similarity Search: An Experimental Evaluation of the State of the Art. *PVLDB* 12, 2 (2018), 112–127.

[16] Karima Echihabi, Kostas Zoumpatianos, Themis Palpanas, and Houda Benbrahim. 2019. Return of the Lernaean Hydra: Experimental Evaluation of Data Series Approximate Similarity Search. *PVLDB* 13, 3 (2019), 403–420.

[17] Christos Faloutsos, M. Ranganathan, and Yannis Manolopoulos. 1994. Fast Subsequence Matching in Time-Series Databsases. In *SIGMOD*. 419–429.

[18] Panagiota Fatourou, Eleftherios Kosmas, Themis Palpanas, and George Paterakis. 2023. FreSh: A Lock-Free Data Series Index. In *SRDS*. 209–220.

[19] Hassan Ismail Fawaz, Germain Forestier, Jonathan Weber, Lhassane Idoumghar, and Pierre-Alain Muller. 2019. Deep learning for time series classification: a review. *DMKD* 33, 4 (2019), 917–963.

[20] Yanjie Gao, Yu Liu, Hongyu Zhang, Zhengxian Li, Yonghao Zhu, Haoxiang Lin, and Mao Yang. 2020. Estimating GPU memory consumption of deep learning models. In *ESEC/FSE*. 1342–1352.

[21] Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. 2014. Optimized Product Quantization. *TPAMI* 36, 4 (2014), 744–755.

[22] Anna Gogolou, Theophanis Tsandilas, Karima Echihabi, Anastasia Bezerianos, and Themis Palpanas. 2020. Data Series Progressive Similarity Search with Probabilistic Quality Guarantees. In *SIGMOD*. 1857–1873.

[23] Anna Gogolou, Theophanis Tsandilas, Themis Palpanas, and Anastasia Bezerianos. 2019. Comparing Similarity Perception in Time Series Visualizations. *TVCG* 25, 1 (2019), 523–533.

[24] Qiang Huang, Jianlin Feng, Yikai Zhang, Qiong Fang, and Wilfred Ng. 2015. Query-Aware Locality-Sensitive Hashing for Approximate Nearest Neighbor Search. *PVLDB* 9, 1 (2015), 1–12.

[25] Hervé Jégou, Romain Tavenard, Matthijs Douze, and Laurent Amsaleg. 2011. Searching in one billion vectors: Re-rank with source coding. In *ICASSP*. 861–864.

[26] Rong Kang, Wentao Wu, Chen Wang, Ce Zhang, and Jianmin Wang. 2021. The case for ml-enhanced high-dimensional indexes. In *AIDB@VLDB*.

[27] Hans Kellerer, Ulrich Pferschy, and David Pisinger. 2004. *Knapsack problems*. Springer.

[28] Eamonn J. Keogh, Kaushik Chakrabarti, Sharad Mehrotra, and Michael J. Pazzani. 2001. Locally Adaptive Dimensionality Reduction for Indexing Large Time Series Databases. In *SIGMOD*. 151–162.

[29] Kyoungmin Kim, Jisung Jung, In Seo, Wook-Shin Han, Kangwoo Choi, and Jaehyok Chong. 2022. Learned Cardinality Estimation: An In-depth Study. In *SIGMOD*. 1214–1227.

[30] Haridimos Kondylakis, Niv Dayan, Kostas Zoumpatianos, and Themis Palpanas. 2018. Coconut: A Scalable Bottom-Up Approach for Building Data Series Indexes. *PVLDB* 11, 6 (2018), 677–690.

[31] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In *SIGMOD*. 489–504.

[32] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *Nature* 521, 7553 (2015), 436–444.

[33] Conglong Li, Minjia Zhang, David G. Andersen, and Yuxiong He. 2020. Improving Approximate Nearest Neighbor Search through Learned Adaptive Early Termination. In *SIGMOD*. 2539–2554.

[34] Guoliang Li, Xuanhe Zhou, and Lei Cao. 2021. AI Meets Database: AI4DB and DB4AI. In *SIGMOD*. 2859–2866.

[35] Mingjie Li, Ying Zhang, Yifang Sun, Wei Wang, Ivor W. Tsang, and Xuemin Lin. 2020. I/O Efficient Approximate Nearest Neighbour Search based on Learned Functions. In *ICDE*. 289–300.

