Towards Designing and Learning Piecewise Space-Filling Curves (Technical Report)

Jiangneng Li[1], Zheng Wang[1], Gao Cong[1], Cheng Long[1], Han Mao Kiah[1], Bin Cui[2]
[1]Nanyang Technological University, [2]Peking University
{jiangnen002,zheng011}@e.ntu.edu.sg,{gaocong,c.long,hmkiah}@ntu.edu.sg,bin.cui@pku.edu.cn

ABSTRACT

To index multi-dimensional data, space-filling curves (SFCs) have been used to map the data to one dimension, and then an one-dimensional indexing method such as the B-tree is used to index the mapped data. The existing SFCs all adopt a single mapping scheme for the whole data space. However, a single mapping scheme often does not perform well on all the data space. In this paper, we propose a new type of SFC called piecewise SFCs, which adopts different mapping schemes for different data subspaces. Specifically, we propose a data structure called Bit Merging tree (BMTree), which can generate data subspaces and their SFCs simultaneously and achieve desirable properties of the SFC for the whole data space. Furthermore, we develop a reinforcement learning based solution to build the BMTree, aiming to achieve excellent query performance. Extensive experiments show that our proposed method outperforms existing SFCs in terms of query performance.

PVLDB Reference Format:
Jiangneng Li[1], Zheng Wang[1], Gao Cong[1], Cheng Long[1], Han Mao Kiah[1], Bin Cui[2]. Towards Designing and Learning Piecewise Space-Filling Curves (Technical Report) . PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

1 INTRODUCTION

A space-filling curve (SFC) is a way to map a multi-dimensional data point $\mathbf{x}$ to a value v, which can be represented by a mapping function $T : \mathbf{x} \mapsto v$. It has been widely used for multi-dimensional indexing, and the idea is to first map multi-dimensional data points to values (which are one-dimensional) and then use those indexing methods that have been developed for one-dimensional data, such as conventional B-Tree [4] and more recent learned indexes [15], to index the mapped values. This has been exploited both in the literature [9, 10, 17, 34, 42, 45, 46] and by various database systems such as PostgreSQL [2], Amazon DynamoDB [39], Apache HBase [26], etc.

There are extensive studies on designing SFCs, such as the Z-curve [28–30], C-curve [12], and Hilbert curve [12, 13, 24, 25]. The Z-curve, for example, adopts a mapping scheme called *bit interleaving* [36], which first converts the dimensions of input data to *bit strings* (e.g., in Figure 1, it converts data point $\mathbf{x} = (2, 3)$ into its corresponding binary strings with 2 bits for each dimension: $(10_2, 11_2)$). The bit interleaving then merges bits alternatively from different bit strings to form an SFC value (in Figure 1, the bit interleaving adopts the XYXY merging scheme, which merges the bit strings XX and YY to the SFC value XYXY, e.g., mapping $\mathbf{x}$ to 1101_2).

#1 W4D1

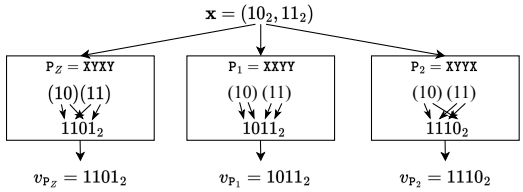


Figure 1: Bit Merging Pattern (BMP), $\mathtt{P}_Z$ (XYXY) is the BMP of Z-curve, XXYY (C-curve) and XYYX are two other BMPs.

However, one common problem is that each type of SFC has its own fixed mapping scheme/function, which cannot be adjusted to fit with different datasets. The choice of an SFC for a dataset will significantly affect the query performance, and no single SFC can dominate the performance on all datasets and query workloads. To design a new SFC to fit with the data and query workload properties, QUILTS [27] extends bit interleaving by considering other ways of merging bit strings (e.g., instead of merging bits following XYXY, we can merge bits by following XXYY or XYYX to generate different SFC values, as described in Figure 1). Each pattern of merging bits is called a *bit merging pattern* (BMP), and each BMP can describe a different SFC (refer to Section 2 for details). QUILTS evaluates all the candidate SFCs described by BMPs based on a given workload and data, and selects the optimal one using heuristic methods. Figure 2 illustrates the high-level idea of QUILTS, where it selects a curve that corresponds to the Z-curve from a set of candidate SFCs, based on a query workload and data.

QUILTS makes the first attempt to utilize data and query workload property to select an optimal SFC. However, like other SFCs, QUILTS applies a single BMP for the entire data space. Optimal SFCs may differ for different data subspaces. For example in Figure 2, Z-curve works best for queries with 2×2 blue rectangles, while C-curve is optimal for the query with a 1×4 yellow rectangle. No single SFC can achieve the best performance for both types of queries. Another issue of QUILTS is that it does not provide an effective way of generating and evaluating candidate SFCs. The heuristic rules used by QUILTS are designed for very specific types of window queries (e.g., with a fixed area) and do not fit with general query processing scenarios where a workload includes more than one query type (with different areas or aspect ratios). For example, a heuristic rule used by QUILTS assumes that grid cells intersecting with a query should be continuous in the SFC order, which may not hold for queries with different aspect ratios (which are explained in Section 3.1).

To address the limitation of an SFC with a single BMP, i.e., a single mapping scheme, our idea is to design different BMPs for different

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

subspaces based on the data and query workload property, aiming to optimize the query performance. The SFC designed according to our idea will comprise multiple BMPs, each corresponding to a subspace, and we call the resulting SFC a *piecewise* SFC. Figure 2 illustrates an example of piecewise SFCs, where we choose Z-curve for the left half subspace and C-curve for the right half subspace, thus achieving the optimal performance for both blue and yellow query rectangles (which needs 12 cell scans).

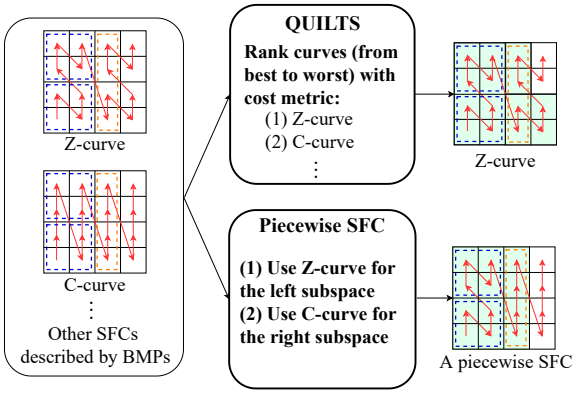


Figure 2: Comparison between QUILTS and a piecewise SFC example, where the green cells are scanned (14 for QUILTS and 12 for the piecewise SFC) for queries in the blue and yellow dashed rectangles.

