LMSFC: A Novel Multidimensional Index based on Learned Monotonic Space Filling Curves

Jian Gao
UNSW, Australia
jian.gao2@unsw.edu.au

Xin Cao
UNSW, Australia
xin.cao@unsw.edu.au

Xin Yao
Huawei Theory Lab, China
yao.xin1@huawei.com

Gong Zhang
Huawei Theory Lab, China
nicholas.zhang@huawei.com

Wei Wang[1,2]
[1]DSA & Guangzhou Municipal Key Laboratory of Materials Informatics, HKUST (Guangzhou), China
[2]HKUST, HKSAR, China
weiwcs@ust.hk

ABSTRACT

The recently proposed learned indexes have attracted much attention as they can adapt to the actual data and query distributions to attain better search efficiency. Based on this technique, several existing works build up indexes for multi-dimensional data and achieve improved query performance. A common paradigm of these works is to (i) map multi-dimensional data points to a one-dimensional space using a fixed space-filling curve (SFC) or its variant and (ii) then apply the learned indexing techniques. We notice that the first step typically uses a fixed SFC method, such as row-major order and z-order. It definitely limits the potential of learned multi-dimensional indexes to adapt variable data distributions via different query workloads.

In this paper, we propose a novel idea of learning a space-filling curve that is carefully designed and actively optimized for efficient query processing. We also identify innovative offline and online optimization opportunities common to SFC-based learned indexes and offer optimal and/or heuristic solutions. Experimental results demonstrate that our proposed method, LMSFC, outperforms state-of-the-art non-learned or learned methods across three commonly used real-world datasets and diverse experimental settings.

PVLDB Reference Format:
Jian Gao, Xin Cao, Xin Yao, Gong Zhang and Wei Wang. LMSFC: A Novel Multidimensional Index based on Learned Monotonic Space Filling Curves. PVLDB, 16(10): 2605 - 2617, 2023.
doi:10.14778/3603581.3603598

1 INTRODUCTION

Nowadays, there are large volumes and a huge variety of multi-dimensional data. For example, in traditional data warehouses and analytical databases, the majority of key data is stored in the *multi-dimensional* fact table. The wide deployments of location-based services and sensors, such as Google Maps, generate huge amounts of *multi-dimensional* data. The data is typically two or three spatial dimensions, and one or several dimensions for various measurements. The common and dominant type of query over these multi-dimensional datasets is the window query, which imposes range constraints on several or all the dimensions.

Multi-dimensional indexes are essential in answering window queries efficiently for a large volume of multi-dimensional datasets. Previous studies have proposed many traditional indexes, including R-tree [12], kd-tree [3], and Quadtree [9]. They are all based on spatial partitioning, while the major difference is whether the overlapping between partitions exists or not.

A space-filling curve is one of the most commonly used methods in multi-dimensional indexes [16, 29]. This is because SFCs have excellent proximity-preserving properties, making them ideal for linearizing data objects. SFCs can be classified into two categories: monotonic SFCs and non-monotonic SFCs. Monotonic SFCs, such as z-order [25], enable quick location of the search range. However, non-monotonic SFCs may result in more computational overhead during query processing. For instance, Hilbert curve [14] requires the enumeration of all values on the boundary of the query window to determine the search range. Therefore, it is more difficult to perform range searches when using non-monotonic SFCs as the linearization method.

Recently, initiated by the seminal work [19], there is a surge in optimizing database indexing via machine learning. It takes unique advantage of optimizing for specific data and query workload instances. As a result, several works have investigated the learned multi-dimensional index. The prevalent approach is to map the multi-dimensional data points into one dimension, and further apply a learned index on the one-dimensional space.

However, they have the following limitations: (1) The unique and critical part of multi-dimensional indexes is mapping from multi-dimensional space to one-dimensional space, while this is not learned or well-learned. Most methods [28, 35] exploit an existing SFC, such as z-order [25], as it possesses good proximity-preserving capabilities. However, a fixed SFC does not necessarily work the best on a given dataset instance. Other works, such as Flood [26], only learn to select a special dimension and then follow the fixed row-major orders on the rest of the $d-1$ dimensions, hence missing the opportunity to better preserve local proximities. (2) The physical

Proceedings of the VLDB Endowment, Vol. 16, No. 10 ISSN 2150-8097.
doi:10.14778/3603581.3603598

layout of the linearized data points, which are stored as disk pages, is not fully optimized. Existing methods typically pack a fixed amount of data points into each page, which may cause the Minimum Bounding Rectangles (MBRs) of the resulting pages to contain much dead space or heavily overlap with each other [32]. (3) Even if the above two issues can be mitigated at index construction time, existing query processing methods still need to visit many pages because their MBRs or z-address ranges *inevitably* overlap with that of the query.

In this paper, we provide a thorough and rigorous investigation of learned multi-dimensional indexes and propose LMSFC to address the above limitations. (i) It is a challenging task to formulate a suitable family of parameterized SFCs that can be efficiently learned and possess salient properties for efficient query processing. For this, we design a learnable monotonic SFC family. Based on the proposed SFC family, we devise an effective solution based on the SMBO to learn an optimal/sub-optimal SFC to adapt to different data distributions and workloads. (ii) We study the physical storage optimization issue of packing multi-dimensional data points into size-limited disk pages in a principled fashion. Thanks to the linearization due to the (learned) SFC, we are able to solve the *otherwise* NP-hard problem optimally by dynamic programming. We further propose a heuristic algorithm that trades the optimality for improved practical speed, which is suitable for large-scale datasets. (iii) In addition to the above offline optimization techniques, we further exploit the unique online query optimization opportunity by proposing a query splitting strategy. We demonstrate that our learned SFC lends itself to an efficient algorithm of splitting the query into two such that the total access to false negative data pages is minimized; this algorithm is then extended to allow multiple splits via recursion.

The contributions of the paper are summarized below:

(1) As far as we are aware, LMSFC is the *first* work to consider learning a space-filling curve that is directly optimized for the least cost query processing on a given instance of the dataset and query workload.
(2) Based on the learned SFC, we propose both offline and online optimization techniques. For the offline optimization, we can optimally or sub-optimally pack multi-dimensional data points into pages that minimize a density based cost function. For the online optimization, we propose recursively splitting the query into sub-queries such that it minimizes the access to pages that do not contain any potential data points for a query.
(3) We compare LMSFC with previous state-of-the-art multi-dimensional indexes in our experimental evaluation. LMSFC achieves the best query performance on three real-world datasets under varying query selectivity, data size, and query aspect ratio. LMSFC can achieve up to 38.2×, 7.2×, and 2.0× speedup against R^*-tree, ZM-index, and Flood, respectively.

The rest of this paper is organized as follows. In Section 2, we define the problem setting and introduce some key notations and concepts. Then we review the related literature in Section 3. Section 4 starts by investigating the requirement of range query processing for an SFC-based index and motivates a parameterized SFC family. We then overview the proposed LMSFC index based on the parameterized SFC family and introduce key steps in its construction in Section 5. We motivate and introduce our query processing method with the novel query splitting strategy in Section 6. Section 7 presents our experimental results and analyses. Finally, we conclude the paper and discuss a few extensions in Section 8.

2 PRELIMINARIES

2.1 Problem Definition

In this paper, we mainly focus on exact query processing of window queries on a multi-dimensional dataset.

Definition 1 (Multi-dimensional Dataset). *A multi-dimensional dataset D consists of n points in a d-dimensional Euclidean space. Each point $x \in D$ can be denoted as $(x^{(1)}, \cdots, x^{(d)})$ where $x^{(i)} \in \mathbb{R}^d$ is the i-th dimensional value of x.*

Without loss of generality, we assume that, with proper scaling, the domain of each coordinate is an integer within $[0, 2^K - 1]$, hence every dimension value can be represented using K binary bits.

A multi-dimensional window is a hyper-rectangle in the d-dimensional space, or formally as $w = [x_L^{(1)}, x_U^{(1)}] \times \cdots \times [x_L^{(d)}, x_U^{(d)}]$, where $x_L^{(i)} \leq x_U^{(i)}$.

Definition 2 (Multi-dimensional Window Query). *Given a multi-dimensional dataset D, a multi-dimensional window query q with the window constraint $q.w$ returns the set of points x from D that is located inside the window $q.w$, i.e. $R(q) = \{ x \mid x \in D \wedge \forall 1 \leq i \leq d, q.w_L^{(i)} \leq x^{(i)} \leq q.w_U^{(i)} \}$.*

As a query is uniquely characterized by its query window, we will use q and $q.w$ interchangeably hereafter.

2.2 Notations

Given a multi-dimensional window w, it is uniquely characterized by (w_L, w_U), i.e., the lower-bound and upper-bound points are defined as $w_L = (x_L^{(1)}, \cdots, x_L^{(d)})$ and $w_U = (x_U^{(1)}, \cdots, x_U^{(d)})$, respectively. These apply to the window constraint of the query window of q too, and we will use the shorthand q_L to denote $q.w_L$.

Given a set of points, we define the multi-dimensionalal *Minimum Bounding Rectangle* (MBR) as the smallest window that encloses the set of points.