[36] Michele Linardi and Themis Palpanas. 2018. Scalable, Variable-Length Similarity Search in Data Series: The ULISSE Approach. *PVLDB* 11, 13 (2018), 2236–2248.
[37] Michele Linardi and Themis Palpanas. 2020. Scalable data series subsequence matching with ULISSE. *VLDBJ* 29, 6 (2020), 1449–1474.
[38] Yury A. Malkov and Dmitry A. Yashunin. 2020. Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *TPAMI* 42, 4 (2020), 824–836.
[39] Vikram Nathan, Jialin Ding, Mohammad Alizadeh, and Tim Kraska. 2020. Learning Multi-Dimensional Indexes. In *SIGMOD*. 985–1000.
[40] NVIDIA Corporation and Affiliates. 2023. CUDA Toolkit Documentation. https://docs.nvidia.com/cuda/. [Online; accessed 29-July-2023].
[41] NVIDIA Corporation and Affiliates. 2023. NVIDIA Management Library (NVML). https://developer.nvidia.com/nvidia-management-library-nvml. [Online; accessed 3-July-2023].
[42] Themis Palpanas. 2015. Data Series Management: The Road to Big Sequence Analytics. *SIGMOD Rec.* 44, 2 (2015), 47–52.
[43] Themis Palpanas and Volker Beckmann. 2019. Report on the First and Second Interdisciplinary Time Series Analysis Workshop (ITISA). *SIGMOD Rec.* 48, 3 (2019), 36–40.
[44] Harris Papadopoulos, Kostas Proedrou, Volodya Vovk, and Alexander Gammerman. 2002. Inductive Confidence Machines for Regression. In *ECML*, Vol. 2430. 345–356.
[45] John Paparrizos and Luis Gravano. 2015. k-Shape: Efficient and Accurate Clustering of Time Series. In *SIGMOD*. 1855–1870.
[46] John Paparrizos, Yuhao Kang, Paul Boniol, Ruey S. Tsay, Themis Palpanas, and Michael J. Franklin. 2022. TSB-UAD: An End-to-End Benchmark Suite for Univariate Time-Series Anomaly Detection. *PVLDB* 15, 8 (2022), 1697–1711.
[47] Botao Peng, Panagiota Fatourou, and Themis Palpanas. 2020. MESSI: In-Memory Data Series Indexing. In *ICDE*. 337–348.
[48] Botao Peng, Panagiota Fatourou, and Themis Palpanas. 2021. Fast Data Series Indexing for In-Memory Data. *VLDBJ* (2021).
[49] Botao Peng, Panagiota Fatourou, and Themis Palpanas. 2021. ParIS+: Data Series Indexing on Multi-Core Architectures. *TKDE* 33, 5 (2021), 2151–2164.
[50] Botao Peng, Panagiota Fatourou, and Themis Palpanas. 2021. SING: Sequence Indexing Using GPUs. In *ICDE*. 1883–1888.
[51] Rita P. Ribeiro and Nuno Moniz. 2020. Imbalanced regression and extreme value prediction. *Machine Learning* 109, 9-10 (2020), 1803–1835.
[52] Herbert Robbins and Sutton Monro. 1951. A stochastic approximation method. *The Annals of Mathematical Statistics* (1951), 400–407.
[53] Shubhra Kanti Karmaker Santu, Md. Mahadi Hassan, Micah J. Smith, Lei Xu, Chengxiang Zhai, and Kalyan Veeramachaneni. 2022. AutoML to Date and Beyond: Challenges and Opportunities. *ACM Computing Survey* 54, 8 (2022), 175:1–175:36.
[54] Gaurav Saxena, Mohammad Rahman, Naresh Chainani, Chunbin Lin, George Caragea, Fahim Chowdhury, Ryan Marcus, Tim Kraska, Ippokratis Pandis, and Balakrishnan (Murali) Narayanaswamy. 2023. Auto-WLM: Machine Learning Enhanced Workload Management in Amazon Redshift. In *SIGMOD*. 225–237.
[55] Jin Shieh and Eamonn J. Keogh. 2008. *i*SAX: indexing and mining terabyte sized time series. In *KDD*. 623–631.
[56] S Soldi, Volker Beckmann, Wayne H Baumgartner, Gabriele Ponti, Chris R Shrader, P Lubiński, HA Krimm, F Mattana, and Jack Tueller. 2014. Long-term variability of agn at hard x-rays. *Astronomy and Astrophysics* 563 (2014), A57.
[57] Matthias Steffen. 1990. A simple method for monotonic interpolation in one dimension. *Astronomy and Astrophysics* 239 (1990), 443.
[58] Ji Sun, Guoliang Li, and Nan Tang. 2021. Learned Cardinality Estimation for Similarity Queries. In *SIGMOD*. 1745–1757.
[59] Chad Trabant, Alexander R Hutko, Manochehr Bahavar, Richard Karstens, Timothy Ahern, and Richard Aster. 2012. Data products at the IRIS DMC: Stepping stones for research and other applications. *Seismological Research Letters* 83, 5 (2012), 846–854.
[60] Gido M. van de Ven, Tinne Tuytelaars, and Andreas S. Tolias. 2022. Three types of incremental learning. *Nature Machine Intelligence* 4, 12 (2022).
[61] Vladimir Vovk, Alexander Gammerman, and Glenn Shafer. 2005. *Algorithmic learning in a random world*. Vol. 29. Springer.
[62] Qitong Wang and Themis Palpanas. 2021. Deep Learning Embeddings for Data Series Similarity Search. In *KDD*. 1708–1716.
[63] Qitong Wang, Stephen Whitmarsh, Vincent Navarro, and Themis Palpanas. 2022. iEDeaL: A Deep Learning Framework for Detecting Highly Imbalanced Interictal Epileptiform Discharges. *PVLDB* 16, 3 (2022), 480–490.