To design piecewise SFCs, we address the following three challenges. First, it is a new and open problem of how to design effective BMPs for different subspaces. We propose a novel idea of seamlessly integrating the subspace partitioning and BMP generation. We develop the *Bit Merging Tree* (BMTree) to recursively generate both subspaces and the corresponding BMPs. In the BMTree, (1) each node represents a bit from one selected dimension for BMPs, and its value (0 or 1) plays the roles of partitioning data at the node into two child nodes, and (2) each leaf node represents a subspace, and the sequence of bit string from the root to the leaf node represents the BMP for the subspace.

Second, the piecewise SFC design makes it challenging to guarantee two desirable properties of the overall mapping function: monotonicity [18] and injection. Monotonicity is a desirable property for designing window query algorithms. Intuitively, monotonicity property will guarantee that the SFC values of data points in a query rectangle fall in the range of the SFC values formed by two boundary points of the query rectangle. Combining different SFCs for different subspaces to obtain a final SFC for the whole space may lead to the risk of breaking the monotonicity property. Similarly, it may also lead to the injection violation, i.e., the mapping function may not return a unique mapped value for each input. We construct the BMTree in a principled way such that the two desired properties are guaranteed (details can be found in Section 3.3).

Third, to address the limitation of heuristic algorithms in the SFC design, we propose to model building the BMTree as a Markov decision process (MDP) [33], aiming to develop data driven solutions to designing suitable BMPs for different subspaces. Specifically, we define the states, actions, and rewards signals of the MDP framework to build the BMTree such that the piecewise SFCs modeled by the BMTree can optimize the query processing performance. We leverage the reinforcement learning technique, Monte Carlo Tree Search (MCTS) [6], to learn a performance-aware policy and avoid local optimal. To improve the performance, we design a greedy action selection algorithm for the MCTS algorithm. Moreover, to improve the training efficiency, we define a metric called *ScanRange* as a proxy of the query processing performance (e.g., query latency and I/O cost) and use ScanRange for defining the reward signals.

In summary, we conclude our main contributions as follows:
(1) We propose the idea of piecewise SFCs for designing SFCs, which allows to design different BMPs for different subspaces by considering the data and query workload property. To the best of our knowledge, the idea is new in the literature.
(2) To design piecewise SFCs, we propose the BMTree to partition the data space into subspaces and generate a BMP for each subspace. We prove that the piecewise SFC represented by a BMTree satisfies two properties, namely injection and monotonicity, which are important for designing query processing algorithms.
(3) To build the BMTree, we develop an RL based solution by modeling BMP design as a decision making process, and design a MCTS based BMTree construction algorithm. We design a greedy based action selection algorithm to guide MCTS. We also develop the ScanRange metric to efficiently measure the window query performance on an SFC, which speeds up the reward computing during the learning procedure. To the best of our knowledge, this is among the first learning based approaches for designing an SFC.
(4) We integrate our learned SFCs into the classic database index B+ Tree in PostgreSQL, and compare with baseline SFCs on the querying performance under PostgreSQL. We also apply our learned SFCs to a learned spatial index RSMI [34]. Experimental results under both settings consistently show our method outperforms the baselines in terms of the query performance.

2 PROBLEM STATEMENT & PRELIMINARIES

2.1 Problem Definition

Let $\mathcal{D}$ denote a database, where each data point $\mathbf{x} \in \mathcal{D}$ has n dimensions, denoted by $\mathbf{x} = (d_1, d_2, \ldots, d_n)$. For the ease of understanding, we consider a 2-dimensional data point $\mathbf{x} = (x, y)$, and can be easily extended to n dimensions. $\mathbf{x}$ can be converted to bit strings as: $\mathbf{x} = ((x_1 x_2 \ldots x_m)_2, (y_1 y_2 \ldots y_m)_2)$. where each x_i, y_j $(1 \leq i, j \leq m)$ are 0 or 1 (i.e., $x_i, y_j \in \{0, 1\}$) and m is the length of bit string, which is dependent on cardinality of dimension x and y. Take $\mathbf{x} = (4, 5)$ for example, it can be converted to $\mathbf{x} = (100_2, 101_2)$. In previous studies on SFC based multidimensional indexes [4, 36, 38], values of data points are typically mapped to fine-grained grid cells for discretization. SFC maps $\mathbf{x}$ into a scalar value v (called SFC value) with a mapping function $T(\mathbf{x}) \rightarrow v$. An SFC value v can be used as the key value of data $\mathbf{x}$ to determine the order of $\mathbf{x}$ in $\mathcal{D}$.

Problem 1 (SFC Design). *Given a database $\mathcal{D}$ and a query workload Q, we aim to develop a mapping function T, which maps each data point $\mathbf{x} \in \mathcal{D}$ into an SFC value v, s.t. with an index structure (e.g., B+ Tree) built on the SFC values of data points in $\mathcal{D}$, the query performance (e.g., I/O cost and querying time) on Q is optimized.*

Apart from database $\mathcal{D}$, the SFC problem takes as input (1) a query workload Q and (2) an index structure. We generate window query workloads by following the previous work [27, 34] with three #1 D3

different distributions, including uniform (UNI), Gaussian (GAU), and skew (SKE) distributions. We adopt B+Tree and RSMI [21], which represent classic and learning-based index structures, respectively.

2.2 Preliminaries on SFC

We present two desired properties for a mapping function T — *injection* and *monotonicity*. We then describe the curve design methods in the Z-curve and Quilts, which also satisfy these properties.

Injection.[1] An SFC design is expected to satisfy the property named injection, which guarantees a unique mapping from $\mathbf{x}$ to v. This is to ensure that SFC value v can be used as a key value of $\mathbf{x}$ for ordering data and indexing. It is defined as follows.

Definition 2.1 (Injection). Given a function $T : \mathbf{x} \rightarrow v$, T is injective if $\mathbf{x}$ maps to a unique value v, s.t. $\forall \mathbf{x_1} \neq \mathbf{x_2}, T(\mathbf{x_1}) \neq T(\mathbf{x_2})$.

The injection property is desirable for an index to narrow the search space for better query performance. Consider an extreme situation where all data points map to the same value. Then an index based on the SFC values cannot narrow the search space for a query.

Monotonicity. The monotonicity [18] is defined as follows.