For a binary integer v, we denote the j-th bit of the binary representation of v as v_j. Note that j starts from 0, which corresponds to the right-most bit of v. The most significant bit of v is the bit set to 1 and with the maximum bit index. For example, let $v = (\texttt{00101101})_2$ (we use $()_2$ to represents binary strings), $v_2 = 1$ and the most significant bit of v is 5.

Table 1 lists frequently used notations.

3 RELATED WORK

Indexes are essential for processing queries on large datasets. Traditional indexes are optimized for the worst-case performance. More importantly, they miss the opportunity to exploit statistical information about the data and query workloads to optimize their index structure and physical layout. Recently, many Machine Learning-based indexes have been proposed, which achieve smaller index sizes and/or faster query processing speed. RMI [19] is a well-known

Table 1: Table of Notations

Notation	Description
D	A set of multi-dimensional data records
d	The dimensionality of D
x, $x_j^{(i)}$	A multi-dimensional data point, and the j-th bit of the i-th dimension value of x
q, q_L, q_U	A multi-dimensional window query, and its lower-bounding and upper-bounding points, respectively
f	A learned SFC's mapping function
θ	The parameters of f
K	The maximum number of bits to represent coordinate values of x

work that first notices the similarity between the exact search on one-dimensional array of non-descending values and the classic regression problem. It then proposes several instances of learned indexes, which use a complex model to predict the logical location of the targeted key and then perform a local search to fix possible errors bought by the ML model. Later works further improve the model's performance or consider other variants. For instance, PGM [8] lets the user specify the error bound a priori and then uses simpler linear regression models with optimal segmentation algorithms to construct the learned index. Fiting-tree [10] also uses a tunable error parameter to tradeoff the index size for lookup performance. Given a dataset, Fiting-tree applies a cost model to estimate the space consumption and latency to find a suitable error bound. Radix Spline [18] considers a linear spline to approximate CDF. The prefixes of the selected spline points are stored in an auxiliary radix table to accelerate the search process. ALEX [5] and LIPP [37] supports data updates. LIPP further eliminates the local search via a novel adjustment strategy to redistribute keys in each node. SOSD [17, 23] proposes some benchmarks to evaluate different learned indexes.

Existing learned index techniques cannot be directly applied to multi-dimensional data, as there is no natural ordering among multi-dimensional data points. Currently, the prevalent approach is to apply a *linearization* method to convert the problem into a one-dimensional search problem, on which existing learned indexes can then be applied. z-order [28, 35] and row-major order [22, 26] are most widely used. Other linearization methods use learned linear or non-linear mapping [21], e.g., clustering followed by the distance to the cluster center [4]. To reduce the challenges of query skewness and data correlations, Tsunami [6] extends Flood via partitioning data space and modeling conditional CDFs. Qd-tree [38] utilizes Reinforcement Learning to optimize the space partitioning. SPRIG [39] uses a spatial interpolation function to locate the search range in the grid. [30] learns a quadtree structure and applies a z-order variant on each node.

Similar to most space-filling curve-based approaches, an orthogonal aspect for learned multi-dimensional indexes is to preprocess the data using simple linear transformations or sophisticated non-linear transformations, such as the Rank Space transformation [28].

Traditional multi-dimensional indexes often recursively decompose the space into disjoint or overlapping partitions. Grid File [27], kd-tree [3] and Quadtree [9] are typical examples of the former category, and R-tree [12] and its variants [2, 16, 31] are typical for the latter category. Since R-tree [12] variants have been widely adopted in commercial systems, there are also proposals to integrate learning into R-tree. AI+R [1] employs an ML model to predict the set of leaf nodes for a window query, together with a backup R-tree. [11] aims at learning the key subtree splitting routine in R-tree construction and adapts to the problem instance. In order to reduce the search range on each leaf node, [13] embeds an ML model on the selected sort dimension to accelerate the search procedure.

There are other ways of integrating learning into database indexing. [7] uses ML models to learn a balanced space partition that preserves spatial proximity well. LIMS [33] adopts an ML-based data clustering method to solve similarity search in metric spaces.

4 A FRAMEWORK FOR LEARNED SFCS

In this section, we first summarize window query processing issues for an SFC to motivate us to design a family of parameterized SFCs that possess salient properties for query processing.

4.1 Window Query Processing with SFCs

A space-filling curve (SFC) is a method of mapping the multi-dimensional data space into the one-dimensional data space. As we assume the data points have integer coordinate values within $[0, 2^K - 1]$, this naturally leads to a regular partitioning of multi-dimensional space into 2^{Kd} possible points, or *cells* (SFCs can also be applied on a coarser granularity. E.g., Flood [26] can be deemed as using a fixed SFC on *grids*). an SFC is a bijective function f between these cells and integers within $[0, 2^{Kd} - 1]$. We call x in the multi-dimensional space the *original address* and $f(x)$ as its corresponding *z-address* with respect to an SFC f. Intuitively, as $f(x)$ is a one-dimensional integer, it specifies a way to traverse all the cells exactly once. SFCs are known for their ability to preserve the multi-dimensional proximity in the linear order [20]. Hence, they are widely used, especially in applications with a need to linearize multi-dimensional data such as images, tables and spatial data. Some well-known SFCs are Hilbert curve, Z-order curve, and Gray curve.

To answer a range query q on D, assuming that points in D have been mapped to the corresponding z-addresses, we can (i) compute the query's one-dimensional z-address range q_z, (ii) retrieve every point whose z-address falls within the interval q_z, and (iii) filter out those points that do not fall into the query window q. The tightest z-address range can be defined as: $[\min_{x \in q} f(x), \max_{x \in q} f(x)]$. However, computing these extreme values is difficult in general. In the worst case, we may need to enumerate all x within q, hence with a cost proportional to the *volume* of the query *and* beating the purpose of efficient query processing. For certain SFCs with better properties, such as the Hilbert curve, we still need to enumerate all x on the boundary of the query window, hence inducing a cost proportional to the *circumference* of the query.

Nevertheless, we identify a subclass of SFCs such that the above minimization and maximization can be computed efficiently in $O(d)$ time and hence do not depend on the size of the query.

4.2 Monotonic Space-Filling Curves

We will first define the criterion that the mapping function of an SFC is monotonic, and show that the tightest z-address range can

be computed from the lower and upper bounding points of the query window.

Definition 3 (Monotonic Function in a Multi-dimensional Space). *Let $a \preceq b$ defined as true if and only if $\forall i, a^{(i)} \leq b^{(i)}$. Then a function g is monotonic, if for all a and b, if $a \preceq b$, then $g(a) \leq g(b)$.*

Theorem 1. *If an SFC corresponds to a monotonic mapping function f, given a spatial query rectangle q, the query result $r \subseteq \{ x \mid x \in D \wedge f(q_L) \leq f(x) \leq f(q_U) \}$. In other words, the tightest z-address range of q can be efficiently computed as $[f(q_L), f(q_U)]$.*

Proof. Let $q_{\min} \stackrel{\text{def}}{=} \min\{ x \in q \}$ with respect to $\preceq$ (i.e., $q_{\min}$ is q_L). Then by definition for any $x \in q, q_{\min} \preceq x$. As f is monotonic, then $f(q_{\min}) \leq f(x)$. Similarly, we can show that $q_{\max} \stackrel{\text{def}}{=} \max\{ x \in q \}$ (i.e., $q_{\max}$ is q_U) and for any $x \in q, f(x) \leq f(q_{\max})$. □

Example 1. *Among three commonly used SFCs, only the z-order curve has the monotonic property. We give counter-examples for the Hilbert curve and the Gray curve in Figure 1.*

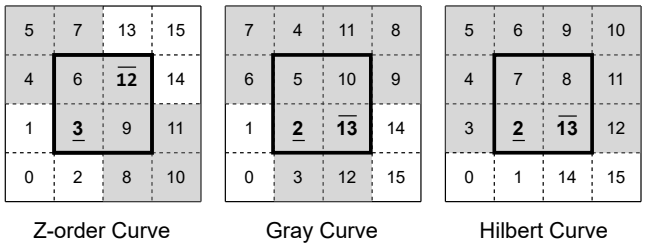


Figure 1: Hilbert and Gray Curves are not Monotonic (Bold Black Rectangle is the Query Window; the Tightest z-address Ranges are Marked in Bold Font)

4.3 Parameterized Z-Order SFCs

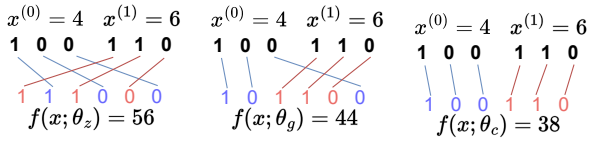
Inspired by the monotone property of the z-order curve, we identify a family of monotonic SFCs that generalizes the z-order curve. In addition, any instance within the family has favorable properties to enable efficient query processing.

We consider the following SFC parameterized by a parameter $\theta = [\theta^{(1)}, \ldots, \theta^{(d)}]$:

$$f(x;\theta) = \sum_{i=1}^{d} \sum_{j=1}^{K} \theta_j^{(i)} \cdot x_j^{(i)}, \tag{1}$$

where each $\theta^{(i)}$ is a K-dimensional vector, $x_j^{(i)}$ represents the j-th bit of the i-th dimension value of x, d is the dimensionality and K is the maximum number of bits for $x^{(i)}$. In fact, $\theta_j^{(i)} = 2^l$ indicates that $x_j^{(i)}$ will be mapped to the $(l+1)$-th bit of the binary representation of $f(x;\theta)$. If the content is clear, we use $f(x)$ for short instead of $f(x;\theta)$.