[64] Shang Wang, Peiming Yang, Yuxuan Zheng, Xin Li, and Gennady Pekhimenko. 2021. Horizontally Fused Training Array: An Effective Hardware Utilization Squeezer for Training Novel Deep Learning Models. In *MLSys*.
[65] Yang Wang, Peng Wang, Jian Pei, Wei Wang, and Sheng Huang. 2013. A Data-adaptive and Dynamic Segmentation Index for Whole Matching on Time Series. *PVLDB* 6, 10 (2013), 793–804.
[66] Zeyu Wang, Peng Wang, Themis Palpanas, and Wei Wang. 2023. Graph- and Tree-based Indexes for High-dimensional Vector Similarity Search: Analyses, Comparisons, and Future Directions. *IEEE Data Eng. Bull.* 46, 3 (2023), 3–21.
[67] Zeyu Wang, Qitong Wang, Peng Wang, Themis Palpanas, and Wei Wang. 2023. Dumpy: A Compact and Adaptive Index for Large Data Series Collections. *PACMMOD* 1, 1 (2023), 111:1–111:27.
[68] Zeyu Wang, Qitong Wang, Peng Wang, Themis Palpanas, and Wei Wang. 2024. DumpyOS: A data-adaptive multi-ary index for scalable data series similarity search. *VLDBJ* 33, 6 (2024), 1887–1911.
[69] Jiuqi Wei, Botao Peng, Xiaodong Lee, and Themis Palpanas. 2024. DET-LSH: A Locality-Sensitive Hashing Scheme with Dynamic Encoding Tree for Approximate Nearest Neighbor Search. *PVLDB* 17, 9 (2024), 2241–2254.
[70] Ziniu Wu, Parimarjan Negi, Mohammad Alizadeh, Tim Kraska, and Samuel Madden. 2023. FactorJoin: A New Cardinality Estimation Framework for Join Queries. *PACMMOD* 1, 1 (2023), 41:1–41:27.
[71] Djamel Edine Yagoubi, Reza Akbarinia, Florent Masseglia, and Themis Palpanas. 2017. DPiSAX: Massively Distributed Partitioned iSAX. In *ICDM*. 1135–1140.
[72] Chengrun Yang, Jicong Fan, Ziyang Wu, and Madeleine Udell. 2020. AutoML Pipeline Selection: Efficiently Navigating the Combinatorial Space. In *KDD*. 1446–1456.
[73] Yuzhe Yang, Kaiwen Zha, Ying-Cong Chen, Hao Wang, and Dina Katabi. 2021. Delving into Deep Imbalanced Regression. In *ICML*, Vol. 139. 11842–11851.
[74] Kostas Zoumpatianos, Stratos Idreos, and Themis Palpanas. 2014. Indexing for interactive exploration of big data series. In *SIGMOD*. 1555–1566.

Received July 2024; revised September 2024; accepted November 2024