Definition 2.2 (Monotonicity). Given two n-dimensional data points (denoted as $\mathbf{x}'$ and $\mathbf{x}''$), whose SFC values are denoted as $T(\mathbf{x}')$ and $T(\mathbf{x}'')$. When a mapping function T holds monotonicity, if $d_i' \geq d_i''$ is satisfied for $\forall i \in [1, n]$, it always has $T(\mathbf{x}') \geq T(\mathbf{x}'')$.

Maintaining monotonicity is a desirable property for mapping data points to SFC values as explained below. Given a 2-dimensional window query represented by its minimum (bottom-left corner) and maximum (top-right corner) points (i.e., $\mathbf{q}_{min} = (x_{min}, y_{min})$, $\mathbf{q}_{max} = (x_{max}, y_{max})$). Let $\mathcal{P} = \{(x, y) \mid x_{min} \leq x \leq x_{max}, y_{min} \leq y \leq y_{max}\}$ denote the query results bounded by the query window. If the monotonicity property holds, the result points in $\mathcal{P}$ are within the range bounded by the SFC values of $\mathbf{q}_{min}$ and $\mathbf{q}_{max}$. This is because for any data point $\mathbf{p} \in \mathcal{P}$, whose SFC value $T(\mathbf{p})$ always holds that $T(\mathbf{q}_{min}) \leq T(\mathbf{p}) \leq T(\mathbf{q}_{max})$. The property is desirable since it enables us to design simple and efficient algorithms for processing a window query by checking data points whose SFC values are within the bounded range only; Otherwise, the algorithm does not work. For example, the Hilbert curve and its variants [12, 24, 25] do not satisfy the monotonicity property, which makes it hard to identify the scanning range for a window query in the space of their SFC values and requires maintaining additional structure to design more complicated algorithms [16].

Computing SFC values in Z-curve [28–30] and QUILTS [27]. Both Z-curve and QUILTS guarantee the injection and monotonicity properties. Figure 1 examplifies how the Z-curve and QUILTS map a data point $\mathbf{x}$ to a scalar SFC value v. The curve design in the Z-curve and QUILTS are presented as follows.

The SFC value of $\mathbf{x}$ in the Z-curve is computed via bit interleaving, which generates a binary number consisting of bits (0 or 1) filled alternatively from each dimension's bit string. The Z-curve value of a 2-dimensional data point $\mathbf{x}$ is computed by function T_z:

$$T_z(\mathbf{x}) = (x_1 y_1 x_2 y_2 \dots x_m y_m)_2 \tag{1}$$

[1]This property is defined on discretized input. No injection is guaranteed in continuous space since no bijection mapping exists between $\mathbb{R}$ and $\mathbb{R}^n$ [32].

It assumes that all dimensions have the same bit string length, and the zero-padding technique is usually applied to fit the length equally by padding zeros at the head of each bit string.

QUILTS generalizes the bit interleaving pattern of the Z-curve to more general *bit merging pattern*, each of which represents a way of merging bits. We take two-dimensional data for example, QUILTS defines a bit merging pattern as follows.

Definition 2.3 (Bit Merging Pattern). A bit merging pattern (BMP) is a string P of length $2m$ over the alphabet $\{\mathtt{X}, \mathtt{Y}\}$ s.t. it contains exactly m X's and m Y's. Given a $\mathtt{P} = p_1 p_2 \dots p_{2m}$, the SFC described by P is defined as follows. We set

$$T_{\mathtt{P}}(\mathbf{x}) = (b_1 b_2 \dots b_{2m})_2 \tag{2}$$

according to the following rule: (1) Since P contains exactly m X's, we let $I = \{i_1, \dots, i_m\}$ be the list of ordered indices such that $p_{i_\ell} = \mathtt{X}$. Then we set $b_{i_\ell} = x_\ell$ for $1 \leq \ell \leq m$. (2) Similarly, for the value of y, we consider $J = \{j_1, \dots, j_m\}$ where $p_{j_\ell} = \mathtt{Y}$, and assign b_{j_ℓ} the bit value of y_ℓ. For example, given the BMP $\mathtt{P} = \mathtt{XXYY}$, the value of data point $\mathbf{x}$ computed by $T_{\mathtt{P}}$ is $T_{\mathtt{P}}(\mathbf{x}) = (x_1 x_2 y_1 y_2)_2$. Notice that both x and y are subsequences of $T_{\mathtt{P}}(\mathbf{x})$.

SFCs represented with different BMPs form an SFCs set. QUILTS considers this set and selects the optimal SFC evaluated on a given query workload as the output curve. We prove the monotonicity property of SFCs with BMPs, which guarantees the monotonicity property of our method in Section 3.5.

Lemma 2.4 (Monotonicity of SFCs with BMPs). *An SFC with a BMP achieves the monotonicity property.*

Proof. Given a BMP P, we prove the monotonicity property in 2-dimension space which is easy to extend to n. Given $\mathbf{x} = (x, y)$ and $\mathbf{x}' = (x', y')$ satisfying $x \geq x'$ & $y \geq y'$, $T_{\mathtt{P}}(\mathbf{x}) = b$ and $T_{\mathtt{P}}(\mathbf{x}') = b'$. (1) If $\mathbf{x} = \mathbf{x}'$, then $T_{\mathtt{P}}(\mathbf{x}) = T_{\mathtt{P}}(\mathbf{x}')$; (2) if $\mathbf{x} \neq \mathbf{x}'$, assume $x > x'$. There is a smallest indice $s(x, x')$ such that $x_{s(x,x')} = 1 > x'_{s(x,x')} = 0$ while $x_i = x'_i$ when $0 < i < s(x, x')$. (i) If $y = y'$, assume P put $x_{s(x,x')}$ to bit b_t of T_P, we have $b_t = 1 > b'_t = 0$ while $b_j = b'_j$ for all $0 < j < t$, $T_{\mathtt{P}}(\mathbf{x}) > T_{\mathtt{P}}(\mathbf{x}')$. (ii) If $y > y'$, we pick the one from $x_{s(x,x')}$ and $y_{s(y,y')}$ which maps to the smaller index in $T_{\mathtt{P}}$, the condition in (i) still satisfies and $T_{\mathtt{P}}(\mathbf{x}) > T_{\mathtt{P}}(\mathbf{x}')$. Same proof applied when $x = x'$ & $y > y'$. With (1) and (2), lemma proved. □

3 PROPOSED SOLUTION

3.1 Motivations and Challenges

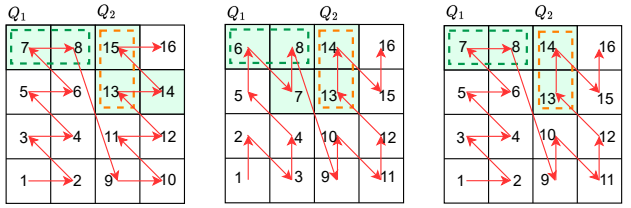
Motivation 1: Piecewise SFC design. QUILTS and earlier SFCs based on BMPs only use one BMP to compute SFC values for all data points, which may not perform well for query processing.