Example 2. *Figure 2 demonstrates several instances of the SFCs within our parameterized family. Figure 2(a) shows the ordered bit-interleaving way of z-order to compute $f(x)$ for the 2-dimensional data point $x = (4, 6)$. Here, $K = 3$, and its parameter $\theta_z = [\theta^{(1)}, \theta^{(2)}] = [[1, 4, 16], [2, 8, 32]]$, and the resulting z-address is 56. Figure 2(b) demonstrates a new SFC for the same data point, but with $\theta_g = [[1, 16, 32], [2, 4, 8]]$, $f(x;\theta_g) = 44$. Finally, Figure 2(c) demonstrates yet another SFC, which is known as the column-major order, with $\theta_c = [[8, 16, 32], [1, 2, 4]]$, $f(x;\theta_c) = 38$.*



(a) Original Z-order (b) Generalized Z-order (c) Column-major order

Figure 2: z-address Calculation for Several SFCs within our Parameterized SFC Family

To ensure the family of SFCs preserves the monotonic and bijective properties, it suffices to impose the following constraints on $\theta_j^{(i)}$s:

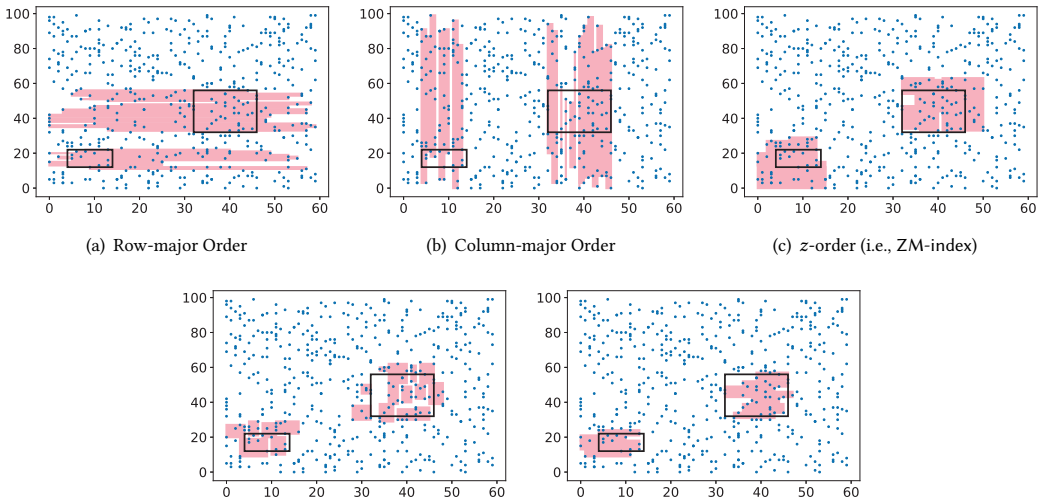
(1) $\theta_j^{(i)} \in \{ 2^0, \ldots, 2^{Kd-1} \}$.
(2) $\forall i, j, i', j', i \neq i' \vee j \neq j'$, then $\theta_j^{(i)} \neq \theta_{j'}^{(i')}$.
(3) $\forall j < j', \theta_j^{(i)} < \theta_{j'}^{(i)}$.

Both bijective and monotonic properties are *important conscious choices* in our work to balance (1) the potential of exploiting the spatial locality via learning an SFC and (2) retaining properties that facilitate efficient query processing. Violating any constraint breaks the properties of the mapping function. As an example, consider $\theta = [[1, 4], [2, 10]]$, where the first constraint is violated. Take $f(x) = 8$. There is no point that can be mapped to this z-address, indicating that the mapping function is injective but not bijective. In another example, when $\theta = [[1, 4], [2, 2]]$, the second constraint is not satisfied. Consider data points $A = (1, 1)$ and $B = (1, 2)$, then $f(A) = f(B) = 3$. Therefore, the mapping function is not bijective. Lastly, when $\theta = [[1, 4], [8, 2]]$, the third constraint is violated. Given $A = (1, 1)$ and $B = (2, 2)$, then $f(A) = 9 > f(B) = 6$, but $A \preceq B$ according to Definition 3. Therefore, the mapping function is not monotonic.

We note that the Z-order curve is hence a special instance of the above monotonic SFC. Specifically, it corresponds to $\theta_j^{(i)} = 2^{(j-1)\cdot d+(i-1)}$. We also note that computing the above $f(\cdot)$ is efficient as $f(x)$ can be computed by "scrambling" the bits of x according to θ using bit operations efficiently.

As different SFCs induce different linear ordering of the data points in the dataset, they will result in different query processing costs for a given workload. This motivates our learning of a good SFC (detailed in the next Section). Below we provide an example with visualizations to demonstrate this.

Example 3. *In Figure 3, we build and visualize query processing costs on five indexes based on different linearization methods on the same dataset and query workload. We generated a set of* random *points in a 2D space, and used the two randomly generated queries (denoted as thick black boxes) as the query workload.*



(a) Row-major Order (b) Column-major Order (c) z-order (i.e., ZM-index)

(d) Learned Grid + Row-major Order (i.e., Flood) (e) Learned Monotonic SFC (i.e., LMSFC)

Figure 3: Visualizing Five SFC-based Indexes (Thick black boxes are queries, and data pages accessed during query processing are shaded in red)

Figures 3(a) and Figures 3(b) are the usual row-major and column-major order, respectively. Figure 3(c) uses the z-order, which results in the ZM-index. Figure 3(d) is the Flood, which is based on learnable grid partitioning of the space, and then orders the grids using a row-major order (with a rearranged dimension order). Finally, Figure 3(e) shows our proposed method LMSFC, which is based on a learned monotonic SFC. We shade the data points accesses during the query processing in red.[1] *As we can see from Figure 3, learned indexes, i.e., Flood and our LMSFC, access fewer data points thanks to the learning on the specific instance.*

5 LMSFC INDEX CONSTRUCTION

5.1 Overview

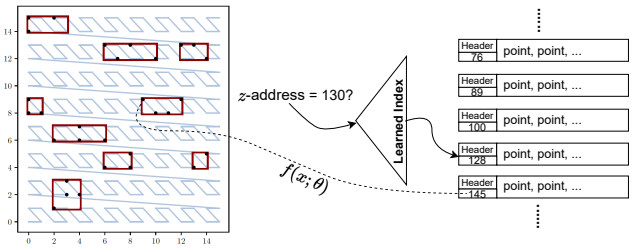


Figure 4: Overview of LMSFC Index Construction

Based on the parameterized SFC family defined in the previous section, we propose a novel multi-dimensional index, LMSFC, based on Learned Monotonic Space Filling Curves.

[1]For all the methods in the figure, we pack a fixed number of data points into each page and the shaded area are based on pages whose MBRs overlap with the query window, hence the seemingly irregular shape of the shaded areas.

We give a high-level sketch of our LMSFC method in Figure 4. In our method, we first learn a good monotonic SFC, $f(x;\theta)$ in Figure 4, which minimizes the query processing cost of the sampled query workload. The learned SFC gives us a total order for the multi-dimensional data points. We then propose both an optimal dynamic programming-based and a sub-optimal yet faster heuristic paging algorithm to load data points into pages (the thick red rectangles in Figure 4). Finally, we extract the smallest z-addresses from each page to form a sorted array, and employ a state-of-the-art learned index (e.g., pgm [8]) on that, which facilitates the lookup from z-addresses to pages.

In addition, we also include several optimizations to speed up the query processing. Specifically, we present a novel query splitting strategy to minimize the access to spurious pages due to the dimension reduction effect of the SFC mapping. We also extend the sort dimension optimization [26] to the page-level granularity.

In the rest of this section, we focus on the three steps (i.e., learning an optimal SFC, Cost-based Paging and Page-level Sort Dimension) in this section and leave the query processing related techniques to the next Section.

5.2 Learning an Optimal SFC

The goal of learning a parameterized SFC is to find the θ^* such that the resulting query processing cost is minimized, i.e., we formulate this as an optimization problem as:

$$\theta^* = \arg\min_{\theta} \mathbf{E}_{q\sim Q}[\![QueryTime(D, q; \theta)]\!] \tag{2}$$

where Q represents the distribution of the query workload, q is an i.i.d. sample from the distribution, and $QueryTime()$ function returns the actual query execution time on the given dataset.

To optimize Equation (2), we can first apply the finite sample approximation, i.e., by taking sampled queries from $\boldsymbol{Q}$ and replacing the expectation with the sample average. Nevertheless, the $QueryTime$ function is hard to model or approximate accurately as it includes various complex optimizations. Furthermore, directly evaluating the $QueryTime$ function incurs exorbitant costs.

Also, note that θ is a high-dimensional discrete parameter (with constraints), the number of choices of θ (and hence $f(\cdot)$) is exponential both in the dimensionality d and in K, rendering it impossible to solve the optimization problem via brute force in practice.