Example 3.1. Figure 3 shows a 4×4 grid space, where the green and yellow dashed rectangles represent two window queries Q_1 (horizontal) and Q_2 (vertical), respectively. The red lines represent the ordering of grid cells w.r.t. three SFCs. Take SFC-1 for example, whose $\mathtt{P}_1 = \mathtt{XYYX}$ and the computed value for input $\mathbf{x} = ((x_1 x_2)_2, (y_1 y_2)_2)$ is $T_{\mathtt{P}_1}(\mathbf{x}) = (x_1 y_1 x_2 y_2)_2$. Note that in SFC-1, x_1 is put as the first bit in the combined bit string, and thus any data point with $x_1 = 0$ (which resides in the left half of Figure 3 (a)) will have a smaller mapped value than any data point with $x_1 = 1$ (which resides in the right half of Figure 3(a)). We label the grid

ids based on the mapped values of grid cells computed by the SFC curves. As discussed in Section 2.2, a typical algorithm first locates the grid ids on the minimum (bottom-left corner) and the maximum (top-right corner) points of a query window.

Different SFCs will result in accesses of different grid cells for answering the two window queries Q_1 and Q_2. (1) With SFC-1, to answer query Q_1, we scan the range from the minimum point (grid 7) to the maximum point (grid 8), resulting in 2 grid scans. For Q_2, the grid ids for the minimum and maximum points are 13 and 15, respectively. Hence we need 3 grid scans ranging from grid 13 to 15. (2) With SFC-2 ($\mathtt{P}_2$ = XYXY), we need 3 grid scans (from grid 6 to 8) for Q_1 and 2 grid scans (from grid 13 to 14) for Q_2.

In the example, SFC-1 performs better for Q_1 while SFC-2 is better for Q_2. A natural idea is whether we can combine the advantages from the two BMPs of SFC-1 and SFC-2, i.e., we use XYYX to organize the data at the left hand side and XYXY to organize the data at the right hand side. The design will result in a *piecewise* SFC, shown as SFC-3 in Figure 3(c). With SFC-3, we need 2 grid scans for both Q_1 and Q_2, where the scanning ranges for Q_1 and Q_2 are from grid 7 to 8 (similar to SFC-1), and grid 13 to 14 (similar to SFC-2). This example motivates the need of designing a piecewise SFC.



(a) SFC-1 ($x_1\,y_1\,y_2\,x_2$) grid scans: 2 for Q_1, 3 for Q_2

(b) SFC-2 ($x_1\,y_1\,x_2\,y_2$) grid scans: 3 for Q_1, 2 for Q_2

(c) SFC-3 (piecewise SFC) grid scans: 2 for Q_1, 2 for Q_2

Figure 3: Motivation for piecewise SFC, SFC-1 is described by the BMP XYYX while SFC-2 by XYXY. In contrast, SFC-3 (ours) is described by two BMPs: left by XYYX and right by XYYX, where the green shade highlights the scanned grids .

Motivation 2: Learning based method for SFC design. Classic SFCs (Z-curve, Hilbert curve, etc.) are based on a single bit merging pattern and fail to utilize database instance to design the SFC. In contrast, QUILTS proposes to utilize the given database and query workload to evaluate and select an SFC from an SFC set in which each SFC is described by a BMP. However, QUILTS does not directly evaluate SFC w.r.t. query performance but uses heuristic rules to generate candidate SFCs. The heuristics rules will select BMPs such that the resulted grid cells intersecting with a query would be continuous in the curve order, which achieves fewer grid scans. These heuristics only work for query workload containing limited types of window queries (e.g., with the same aspect ratio), and are not effective under general situations (where more than one query type with different aspect ratios and region areas exist). Due to these limitations, it calls for more principled solutions to utilize database and query workload for generating and selecting an SFC, and learning-based methods would be promising for the purpose.

Challenges. Piecewise SFC design brings up three main challenges as discussed in Introduction. (1) How to partition the space and design an effective BMP for each subspace? The piecewise SFC design needs to consider both space partitioning and BMP generation. (2) How to design piecewise SFCs such that two desirable properties, monotonicity and injection, hold? Combining different SFCs for different subspaces to obtain a piecewise SFC for the whole space may lead to the risk of breaking the properties. For instance, two data points with distinct BMPs in a piecewise SFC may end up with identical SFC values. (3) How to design a data driven approach to build the BMTree, given a database and query workload?

3.2 Solution Overview

Bit Merging Tree (BMTree) for Piecewise SFC Design. To address the first challenge, we propose a novel way of seamlessly integrating the subspace partitioning and BMP generation by building the BMTree, a binary tree models a piecewise SFC. Each node of BMTree is filled with a bit from a dimension. The filled bit partitions the space into two subspaces corresponding to two child nodes. The left branch is the subspace where data points have a bit value of 0 and the right branch with 1. The BMTtree partitions the whole data space into subspaces, each corresponding to a leaf node with its BMP being the concatenated bit sequence from the root to the leaf node. We present the BMTree structure in Section 3.3.

Furthermore, the BMTree mechanism guanratees that the generated piecewise SFC satifies the two properties, which addresses the second challenge. We prove the piecewise SFC represented by a BMTree satisfies both monotonicity and injection in Section 3.5.

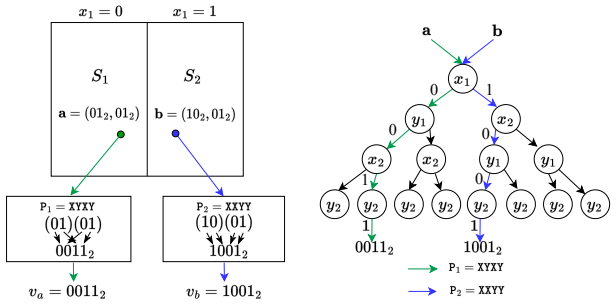
RL based Algorithm for Constructing a BMTree. To address the third challenge, we design a learning-based method that learns from data and query to build the BMTree. We model the building of BMTree as a Markov decision process [33]. The process of building a BMTree comprises a sequence of actions to select bits for tree nodes with a top-down order. To learn effective policy for building the BMTree, we propose a new approach to integrating a greedy policy into the Monte Carlo Tree Search (MCTS) framework [6]. Specifically, we develop a greedy policy that is used to select a action to fill a bit for each node to build a tree. For each node, the greedy policy chooses the bit that achieves the most significant reward among all the candidate bits. Afterwards, we apply the greedy policy as a guidance policy and use MCTS to optimize the BMTree with the objective of providing good query performance and avoiding local optimal. We present the proposed solution in Section 3.4 and the time complexity analysis in Section 3.5.

3.3 Bit Merging Tree (BMTree)

We proceed to present how to develop a piecewise SFC, modeled by Bit Merging Tree (BMTree), which is a binary tree.

Designing a BMP. To design a BMP P, we need to decide which character (X or Y in the two-dimensional case) is filled in each position of P. A left-to-right design procedure decides the filling characters in the order from p_1 to p_{2m}. The key to the BMP design is to have a policy deciding which dimension (X or Y) to fill into each position of P.

Designing piecewise SFC with multiple BMPs. We next discuss piecewise SFC design. As discussed in Section 3.1, one challenge of designing a piecewise SFC is how to tackle two subtasks that are mixed together, namely subspace partitioning and BMP design for each subspace. It is also challenging to guarantee that the piecewise SFC comprising different BMPs for different subspaces still satisfies



(a) A piecewise SFC. **(b) The BMTree.**

Figure 4: (a) An example of a piecewise SFC, which comprises two BMPs P_1 and P_2 for computing values of data points a and b. (b) A BMTree that combines the two BMPs.

both injection and monotonicity properties. To address the challenges, we propose a novel solution to simultaneously generating the subspaces and designing BMPs for subspaces.

We follow the left-to-right BMP design, and start with an empty string P. For example, if we fill X in the first position of P, bit x_1 will be filled to b_1 position of P; Then the whole data space is partitioned into two subspaces w.r.t. the value of bit x_1, where one subspace corresponds to $x_1 = 0$ and the other corresponds to $x_1 = 1$. This partitioning enables us to separately design different BMPs for the two subspaces. Note that the BMPs for each subspace will share X as the first character, but can have distinct filling choices for the next $2m - 1$ characters. By recursively repeating this operation, we fill in the subsequent characters for each BMP for each subspace, thus generating multiple subspaces each with a different BMP. An elegant perspective of our idea is that we integrate the subspace partitioning and BMP generation seamlessly.

Example 3.2. An example of piecewise SFC is given in Figure 4a, where dimension x and y are bit strings of length 2. First, X is selected, and then the whole space is partitioned w.r.t. value of bit x_1 into two subspaces where subspace S_1 corresponds to $x_1 = 0$ and $x_1 = 1$ is for subspace S_2. Next, we separately design BMPs for S_1 and S_2, where all BMPs under S_1 share the first bit $x_1 = 0$ and BMPs under S_2 share the first bit $x_1 = 1$. We generate two example BMPs: $\mathsf{P}_1 = \mathtt{XYXY}$ for S_1 and $\mathsf{P}_2 = \mathtt{XXYY}$ for S_2, which is the BMP of C-curve. Finally, we get a piecewise SFC that comprises T_{P_1} for S_1 and T_{P_2} for S_2. This piecewise SFC represents the function: $T(\mathbf{x}) = \begin{cases} (x_1y_1x_2y_2)_2 & \text{if } x_1 = 0 \\ (x_1x_2y_1y_2)_2 & \text{if } x_1 = 1 \end{cases}$. Therefore, if a data point **a** is located in S_1, we will apply T_{P_1} to compute SFC value; otherwise, if data **b** is in S_2, T_{P_2} is applied.

To facilitate the process of designing piecewise SFCs, we propose the Bit Merging Tree (BMTree) structure, which is used to simultaneously partition the space and generate BMPs. Figure 4b shows the corresponding BMTree for the example piecewise SFC in Figure 4a. Since the example piecewise SFC is developed with only 2 BMPs, the left subtree of the root node shares P_1 while the right subtree shares P_2. Next, we present the BMTree.

Bit Merging Tree (BMTree). A BMTree is a binary tree modeling a piecewise SFC T_{T}, and is denoted by T. The depth of a BMTree T equals the length of a BMP, denoted by $2m$ for the 2-dimensional space. Every node of T corresponds to a bit of x_i or y_i, $1 \le i \le m$. The left (resp. right) child denotes the subspace with bit value 0 (resp. 1). Each path from the root node to a leaf node represents a BMP for the subspace of the leaf node, which is the concatenation of all the bits of the nodes in the path. The SFC value $T_{\mathsf{T}}(\mathbf{x})$ of a data point **x** is computed by traversing a path of T as follows. We start from the root node, and for each traversed node, denoted by x_i, if $x_i = 0$, we visit the left child node; otherwise go to the right child. When we reach a leaf node, the corresponding BMP of the traversed path is used to compute for $T_{\mathsf{T}}(\mathbf{x})$. The green path in Figure 4b is the path traversed for point **a**, which represents BMP P_1 while the blue path traversed for **b** represents P_2.

Algorithm 1: BFS BMTree Construction Algorithm

input : Decision Policy π;
output: Constructed T;
1 Initial queue $H.push(N_{root})$; /* push root node */
2 **while** $H \neq \emptyset$ **do**
3 $\quad N \leftarrow H.pop()$;
4 $\quad \mathcal{B} = \{(d_i, ind_i)$: all bits available to $N\}$;
5 $\quad (d_j, ind_j) \leftarrow \pi(\mathcal{B})$;
6 $\quad$ Assign (d_j, ind_j) to N;
7 $\quad H.push(N.left, N.right)$; /* add children */
8 **end**
9 **return** T

To construct a BMTree, we develop a breadth-first construction algorithm (Algorithm 1) to fill bits to BMTree's nodes. The algorithm fills bits into BMTree level by level, consistent with the left-to-right BMP design. Under a n-dimensional space, we note (d_i, ind) for the bit in dimension d_i with index ind. In the algorithm, a queue H is initialized with the root node (line 1). The algorithm iteratively pops node N from the queue (line 3). We use $\mathcal{B}$ to record the bits that can be used to fill N (line 4), where bit d_i is available if less than m d_i's have been filled to N's corresponding BMP. Then a learned policy (to be introduced in Section 3.4) will decide which bit to fill (line 5). After filling a bit for N (line 6), the child nodes of N will be pushed into H if N is not a leaf node (line 7).