Lemma 1. *The number of different monotonic SFCs for d-dimensional space is $\Omega(d!)^K$.*

Due to the discrete nature, gradient descent style algorithms, used in previous learned index work [26], cannot be applied to solve this optimization problem either. Furthermore, the finite-sample approximate also introduces a small yet avoidable noise to the optimization.

In view of the above challenges, we adopt the state-of-the-art Bayesian optimization algorithm, SMBO [15], to approximately solve the above optimization problem and return a high-quality SFC that has a low average query time. SMBO builds a surrogate model to approximate the relationship between the parameters and the actual objective function value.

The learning process follows the standard SMBO learning framework. We use the Random Forest model as the surrogate model instead of the typical Gaussian Process model, which improves the learning speed and does not rely on the multi-dimensional Gaussian assumption and the choice of the kernel function. One of the advantages of using a surrogate model is that it is cheap to evaluate while also capturing the approximation noise in the objective function. Firstly, the surrogate model is built between the initial candidates and their performance based on the $QueryTime$ evaluation results. We evaluate the objective function by building the index on sampled datasets to save the evaluation time. By default, we conservatively use the 5% sampled dataset, which can maintain both low query time and learning cost. More learning process experiments are presented in Section 7.9. Then, the SMBO algorithm uses an acquisition function (e.g., the Expected Improvement [15]) computed on the surrogate model to suggest other candidates for evaluation in the next iteration by automatically balancing exploitation and exploration. The surrogate model gets updated during each iteration. Finally, we choose the candidate with the least cost in terms of the objective function.

5.3 Cost-based Paging

In order to accommodate external I/Os and allow for extra optimizations[2], we need to perform *paging*, which partitions the dataset D into multiple pages. As usual, we assume that each page has a maximum size of B bytes and must satisfy a min fill factor constraint specified by $f \in (0, 1]$. [3] That is, the number of bytes used in each page must be within the range of $[fB, B]$ bytes.

Finding optimal paging for multi-dimensional dataset is NP-hard [38], hence existing multi-dimensional indexing methods usually perform paging based on some heuristics, e.g., R-tree and its variants used the heuristics to minimize the margin, dead space and overlap area of the MBRs of the resulting pages [2, 31]. Not surprisingly, this practice is inherited by multi-dimensional indexes based on SFCs. For example, RSMI [28] simply loads the maximum number of points into each page, which we term as *fixed-sized paging*.

We observe that paging is important and actually can be solved *optimally* for SFC-based multi-dimensional indexes. This is because we can record the MBRs of the data points within each page and use the MBR to further optimize the query processing. On one hand, a page can be skipped if a page's MBR is disjoint with a query; on the other hand, if a page's MBR is contained in a query, we can process the data points on the page sequentially without other filtering overhead. In both cases, such optimizations are more likely if the MBR of a page is small. Default one-dimensional paging methods, such as the fixed-size paging method, are not aware of the MBR of the pages and cannot perform active optimizations for it.

Based on the above observations, we design a scoring function $S(P)$ that is intuitively the density of a page P, or $S(P) = \frac{vol(P)}{size(P)}$, where $vol(P)$ and $size(P)$ gives the volume of MBR of the page P and the number of data points in the page P, respectively.

We then formulate the *optimal cost-based paging* problem as finding a paging solution, i.e., a partitioning $P \stackrel{\text{def}}{=} \{P_1, \ldots, P_{k(P)}\}$ over D (where $k(P)$ denotes the number of resulting pages), such that the total score of the solution P is minimized, i.e.,

$$P^* = \arg\min_{P} \sum_{j \in \{1,\ldots,k(P)\}} S(P_j), \text{ subject to } size(P_j) \in [fB, B]$$

In the following, we first give an algorithm to solve the above problem optimally based on Dynamic Programming (DP), and then give a sub-optimal but fast heuristic paging algorithm. Both methods achieve a better paging layout than the fixed-sized paging and hence improve the query performance.

5.3.1 Dynamic Programming Paging Method. Thanks to the SFC which provides a linear order for the data points, we are able to circumvent the NP-hardness of the multi-dimensional paging problem by solving the one-dimensional paging problem optimally via dynamic programming.

Let $OPT[i]$ be the optimal cost obtained by an optimal cost-based paging algorithm for the first i data points. Then we can derive the following recurrent equations:

$$OPT[i] = S(Page(D[1\,..\,i])) \qquad , i < \frac{fB}{4d}$$
$$OPT[i] = \min_{s \in [\frac{fB}{4d}, \frac{B}{4d}]} (OPT[i-s] + S(Page(D[i-s+1\,..\,i])) \qquad , \text{otherwise}$$

where $Page(z)$ denotes the page formed by a set of points denoted as z and we assume each integer takes 4 bytes. Obviously, $OPT[n]$ gives the cost of the optimal paging for the entire dataset, and it is easy to use backtracking to report the optimal paging solution P^*. The time complexity of the dynamic programming paging method is $O(\frac{nB}{4d})$ as the scoring function S can be computed in $O(1)$ time via incremental computation.

5.3.2 Heuristic Paging Method. Although the DP algorithm is linear in n, it is still time-consuming in practice as B is typically a large constant (e.g., $B = 8192$ in our experiment). There, we further

[2] e.g., optimizations based on the MBR and/or the sort dimension of the pages.
[3] Technically, we do allow at most one page to occupy less than fB bytes.

propose a heuristic paging method, which can achieve comparable query performance and faster construction time compared with the DP method.

The heuristic algorithm is a greedy packing algorithm, which packs as many data points into the current page as possible until some condition is violated. The condition stipulates that the new MBR (formed by adding the current data point into the page) should *not* enlarge the old MBR by more than α times ($\alpha > 1$ is a hyper-parameter). This condition reduces the chance that the MBR of the resulting page becomes too large (with respect to the number of data points within), where a large MBR may cause much dead space and increase the chance of intersecting with the queries.

5.4 Page-level Sort Dimension

Following Flood [26], we maintain the points in each page sorted in a chosen dimension named *sort dimension*. Unlike Flood where the sort dimension is fixed for *all* pages, we allow using different sort dimensions in different pages, which provides more skipping opportunities to filter as many irrelevant points as possible when processing intersecting pages. This is because different sort dimensions may result in various sizes of the search area after refinement. Thus, we can choose the sort dimension that can achieve the least search cost for each page. In a similar vein, we utilize the query workload information to choose the sort dimension for each page as follows: for each page, we collect the set of intersecting queries in the query workload. We estimate the query cost using each of the d dimensions as the sorting dimension, and choose the one with the least query cost. If there are no intersecting queries for a page, we use a default order, which is determined in the same way as Flood.

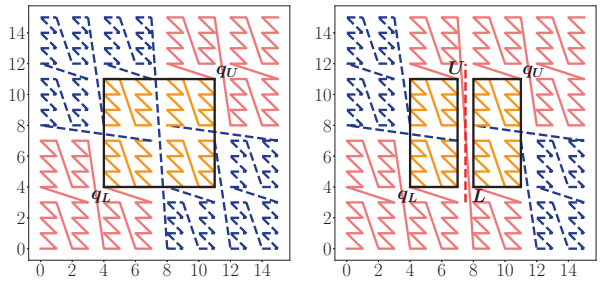
Once we have determined the sort dimension for each page, we order the data points in the page by increasing order of the sort dimension. When we need to scan a page, we can first refine the search range (or physical storage range) according to the corresponding sort dimension. Specifically, given a window query q, the range constraint over sort dimension d^* is $q_L^{(d^*)} \le x^{(d^*)} \le q_U^{(d^*)}$. The points in each page are stored contiguously in increasing order of the corresponding sort dimension. Thus, we can use binary search or a one-dimensional index model to accelerate the search via finding the lower-bound position of $q_L^{(d^*)}$ and the upper-bound position of $q_U^{(d^*)}$ in the physical storage range. As a result, points that do not satisfy the range constraint on the sort dimension are filtered out, which reduces the scanning overhead.

Besides, sort dimension can also reduce the computation cost during verification. Once we determine the search range, we can guarantee the points in this range are satisfied the query's constraint over the sort dimension. Therefore, there is no need to verify the value on the sort dimension, resulting in saved computation overhead.

6 QUERY PROCESSING

As alluded to in Section 4, to answer a query q using an SFC-based index, it takes two steps:

(1) Projection: given a spatial query rectangle q, we determine its scan range in the z-addresses as $[f(q_L), f(q_U)]$ according to Theorem 1.



(a) Learned Z-order w/o query splitting (b) Learned Z-order with query splitting

Figure 5: Example of query partition (black rectangle is a query window, yellow part only contains relevant points within the search range, blue part only contains irrelevant points within the search range)

(2) Scan with filtering: All pages whose z-addresses fall within the range need to be scanned. We can locate these pages using the forward index. Additionally, as we maintain the MBR and page-specific sort dimension information for each page, we can perform folklore optimizations such as skipping irrelevant pages, and scanning only the relevant portion of the page.

The main limitation of the above framework is that it ignores the potentially large numbers of false positive data points within the z-address range due to the SFC mapping.