3.4 MCTS based BMTree Construction

This subsection is to present our solution to Line 4 of Algorithm 1, i.e., learning a decision policy. It is difficult to design heuristic methods to construct the BMTree to optimize the querying performance for a workload on a database instance. This could be observed from QUILTS that uses the heuristic rules for workload containing specific types of window queries only, and fails to directly optimize query performance. In contrast, we propose a reinforcement learning (RL) based method for learning a decision policy that builds the BMTree to optimize the querying performance directly.

To allow an RL policy to construct the BMTree, we model the BMTree construction as a Markov decision process (MDP). Then, we design a BMTree construction framework with a model-based RL method named Monte Carlo Tree Search (MCTS). Unlike traditional algorithms such as greedy or A*, MCTS is an RL approach that demonstrates superior exploration-exploitation balance, reducing the issue of local optimum. MCTS is well-suited for our problem, #1 R2W3D4

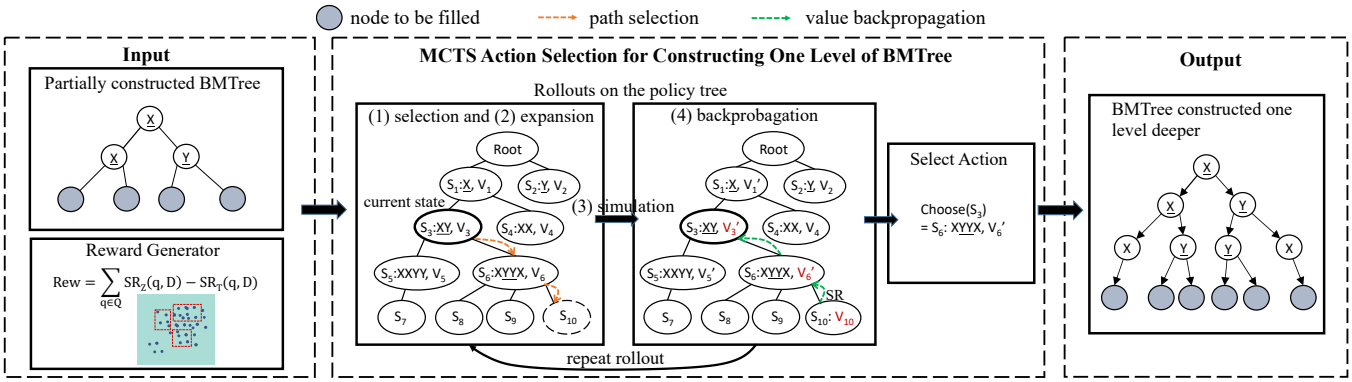


Figure 5: Workflow of Monte Carlo Tree Search Based BMTree construction.

and offers stable performance without extensive parameter tuning, compared with other RL algorithms such as PPO [37].

Figure 5 shows the workflow of the MCTS based BMTree construction framework. We define one action that RL takes to be a series of bits that fill a level of nodes in the BMTree, and the nodes of the next level are then generated. The action space size exponentially grows with the node number. It becomes difficult for RL to learn a good policy with an enormous action space size. To address this, we design a greedy action selection algorithm, which helps to guide MCTS to search for good actions. Moreover, we design a metric named ScanRange to speed up reward computing.

BMTree construction as Decision Process. We proceed to illustrate how we model the BMTree construction as a MDP in detail, including *states*, *actions*, *transitions* and *reward* design.

• States. Each partially constructed BMTree structure T is represented by a state to map each tree with its corresponding query performance. The state of a BMTree is represented with the bits filled to the BMTree's nodes. For example, in Figure 5, the current (partially constructed) BMTree's state is represented as $\mathsf{T} = \{(1:\underline{\mathsf{X}}), (2:\underline{\mathsf{XY}})\}$, where $\underline{\mathsf{X}}$ and $\underline{\mathsf{XY}}$ are bits filled to nodes in level 1 and level 2.

• Actions. Consider a partially constructed BMTree T that currently has N nodes to be filled. We define the actions as filling bits to these nodes. We aim to learn a policy that decides which bit to be filled for each node. Furthermore, the policy also decides if the BMTree will split the subspace of one tree node. If the policy decides to split, the tree node will generate two child nodes based on the filled bit b, and the action is denoted as $\underline{\mathsf{b}}$ with an underline; Otherwise, the tree node only generates one child node, which corresponds to the same subspace as its parent, the action is denoted as b. During the construction, the policy will assign bits to all N nodes. The action is represented as $A = \{a_1, \ldots, a_N\}, a_i = (\mathsf{b}_i, sp_i)$, where b_i denotes the bit for filling node n_i, and sp_i denotes whether to split the subspace. Given T with N nodes to be filled, the action space size is $(2n)^N$ where n is the dimension number, and the factor of 2 comes from the decision of whether to split the subspace.

• Transition. With the selected action A for unfilled nodes in T, the framework will construct T based on A. The transition is from the current partially constructed BMTree T to the newly constructed tree T', denoted as $\mathsf{T}' \leftarrow Transition(\mathsf{T}, A)$. In our framework, we start from an empty tree, and construct the BMTree level by level during the decision process. Each time the action generated by the policy will fill one level of BMTree nodes (starting from level 1) and generate nodes one level deeper.

• Rewards Design. After T is transited into T', we design the reward that reflects the querying performance of T' to evaluate the goodness of action A. One might consider executing queries using the corresponding BMTree to see how well the SFC helps to decrease the I/O cost. However, it is time-consuming. To this end, we propose a metric named *ScanRange (SR)*, which reflects the performance of executing a window query and can be computed efficiently. We construct the reward based on the SR of T'.

Computing Rewards Efficiently. *SR* is calculated as follow. Given a BMTree T, we randomly sample data points from $\mathcal{D}$ with a sampling rate r_s. Then, the sampled data points are sorted according to their SFC values. To compute SFC values on a partially constructed BMTree, we apply a policy extended from the Z-curve to the unfilled portions of the BMP in each subspace. Sorted data points are then evenly partitioned into $\frac{r_s|\mathcal{D}|}{|B|}$ blocks, where $|B|$ denotes # of points per block. For a given window query q represented by its minimum point $\mathbf{q}_{min}$ and maximum point $\mathbf{q}_{max}$, we first calculate the SFC value of the minimum (resp. maximum) point as $v_{min} = T_{\mathsf{T}}(\mathbf{q}_{min})$ (resp. $v_{max} = T_{\mathsf{T}}(\mathbf{q}_{max})$). Then, the blocks that v_{min} and v_{max} fall into are denoted as ID_{min} and ID_{max}. We calculate the *SR* of q given T and $\mathcal{D}$ as $SR_{\mathsf{T}}(q, \mathcal{D}) = ID_{max} - ID_{min}$. The calculation of SR is much cheaper than executing queries.