Example 4. *Consider the query (the black rectangle) in Figure 5(a). The corresponding z-address range can be decomposed into the blue parts (false positives) and the yellow parts. Scanning the whole range, even with filtering, incur much unnecessary overhead in accessing and filtering pages that contain only false positive points.*

This phenomenon was also observed in a few methods such as UB-Tree [29], and a lazy skipping strategy was used. In this strategy, instead of scanning all the pages in the z-address range, it invokes a skipping function, FindNextZaddress, after scanning the current page, to compute the next page that contains the *first* true positive point with respect to q. While this strategy is guaranteed to skip *all* the false positive *pages*, it has the following drawbacks: (i) It incurs significant overhead as this function has to be invoked for every true positive page. FindNextZaddress, technically, will compute the next z-address (which may be virtual and does not correspond to any data point) after that of the last point in the current page, and translate it to a page. This will require accessing the forward index. As we employ a learned index, this incurs a non-trivial overhead for model estimation and local search. Even if we employ a B^+-tree, it may require accessing internal or leaf pages of the index. (ii) The skipping function may only skip very few pages; in fact, in many cases, it will just return the next page.

Instead, we propose a novel proactive skipping strategy based on query splitting, which is especially efficient on monotonic SFCs such as ours. We illustrate its idea in the following example.

Example 5. *Consider the same query (the black rectangle) in Figure 5(a). We can split the query into two parts by cutting it at the value 8 on the x-axis, as illustrated in Figure 5(b). We still plot the*

false positive parts in blue. It reduces the number of false positive parts compared with the case without query splitting — e.g., the part $[4, 7] \times [12, 15]$ *is eliminated.*

6.1 Recursive Query Splitting

We start by introducing a procedure to compute the best way to split a query into exactly two sub-queries (i.e., optimal 1-split), and then we generalize it to obtain multiple sub-queries based on recursive splitting.

Optimal 1-Split Algorithm. Consider a query q that corresponds to a z-address range $[f(q_L), f(q_U)]$. Without loss of generality, assume that we split at the value v on the δ-th dimension. This will split the query into two sub-queries, with the corresponding z-address ranges as $[f(q_L), f(U)]$ and $[f(L), f(q_U)]$, respectively (See Figure 5(b)). We define the *cost* of the split as $f(U) - f(q_L) + f(q_U) - f(L)$, or intuitively, the sum of the z-address ranges of the two resulting sub-queries. This cost function is chosen as it is highly correlated with the actual query processing cost after the split and can be easily computed *without* accessing the data points.

Then we formulate the *optimal 1-split problem* as finding the split (i.e., the dimension and the value) such that the cost of such split is the minimum.

Or formally,

$$(\delta^*, v^*) = \underset{\delta \in [1,d], v \in [q_L^{(\delta)}, q_U^{(\delta)}]}{\arg\min} f(U) - f(q_L) + f(q_U) - f(L)$$

Note that both U and L are determined by δ and v, but we omit the notational dependency for the easy of exposition.

As $f(q_L)$ and $f(q_U)$ are constants for the fixed query q, the above minimization is equivalent to the following maximization problem, i.e., finding the maximum "gap" between $f(U)$ and $f(L)$:

$$\underset{\delta \in [1,d], v \in [q_L^{(\delta)}, q_U^{(\delta)}]}{\arg\max} f(L) - f(U)$$

Plugging in the definition of $f(\cdot)$, it becomes:

$$\underset{\delta \in [1,d], v \in [q_L^{(\delta)}, q_U^{(\delta)}]}{\arg\max} \sum_{i=1}^{d} \sum_{j=1}^{K} \theta_j^{(i)} \cdot (L_j^{(i)} - U_j^{(i)}) \tag{3}$$

For a fixed $\delta \in [1, d]$, we can find the optimal split value v^* as:

$$\begin{aligned} v^* &= \underset{v \in [q_L^{(\delta)}, q_U^{(\delta)}]}{\arg\max} \sum_{j=1}^{K} \theta_j^{(\delta)} \cdot (L_j^{(\delta)} - U_j^{(\delta)}) + C \\ &= \underset{v \in [q_L^{(\delta)}, q_U^{(\delta)}]}{\arg\max} \sum_{j=1}^{K} \theta_j^{(\delta)} \cdot (L_j^{(\delta)} - U_j^{(\delta)}) \end{aligned} \tag{4}$$

where $C = \left(\sum_{i \neq \delta}^{d} \sum_{j=1}^{K} \theta_j^{(i)} \cdot (L_j^{(i)} - U_j^{(i)}) \right)$ is a constant.

Lemma 2. *A solution to the optimization problem of Equation 4 is* $(q_U^{(\delta)} >> l) << l$*, where* l *is the most significant bit of* $q_L^{(\delta)} \oplus q_U^{(\delta)}$*, where* $\oplus$ *denotes* XOR.

Therefore, we can solve the optimal 1-split problem by finding the optimal cut value for each of the d dimensions, hence, the complexity is only $O(d)$.

We note that Lemma 2 holds because of the fact that $U^{(\delta)} + 1 = v = L^{(\delta)}$ and the fact that $\theta_{j+1} \geq 2\theta_j$ (derived easily from the constraints introduced to guarantee the monotonic property). If the monotonic property does **not** hold, then one may need to check every possible v value to perform the optimization, hence taking $O(\|q_U - q_L\|_1)$ time complexity, which means the resulting procedure may be more expensive for "large" queries.

Example 6. *Consider the example in Figure 5(a) again. The query* $q = [4, 11] \times [4, 11]$*, and the learned SFC corresponds to the parameter*

$$\theta = [[2^0, 2^3, 2^5, 2^7], [2^1, 2^2, 2^4, 2^6]]$$

Hence, q's z-address range is $[f(q_L), f(q_U)] = [48, 207]$*. Our optimal 1-split algorithm will first consider the x-axis. In this case, the most significant bit* $l = 3$ *as* $(0100)_2 \oplus (1011)_2 = (1111)_2$*. Then the* v^* *on the axis is* $(1011)_2 >> 3 << 3 = (1000)_2 = 8$*, and then the cost of splitting at 8 on the x-axis can be calculated.*

Recursive Splitting. As one split is often insufficient to reduce the number of irrelevant pages, we adopt our optimal 1-split algorithm recursively to divide the query window into multiple parts. In our implementation, the stopping condition is set as either reaching a recursion depth of k_{maxsplit} or when there is no gap to split. k_{maxsplit} is a parameter that can balance the number of index accesses with the skipping opportunity of disjoint pages. A higher k_{maxsplit} can effectively filter out disjoint pages but causes more index access overhead. Conversely, a lower k_{maxsplit} saves the cost on index access but may not eliminate enough irrelevant pages.

7 EXPERIMENTS

7.1 Experimental Settings

Datasets We use three real-world datasets with different characteristics in our experiments (See Table 2) and they are also used in the previous work. We preprocess the datasets to scale up all coordinates to integers and remove duplicates. **OSM** is a spatial dataset consisting of 250M records randomly sampled from North America in the OpenStreetMap dataset[4]. We use the GPS coordinates (i.e., longitude and latitude) to form a 2D dataset. **NYC** is randomly sampled from records of yellow taxi trips in New York City in 2018 and 2019[5]. We used the pick-up locations, trip distances, and total amounts to form a 3D dataset. **STOCK** consists of daily historical stock prices from 1970 to 2018[6]. We select four features: the high price, the low price, the adjusted close price, and trading volume.

Table 2: Dataset Characteristics

	n (#-of-Points)	d (#-of-Dimensions)	Size (GB)
OSM	250M	2	1.95
NYC	30M	3	0.35
STOCK	30M	4	0.47

Query Workload As the datasets do not come with their query workloads, we generate the default query workloads as follows.

A query is parameterized by its center and its range in every dimension. We generate query centers in one of the two modes:

[4] https://download.geofabrik.de/
[5] https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page
[6] https://www.kaggle.com/ehallmar/daily-historical-stock-prices-1970-2018

(i) *Skewed*, in which query centers are randomly sampled data points. (ii) *Uniform*, in which query centers are randomly sampled within the data space. The width of the queries in each dimension is uniformly sampled from zero to the width of the data space of that dimension scaled by 0.05. All query windows are clipped to be within the data space.

Following [36], the final query workload is obtained by mixing 90% of the skewed queries with 10% uniform queries. The resulting selectivities for the three datasets are about 0.7%, 0.07%, 0.01%, respectively. For each dataset, we generate a training and a test query workload of sizes 1000 independently. All queries use the COUNT aggregate function, i.e., reporting the number of data points within the query window.

Algorithms We compare our proposed LMSFC with the following algorithms:

- **ZM-index** [35] combines the fixed z-order curve and learned index, together with the fixed-size paging.
- R^***-tree** [2] is a traditional and widely-used multi-dimensional index. We use the implement R^*-tree in the Boost C++ Libraries (https://www.boost.org).
- **Flood** [26] is another state-of-the-art learned index for multi-dimensional data, which learns an optimal configuration of multi-dimensional grids for a given data and query workload. It can be approximately viewed as a variable-size SFC that follows row-major order for an appropriate permutation of the d dimensions. We adapt the Flood method with fixed-size paging for a fair comparison.

We note that these algorithms represent prior state-of-the-art indexes in different categories. For example, Flood has outperformed other multi-dimensional indexes, such as Grid File [27], kd-tree [3], UB-tree [29] and Hyperoctree [24], in their experimental evaluation. We do not consider other learned multi-dimensional indexes such as LISA[22] , RSMI[28], as the original codes have various limitations or did not achieve competitive performance in our experiments.

[30] is another related work, which learns a quadtree and applies a z-order variant on each node to adapt query workload. One difference is that LMSFC learns the mapping between data points and addresses, so we can directly locate the search range rather than traversing a tree structure. In addition, we use a BO algorithm to learn better ordering. Another difference is that we directly use the actual query time as the metric rather than the number of false positive points used in [30]. Since the performance of [30] is even worse than the baseline model (i.e., ZM-index), we do not include it in our experiment.

We experimented with Tsunami[6] but did not report its performance here, because its splitting algorithm does not result in any split in any dimension on our query workloads[7], in which case, Tsunami's performance degrades to that of Flood.

We use C++ to implement all the methods. To compare these methods fairly, we run all the experiments either in the in-memory mode (for those that do not support external I/O) or in the warm buffer mode (for those that support external I/O). The page size is set to $B = 8192$ bytes, and the min fill factor $f = 0.25$. For the one-dimensional space mapped from all SFCs (z-order or learned SFCs), we use 64 bits, and $K = \lfloor \frac{64}{d} \rfloor$. And we empirically choose $k_{\text{maxsplit}} = 4$ in recursive query splitting. For more details on the implementation of the proposed algorithms, please refer to the extended version of this paper[8], which includes the relevant pseudo-code and experiments for data updates.

For learned indexes, we use the PGM [8] as a one-dimensional learned index since PGM achieves the competitive range query performance in the static dataset and is easily embedded in the different learned multi-dimensional indexes. The error bound in PGM is empirically set to 128, which is robust for different configurations and datasets.

All the experiments are performed on a machine with i9-7900X CPU @ 3.30GHz and 64 GB main memory running Ubuntu 20.04.4.

7.2 Query Performance

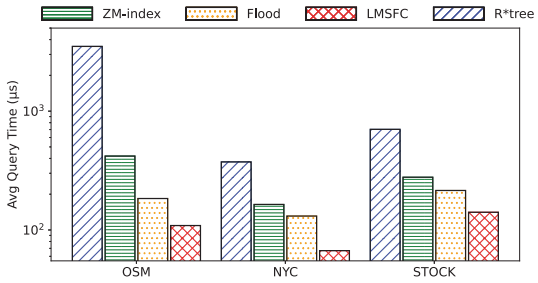


Figure 6: Query Performance

In this section, we compare LMSFC with a traditional multi-dimensional index and other learned multi-dimensional indexes for multi-dimensional range queries.

Figure 6 reports the query time for different indexes on each dataset. LMSFC outperforms all other indexes across all the datasets. LMSFC achieves between 1.52× and 1.96× speedup on query time compared with the runner-up. The main advantage of LMSFC over the baseline (i.e., the Z-order curve) is that the learned SFC preserves multi-dimensional locality better after the mapping into the one-dimension address space. Consequently, close-by points in the multi-dimensional space are more likely assigned into the same page. This leads to pages with smaller/compact MBRs, hence fewer page accesses when answering range queries. We note that LMSFC is much faster than the ZM-index, achieving 3.8× speedup on the OSM dataset. This demonstrates the huge potential of a learned SFC versus a fixed SFC as this is the key difference between the two indexes. All learned multi-dimensional indexes are significantly superior to the traditional multi-dimensional index R^*-tree. This confirms that there is a need to incorporate ML-based methods into database components to improve the performance.

In addition, we further investigate false positive (FP) records scanned by each index. In OSM dataset, the number of scanned FP points by R^*-tree, ZM-index, Flood and LMSFC are 60940, 72291, 25947, and 19067 separately per query. ZM-index scans more FP points than R^*-tree since R^*-tree utilizes a heuristic method to achieve good clustering during packing. By using a query workload as prior knowledge, Flood and LMSFC can adapt their structures to

[7]The skewness of our workload is based on data distribution while Tsunami is not.

[8]https://arxiv.org/abs/2304.12635

access fewer FP points. Besides, LMSFC applies page optimizations to further reduce FP points. Thus, LMSFC achieves the smallest number of false positive points being scanned. In the other two higher dimensional datasets, LMSFC shows superiority in this metric, which are 17.4× and 11.1× less than R^*-tree, 3.8× and 5.0× less than Flood, and 10.1× and 6.4× less than ZM-index. This is because data points are sparse in the higher dimensional data space, resulting in poor clustering during paging. Thus, paging optimization needs to be considered in multi-dimensional indexes.

7.3 Selectivity

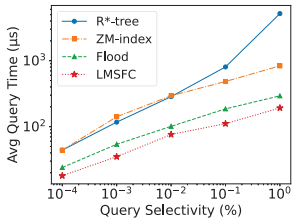


Figure 7: Varying Query Selectivity

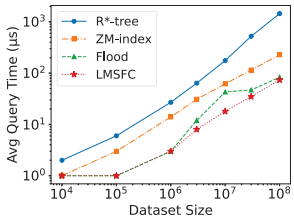


Figure 8: Varying Dataset Size

Next, we study the performances of all the methods with respect to query selectivity. We vary the selectivity from 0.0001% to 1% by uniformly scaling the query windows accordingly.

Figure 7 shows the result on the OSM dataset (and similar results are present in other datasets), where both axes are in logarithmic scales. For all the methods, the query times grow approximately linearly with the query selectivity since more data points are accessed. We notice a deterioration with the R^*-tree when the query selectivity becomes large; this may be due to the fact that when the query window grows, it is more likely to intersect with more pages, and hence more page access and backtracking.

Conceptually, learned indexes only need to scan pages within a certain z-address range, and there is no complex intersecting MBR test or back-tracking. However, R-tree and its variants need to perform more complex MBR intersection queries per inner node and may result in many back-tracking (esp., for higher dimensional cases).

ZM-index achieves similar performance with R^*-tree, but performs consistently across the selectivity range. Flood has a significant improvement consistently over ZM-index, while LMSFC further improves the query performance consistently, demonstrating the wide applicability of learned indexes.

7.4 Dataset Scalability

To investigate the scalability, we sub-sample the OSM dataset to create datasets of the same distribution but with varying sizes. Figure 8 shows the result, where both axes are in logarithmic scales. We can see that all the indexes scale approximately linearly with the data size, whereas LMSFC performs the best, followed by Flood, ZM-index, and finally R^*-tree. We also investigated the reason why Flood behaves noticeably worse than expected for 10M data points. It is partly due to the sub-optimal configuration it learned from the sampled dataset. If we allow Flood to learn from, e.g., 30% of the dataset, the resulting performance matches the approximate linear trend much more closely.

7.5 Aspect Ratio

We investigate how the performance varies with the aspect ratio of the query window. We fix the selectivity to be 1% and then vary the aspect ratio from 0.125 to 8.0. The aspect ratio is defined as the ratio of the width of two dimensions of the query window. For datasets of more than 2D (i.e., NYC and STOCK), we randomly select one dimension, called *variable dimension*, for a given dataset to enforce the aspect ratio. We then start with a query window of equal size on all dimensions, and then modify the length of the variable dimension to satisfy the ratio constraint. Finally, with the aspect ratio fixed, we scale the query window to enforce the same selectivity. For example, the three sides of a 3D query window with ratio of 4 and 0.25 will have a side length ratio of 4:1:1 and 0.25:1:1, respectively, assuming the first dimension is the variable dimension.

As shown in Figure 9, LMSFC offers the fastest query speed among all the indexes. The two learned indexes, LMSFC and Flood show much more stable performance than the other two non-learned indexes, demonstrating that learned indexes can adapt well to the query workload to achieve consistent and superior performance. Furthermore, we notice that LMSFC outperforms Flood across all settings, especially in the STOCK dataset, which is partly due to the fact that Flood has to learn a $(d-1)$-dimensional grid, which is harder for larger d. Finally, we notice that there are cases where non-learned indexes behave significantly worse even for "symmetric" aspect ratios. E.g., on the NYC dataset, R^*-tree's performance is almost 3x at aspect ratio of 8.0 as compared with that at aspect ratio of $\frac{1}{8.0}$, demonstrating the need for learned indexes for multi-dimensional datasets.

7.6 Ablation Study

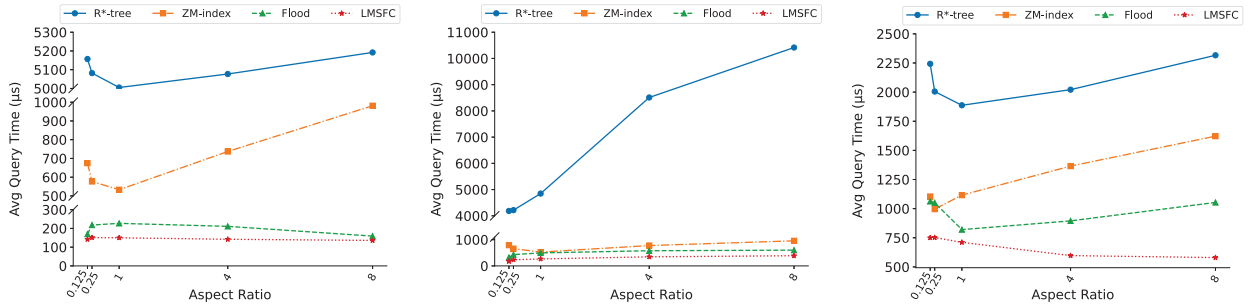
In this section, we investigate the impact of different optimization components in LMSFC on the performance by performing ablation studies.