We develop a reward generator based on the defined *SR*. We take the performance of the Z-curve as a baseline. Given the dataset $\mathcal{D}$ and a query workload $\mathcal{Q}$. The generator sorts the data points based on their SFC values, and compute the reward as:

$$Rew = \sum_{q \in \mathcal{Q}} (SR_{\mathsf{Z}}(q, \mathcal{D}) - SR_{\mathsf{T}}(q, \mathcal{D})) \tag{3}$$

Intuitively, the reward is positive if the BMTree constructed by the policy achieves a lower SR than the Z-curve. This aligns with our objective to minimize the SR. We normalize the reward by dividing the reward of the Z-curve.

Example 3.3. We give an example of how the decision process works in Figure 5. The partially constructed BMTree is represented with the bits filled to different levels, denoted by $\mathsf{T} = \{(1:\underline{\mathsf{X}}), (2:\underline{\mathsf{XY}})\}$ where each tuple is the bits filled to the corresponding BMTree level. The learned policy selects the action $A = \underline{\mathsf{XY}}\mathsf{X}$. The next level of the BMTree is constructed based on A. The reward signal

is computed based on the performance of the one level deeper constructed BMTree. The BMtree will continue to be input for building the next level.

We proceed to present the proposed MCTS framework, including a BMTree T under construction, a policy tree that keeps updating, and a reward generator that generates the reward based on T.

Policy Tree. MCTS [6, 47] is a model-based RL method. The high-level idea of MCTS is to search in a tree structure, where each node of the tree structure denotes a state. Given the current state, the objective of MCTS is to find the optimal child node (i.e., the next state) that potentially achieves an optimal reward. The tree structure is named *policy tree* [47], and we define it as follows:

Definition 3.4 (Policy Tree). The policy tree is to model the environment. Each node of the policy tree corresponds to a state (or a partially constructed BMTree). Moreover, every node stores: (1) action A transits the parent BMTree to itself and (2) a reward value that reflects the goodness for choosing the node. The root node of the policy tree corresponds to an empty BMTree, and each path of the policy tree from the root node to the leaf node corresponds to a decision procedure of constructing a BMTree.

Rollouts. To choose an action, MCTS first checks the reward that different action choices can achieve. To achieve this, MCTS will make several attempts in which it simulates several paths in the policy tree and then checks if the attempted path results in a good performance. MCTS then updates the policy tree based on the simulations, named *rollout*. A rollout consists of four phases: (1) selection, which selects the attempted path corresponding to a BMTree construction procedure, (2) expansion, which adds the unobserved state node to the policy tree, (3) simulation, which tests the selection's the performance, and (4) backpropagation, which updates the reward value. We proceed to present our design of the four steps.

(1) Selection. The selection step aims to select a path in the policy tree that potentially achieves good performance. Starting from the current state $\mathtt{S}_t$ with the initialized path: $\mathtt{Path} = \{\mathtt{S}_t\}$, we first check if all child nodes have been observed in the previous rollouts. If there are unobserved nodes, we choose one of them and add it to the path. Otherwise, we apply the Upper Confidence bounds applied to Trees (UCT) action selection algorithm [14] to select a child node, which balances the exploration and exploitation. Specifically, UCT selects the child node with the maximum value $v_{uct} = \frac{\mathtt{V}_{t+1}}{num(\mathtt{S}_{t+1})} + c \cdot \sqrt{\frac{\ln(num(\mathtt{S}_t))}{num(\mathtt{S}_{t+1})}}$, $\mathtt{S}_{t+1} = Transition(\mathtt{S}_t, A)$, where $\mathtt{V}_{t+1}$ is the reward value of the child node $\mathtt{S}_{t+1}$ transited from $\mathtt{S}_t$ by action A; $num(\mathtt{S}_{t+1})$ and $num(\mathtt{S}_t)$ denote the times of observing node $\mathtt{S}_{t+1}$ and node $\mathtt{S}_t$ during rollouts; c is a factor defaulted as 1. The selected node $\mathtt{S}_{t+1}$ is then added to the path: $\mathtt{Path} = \{\mathtt{S}_t \rightarrow \mathtt{S}_{t+1}\}$. The selection step continues until the last node of Path is an unobserved node. It then returns the Path for the next step.

(2) Expansion. In the expansion step, the unobserved nodes in Path are added to the policy tree. The times of observing each node $\mathtt{S}_t$ in Path, denoted by $num(\mathtt{S}_t)$, is recorded for the UCT algorithm.

(3) Simulation. We simulate the performance of the selected Path by constructing the BMTree based on the actions stored in the nodes of the path. The constructed BMTree is then input to the reward generator to compute the ScanRange metric.

(4) Backpropagation. In this step, we update the value of each node in Path. We apply the maximum value update rule, which updates the value of a state $\mathtt{S}_t$ with the maximum reward it could gain from simulation, computed by $\mathtt{V}'_t = \max(\mathtt{V}_t, Rew)$, where $\mathtt{V}_t$ is the old value of state $\mathtt{S}_t$, Rew is the reward gained during the simulation and $\mathtt{V}'_t$ is the updated value.

Example 3.5. In the example of MCTS rollout in Figure 5, State $\mathtt{S}_3$ corresponds to the input partially constructed BMTree. During the rollouts, we select path $\{\mathtt{S}_3 \rightarrow \mathtt{S}_6 \rightarrow \mathtt{S}_{10}\}$ in the selection step. It then expands the new observed state $\mathtt{S}_8$ to the policy tree in the expansion step. We construct the BMTree based on the selected path and compute ScanRange. In the backpropagation step, the values of $\mathtt{S}_3$, $\mathtt{S}_6$ and $\mathtt{S}_{10}$ are then updated whose values are in red color, based on the ScanRange computed in $\mathtt{S}_{10}$.

After the rollouts procedure, the algorithm selects the action with the highest reward value, and BMTree T is then constructed correspondingly. In the example, $\mathtt{S}_6$ is selected with the largest value $\mathtt{V}'_6$ compared with other child nodes. It then returns the action XYYX as the action to build the BMTree one level deeper.