We compared the following variants of the proposed method:

- ZM-index. This baseline uses the fixed z-order curve with a learned index, with fixed-size paging.
- LO. We replace the z-order in ZM-index by our learned SFC.
- LO + C1. On top of LO, we add the sort dimension optimization (SD).
- LO + C2. On top of LO + C1, we add the Recursive Query Splitting (RQS) optimization.
- LMSFC. This is our proposed method, which has Dynamic Programming Paging (DP) optimization added to LO + C2.

Figure 10 illustrates that adding more components can consistently and continuously improve the baseline model. Across all datasets, learned z-order (LO) almost achieves the biggest improvement on ZM-index. This is because learned z-order has the ability to adapt given query workload via optimizing.

We notice that LO + C1 and LO + C2 cannot improve too much on STOCK dataset since the multi-dimensional data is too sparse in the higher dimensionality dataset. As they still use fixed-size paging, points in each page are more scattered and thus form a much larger MBR. As a result, sort dimension optimization cannot



(a) OSM (b) NYC (c) STOCK

Figure 9: Different Aspect Ratio

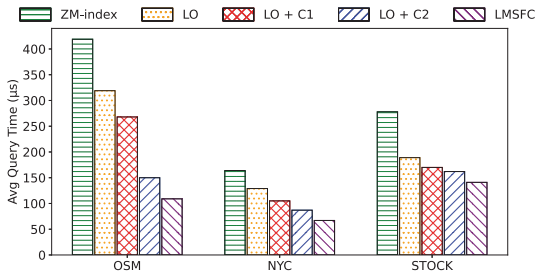


Figure 10: Ablation Study

skip too many points and more pages intersect with the query window. Note that LMSFC replaces the fixed-size paging with DP-based paging, and the above issue is alleviated, hence the noticeable performance improvement.

7.7 Query Splitting

Table 3: Recursive Query Splitting (RQS) vs FindNextZaddress (FNZ)

Model	Index Accesses	Avg Query Time (μs)
ZM-index + RQS	18	306
ZM-index + FNZ	1807	380
LMSFC + RQS	19	109
LMSFC + FNZ	1927	206

We investigate the impact of different query splitting strategies. Existing Z-order-based indexes either do not use any query splitting [26, 35] or use the query splitting method via repeated invocation of the FindNextZaddress (FNZ) function first proposed in [34] and had been used in UB-tree [29]. We name the recursive query splitting method in our proposal as the RQS.

We experiment with the two splitting strategies on both ZM-index and LMSFC and show the results in Table 3, where "index accesses" record the average number of times the forward index is accessed to perform the z-address to page lookup (See explanation in Section 6). We can see that our RQS outperforms FNZ and the improvement is especially significant for LMSFC. This is mainly because FNZ is invoked for every page that intersects the query and hence causes great overhead.

Table 4: Effect of different k_{maxsplit}

k_{maxsplit}	Avg Irrelevant Pages	Avg Query Time (μs)
0	16991	150
1	7531	126
2	4359	117
3	2478	112
4	1288	109
5	523	113

We further investigate the effects on different k_{maxsplit}. Table 4 shows we can achieve the best average query performance when $k_{\text{maxsplit}} = 4$. Although larger k_{maxsplit} can avoid scanning considerable irrelevant pages, the query performance slightly degrades. This is because the query window has been divided into too many parts, which significantly increases the overhead on accessing the learned index.

7.8 Paging Methods

Table 5: Comparing Different Paging Methods (FP, HP, and DP stands for fixed-size paging heuristic paging and dynamic programming paging, respectively)

Model	Avg Query Time (μs)	Index Size (MB)
ZM-index + FP	419	6.8
ZM-index + HP	330	8.5
ZM-index + DP	309	8.8
LMSFC + FP	150	6.8
LMSFC + HP	116	7.4
LMSFC + DP	109	7.7

We investigate different paging methods on the OSM dataset, and the results are shown in Table 5. For both ZM-index and LMSFC, DP paging shows the best query performance since DP can minimize our score function, which intuitively corresponds to relatively densely packed pages. This helps to reduce the dead space within

the page as well as decrease the probability of overlapping with the queries. Note that HP is only slightly worse than DP, but typically with much faster packing time (e.g., 35 seconds for HP versus 546 seconds for DP).

7.9 Index Learning Process

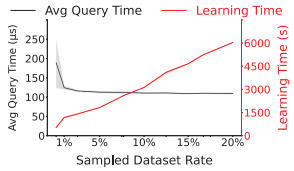


Figure 11: Varying Dataset Size

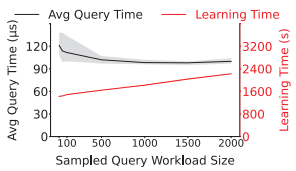


Figure 12: Varying Workload Size

Learning a good SFC using the entire dataset and query workload is infeasible due to the prohibitively long time on sorting the dataset according to the given SFC. Thus, we use sampled datasets and workloads in the learning process, which can significantly reduce the learning cost without significantly degrading the query performance. Figure 11 and Figure 12 illustrate the learning cost and query performance on OSM (other datasets show a similar trend) via varying different sample dataset sizes and query workloads sizes over several trials (minimal and maximum result is shaded). In Figure 11, when we sample a small portion of the dataset, the performance has a large variance. Although a larger sample rate can achieve better performance, the learning process is quite long. Thus, adopting a 2.5%-7.5% sampled rate is good enough to maintain both fast query time and low learning cost. Even if we reduce the sample rate to 0.5%, we can still achieve better performance than Flood but incur only half the learning time.

Based on the 5% sampled dataset, we conduct the experiment on varying query workload sizes to observe whether a large workload size can achieve better query performance. The results are displayed in Figure 12. When a workload size is larger than 500, we can achieve robust performance.

7.10 Index Size and Index Construction

Table 6: Index Size (MB)

	OSM	NYC	STOCK
R^*-tree	26.7	9.8	8.9
Flood	0.9	0.2	0.4
ZM-index	6.8	1.6	2.6
LMSFC	7.7	2.0	4.4

In Table 6, we report the index sizes for the three datasets. All the index sizes are small relative to the respective data size. Our proposed LMSFC has a larger index size than ZM-index or Flood partly because we have optimized page layouts so that pages are not fully filled. Nonetheless, our index size is still acceptable as it is still significantly smaller than the traditional R^*-tree.

We present the index construction times in Table 7. For learned indexes, we further distinguish the learning time and the index building time. R^*-tree suffers from high index construction time in the large dataset because its construction requires optimizing some criteria (e.g., dead space, margin, the overlap between two pages' MBR) for each page. ZM-index has the fastest construction time as there is no learning or optimization involved. Flood has faster learning and building times than LMSFC, because (i) Flood's model is simpler in that the hyper-parameter space is much smaller than ours. In addition, it also optimizes against a learned cost model, hence the hyper-parameter search is faster. (ii) Our LMSFC also includes other optimizations (such as dynamic programming-based paging), hence affecting the index building time. Nonetheless, the index construction is done once for a dataset.

Table 7: Index Learning and Construction Times (Seconds)

	OSM	NYC	STOCK
R^*-tree	9651	708	864
ZM-index	35	5	5
Flood Learning	73	121	431
Flood Building	44	6	10
LMSFC Learning	1821	672	879
LMSFC Building	546	87	117

Similar to Flood, if we can collect the training examples from history, we can train an offline cost model to select the learned SFC with low overhead. We use history instances as training examples to fit a neural network then we freeze the parameters of the model. During learning SFC, we can utilize gradient descent to directly adjust the input to find the optimum. Consequently, the learning process only takes a few minutes and the performance is competitive with the proposed BO algorithm.

8 CONCLUSIONS AND FUTURE WORK

In this paper, we study the problem of learned indexes for multi-dimensional data based on learned space-filling curves. We devised a framework of learning a special class of space-filling curves that is amenable to efficient query processing. In addition, we perform both offline and online optimizations by optimizing the data placement into pages and query splitting to further improve the query processing efficiency. Extensive experimental results demonstrate that the proposed method outperforms both non-learned indexes (such as R^*-tree) and prior state-of-the-art learned multi-dimensional indexes (such as ZM-index and Flood) across a wide range of settings on three real-world datasets.

Although we focus on learned monotonic SFCs in this paper, our idea and methods can be easily generalized to obtain learned non-monotonic SFCs. For example, by dropping the constraints on θ, our method can learn a non-monotonic SFC. For another example, we can consider other parameterized SFC families that generalize other well-known SFCs, such as the Hilbert Curve. We leave such exploration for future work.

ACKNOWLEDGMENTS

Wei Wang was supported by HKUST(GZ) Grant G0101000028, GZU-HKUST Joint Research Collaboration Grant GZU22EG04, and Guangzhou Municipal Science and Technology Project (No. 2023A03J0003). Xin Cao was supported by ARC DP230101534.

REFERENCES

[1] Abdullah-Al-Mamun, Ch. Md. Rakin Haider, Jianguo Wang, and Walid G. Aref. 2022. The "AI+R"-tree: An Instance-optimized R-tree. In *MDM*.

[2] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. 1990. The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. In *SIGMOD*. ACM, 322–331. https://doi.org/10.1145/93597.98741

[3] Jon Louis Bentley. 1975. Multidimensional Binary Search Trees Used for Associative Searching. *Commun. ACM* 18, 9 (1975), 509–517. http://doi.acm.org/10.1145/361002.361007

[4] Angjela Davitkova, Evica Milchevski, and Sebastian Michel. 2020. The ML-Index: A Multidimensional, Learned Index for Point, Range, and Nearest-Neighbor Queries. In *EDBT*. OpenProceedings.org, 407–410. https://doi.org/10.5441/002/edbt.2020.44

[5] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, David B. Lomet, and Tim Kraska. 2020. ALEX: An Updatable Adaptive Learned Index. In *SIGMOD*. ACM, 969–984. https://doi.org/10.1145/3318464.3389711

[6] Jialin Ding, Vikram Nathan, Mohammad Alizadeh, and Tim Kraska. 2020. Tsunami: A Learned Multi-dimensional Index for Correlated Data and Skewed Workloads. *VLDB* 14, 2 (2020), 74–86. http://www.vldb.org/pvldb/vol14/p74-ding.pdf

[7] Yihe Dong, Piotr Indyk, Ilya P. Razenshteyn, and Tal Wagner. 2020. Learning Space Partitions for Nearest Neighbor Search. In *ICLR*. OpenReview.net. https://openreview.net/forum?id=rkenmREFDr

[8] Paolo Ferragina and Giorgio Vinciguerra. 2020. The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds. *VLDB* 13, 8 (2020), 1162–1175. http://www.vldb.org/pvldb/vol13/p1162-ferragina.pdf

[9] Raphael A Finkel and Jon Louis Bentley. 1974. Quad trees a data structure for retrieval on composite keys. *Acta informatica* 4, 1 (1974), 1–9.

[10] Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2019. FITing-Tree: A Data-aware Index Structure. In *SIGMOD*. ACM, 1189–1206.

[11] Tu Gu, Kaiyu Feng, Gao Cong, Cheng Long, Zheng Wang, and Sheng Wang. [n.d.]. The RLR-Tree: A Reinforcement Learning Based R-Tree for Spatial Data. ([n. d.]). https://arxiv.org/abs/2103.04541

[12] Antonin Guttman. 1984. R-Trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD*. ACM, 47–57. https://doi.org/10.1145/602259.602266

[13] Ali Hadian, Ankit Kumar, and Thomas Heinis. 2020. Hands-off Model Integration in Spatial Index Structures *(AIDB@VLDB)*.

[14] David Hilbert and David Hilbert. 1935. Über die stetige Abbildung einer Linie auf ein Flächenstück. *Dritter Band: Analysis· Grundlagen der Mathematik· Physik Verschiedenes: Nebst Einer Lebensgeschichte* (1935), 1–2.

[15] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. 2011. Sequential Model-Based Optimization for General Algorithm Configuration. In *LION (Lecture Notes in Computer Science)*, Vol. 6683. Springer, 507–523. https://doi.org/10.1007/978-3-642-25566-3_40

[16] Ibrahim Kamel and Christos Faloutsos. 1994. Hilbert R-tree: An Improved R-tree using Fractals. In *VLDB*. Morgan Kaufmann, 500–509. http://www.vldb.org/conf/1994/P500.PDF

[17] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. [n.d.]. SOSD: A Benchmark for Learned Indexes. ([n. d.]). http://arxiv.org/abs/1911.13014

[18] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. 2020. RadixSpline: a single-pass learned index. In *aiDM@SIGMOD*. ACM. https://doi.org/10.1145/3401071.3401659

[19] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In *SIGMOD*. ACM, 489–504. https://doi.org/10.1145/3183713.3196909

[20] Jonathan K. Lawder and Peter J. H. King. 2001. Querying Multi-dimensional Data Indexed Using the Hilbert Space-filling Curve. *SIGMOD* 30, 1 (2001), 19–24. https://doi.org/10.1145/373626.373678

[21] 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*. IEEE, 289–300. https://doi.org/10.1109/ICDE48307.2020.00032

[22] Pengfei Li, Hua Lu, Qian Zheng, Long Yang, and Gang Pan. 2020. LISA: A Learned Index Structure for Spatial Data. In *SIGMOD*. ACM, 2119–2133. https://doi.org/10.1145/3318464.3389703

[23] Ryan Marcus, Andreas Kipf, Alexander van Renen, Mihail Stoian, Sanchit Misra, Alfons Kemper, Thomas Neumann, and Tim Kraska. 2020. Benchmarking Learned Indexes. *VLDB* 14, 1 (2020), 1–13. https://doi.org/10.14778/3421424.3421425

[24] Donald Meagher. [n.d.]. Octree encoding: a new technique for the representation, manipulation and display of arbitrary 3-D objects by computer. *Technical Report* ([n. d.]).

[25] Guy M Morton. 1966. A computer oriented geodetic data base and a new technique in file sequencing. (1966).

[26] Vikram Nathan, Jialin Ding, Mohammad Alizadeh, and Tim Kraska. 2020. Learning Multi-Dimensional Indexes. In *SIGMOD*. ACM, 985–1000. https://doi.org/10.1145/3318464.3380579

[27] Jürg Nievergelt, Hans Hinterberger, and Kenneth C. Sevcik. 1984. The Grid File: An Adaptable, Symmetric Multikey File Structure. *ACM Trans. Database Syst.* 9, 1 (1984), 38–71. https://doi.org/10.1145/348.318586

[28] Jianzhong Qi, Guanli Liu, Christian S. Jensen, and Lars Kulik. 2020. Effectively Learning Spatial Indices. *VLDB* 13, 11 (2020), 2341–2354. http://www.vldb.org/pvldb/vol13/p2341-qi.pdf

[29] Frank Ramsak, Volker Markl, Robert Fenk, Martin Zirkel, Klaus Elhardt, and Rudolf Bayer. 2000. Integrating the UB-Tree into a Database System Kernel. In *VLDB*. 263–272. http://www.vldb.org/conf/2000/P263.pdf

[30] Yanhao Wang Sachith Gopalakrishna Pai, Michael Mathioudakis. 2022. Towards an Instance-Optimal Z-Index *(AIDB@VLDB)*.

[31] Timos K. Sellis, Nick Roussopoulos, and Christos Faloutsos. 1987. The R+-Tree: A Dynamic Index for Multi-Dimensional Objects. In *VLDB*. Morgan Kaufmann, 507–518. http://www.vldb.org/conf/1987/P507.PDF

[32] Darius Sidlauskas, Sean Chester, Eleni Tzirita Zacharatou, and Anastasia Ailamaki. 2018. Improving Spatial Data Processing by Clipping Minimum Bounding Boxes. In *ICDE*. IEEE Computer Society, 425–436. https://doi.org/10.1109/ICDE.2018.00046

[33] Yao Tian, Tingyun Yan, Xi Zhao, Kai Huang, and Xiaofang Zhou. 2022. A Learned Index for Exact Similarity Search in Metric Spaces. *CoRR* abs/2204.10028 (2022). https://doi.org/10.48550/arXiv.2204.10028

[34] Herbert Tropf and Helmut Herzog. 1981. Multidimensional Range Search in Dynamically Balanced Trees. *ANGEWANDTE INFO.* 2 (1981), 71–77.

[35] Haixin Wang, Xiaoyi Fu, Jianliang Xu, and Hua Lu. 2019. Learned Index for Spatial Queries. In *MDM*. IEEE, 569–574. https://doi.org/10.1109/MDM.2019.00121

[36] Xiaoying Wang, Changbo Qu, Weiyuan Wu, Jiannan Wang, and Qingqing Zhou. 2021. Are We Ready For Learned Cardinality Estimation? *VLDB* 14, 9 (2021), 1640–1654. http://www.vldb.org/pvldb/vol14/p1640-wang.pdf

[37] Jiacheng Wu, Yong Zhang, Shimin Chen, Yu Chen, Jin Wang, and Chunxiao Xing. 2021. Updatable Learned Index with Precise Positions. *VLDB* 14, 8 (2021), 1276–1288. http://www.vldb.org/pvldb/vol14/p1276-wu.pdf

[38] Zongheng Yang, Badrish Chandramouli, Chi Wang, Johannes Gehrke, Yinan Li, Umar Farooq Minhas, Per-Åke Larson, Donald Kossmann, and Rajeev Acharya. 2020. Qd-tree: Learning Data Layouts for Big Data Analytics. In *SIGMOD*. ACM, 193–208. https://doi.org/10.1145/3318464.3389770

[39] Songnian Zhang, Suprio Ray, Rongxing Lu, and Yandong Zheng. 2021. SPRIG: A Learned Spatial Index for Range and kNN Queries. In *SSTD*. ACM, 96–105. https://doi.org/10.1145/3469830.3470892