Greedy Action Selection. We design the greedy action selection (GAS) algorithm for the selection step in rollouts, which is to help MCTS find potential good action for a partially constructed BMTree. Given T with N nodes to be filled, GAS generates an action A_g by greedily assigning a bit to each BMTree node which achieves the minimum ScanRange compared with other bits when T is filled with that bit. The algorithm is given in Algorithm 2.

In GAS, we first initialize an empty action list A_g and invoke function $Node2Fill(\mathtt{T})$ to extract all unfilled nodes from T (lines 1–2). Then we select a bit for each node (lines 3–7). In each iteration, we invoke function $ExtractBits$ to extract all bits that can be used to fill a node Node. Then we select the best bit b that minimizes ScanRange (line 5), construct the BMTree w.r.t. the selected bit (line 6), and append the selected bit b to A_g (line 7). During the selection step, if the state $\mathtt{S}_g$ transited by the greedy selected action A_g has never been observed, MCTS will select $\mathtt{S}_g$ and add it to the path.

Algorithm 2: Greedy Action Selection (GAS)

```
   input  : a BMTree T, a reward generator Env;
   output : Generated action A_g for T;
1  Initial empty action list A_g = {};
2  Nodes ← Nodes2Fill(T);
3  for Node ∈ Nodes do
4      Bits = ExtractBits(Node);
5      b = argmax_{b_i} (SR(T, b_i, Env)), b_i ∈ Bits;
6      T ← Construct(T, b);
7      A_g.append(b) ;
8  end
9  return A_g
```

MCTS based BMTree construction algorithm. Algorithm 3 outlines the MCTS-based BMTree constructing algorithm. It initializes an empty BMTree (line 1) and a policy tree (line 2). Then it constructs a reward generator based on $\mathcal{D}$ and Q (lines 3–4). It then constructs the BMTree level by level (lines 5–15). It does rollouts

(lines 6–10), including the path selection based on the BMTree T', the path expansion, the reward simulation, and reward generator Env and the backpropagation which updates the corresponding node's value. The algorithm then chooses the best action A (line 12) and constructs T' based on A (line 13). If the current partially constructed BMTree is not better (line 14), we stop digging deeper. We set up a *max depth* M (line 5) and a rollout number RO (line 6).

Algorithm 3: MCTS based BMTree constructing algorithm

```
   input  : A n-dimensional dataset D and a training workload Q;
   output : Trained BMTree T;
1  Reset an empty BMTree T' ;
2  Initial the policy tree with an root state node ;
3  D_s = sample(D);
4  Construct Env based on D_s and Q;
5  for depth from 0 to M do
6      for rollout from 0 to RO do
7          Path = Selection(T');
8          Expansion(Path);
9          Rew = Simulation(T', Path, Env);
10         Backpropagation(Path, Rew);
11     end
12     A = choose(T');
13     T' ← Action(T', A);
14     T ← T' if T' performs better than T; otherwise break;
15 end
16 return T
```

3.5 Analysis and Discussion

Injection Analysis. To prove the injection property of BMTree, we consider a 2-step proof. First, we prove BMTree maps an input with only one output. Then, we prove no two different inputs will have the same SFC value.

Proof. (1) Given an input $\mathbf{x}$, the BMTree computes $T_{\mathsf{T}}(\mathbf{x})$ by traversing a path from the root node to a leaf node. Based on the observation, each $\mathbf{x}$ only have one path, which corresponds to one BMP and corresponding one value. (2) Given two inputs $\mathbf{x}$, $\mathbf{x}'$, two conditions should be considered. i) If $\mathbf{x}$, $\mathbf{x}'$ share the same path, which indicates that $\mathbf{x}$, $\mathbf{x}'$ share the same BMP. We notice that each BMP stored in the BMTree itself is injective. Thus, $\mathbf{x}$ and $\mathbf{x}'$ have the same value if and only if $\mathbf{x} = \mathbf{x}'$; ii) if $\mathbf{x}$ and $\mathbf{x}'$ have distinct paths, which indicates that $\mathbf{x}$ and $\mathbf{x}'$ are associated with different BMPs in T. Based on the BMTree, we know that paths traversed by $\mathbf{x}$ and $\mathbf{x}'$ share the first several nodes (at least the root node) until they branch at a specific node. Then one route to the left child node and the other to the right. Let d denote the depth of the branch node. Then, P is used to compute $T_{\mathsf{T}}(\mathbf{x})$ and P' for $T_{\mathsf{T}}(\mathbf{x}')$ share the same first d bits, while $T_{\mathsf{T}}(\mathbf{x})$ has a different bit value as $T_{\mathsf{T}}(\mathbf{x}')$ at the d^{th} bit, i.e., 0 for one and 1 for the other one. Based on that, $T_{\mathsf{T}}(\mathbf{x}) \neq T_{\mathsf{T}}(\mathbf{x}')$ is confirmed and thus guarantees the injection. □

Monotonicity Analysis. Given $\mathbf{x}$ and $\mathbf{x}'$ satisfying $d_i \geq d_i'$ for each dimension $1 \leq i \leq n$ and a BMTree T, we prove that $T_{\mathsf{T}}(\mathbf{x}) \geq T_{\mathsf{T}}(\mathbf{x}')$. We discuss two cases in terms of the paths of $\mathbf{x}$ and $\mathbf{x}'$, i.e., whether they share the same path on the BMTree or not.

Proof. (1) When $\mathbf{x}$ and $\mathbf{x}'$ share the same path, i.e., they correspond to the same BMP in T. Monotonicity inherits from the shared pattern (Lemma 2.4). (2) When $\mathbf{x}$ and $\mathbf{x}'$ correspond to different paths, these two inputs will share a portion of their paths and depart into different branches at the branch node. Assume the branch node is at the depth d, which means value $T_{\mathsf{T}}(\mathbf{x})$ and $T_{\mathsf{T}}(\mathbf{x}')$ have the same first d bits, and they hold for distinct value at the d^{th} bit. To show the monotonicity, i) we first show the value corresponding to the path branching to right side must be larger than the value corresponding to the path branching to left side. ii) we then show $T_{\mathsf{T}}(\mathbf{x})$ corresponding to the right path and $T_{\mathsf{T}}(\mathbf{x}')$ corresponding to the left side. This is because at the d^{th} bit, it tracks the same dimension and the same bit index from $\mathbf{x}$ and $\mathbf{x}'$, respectively. Given the condition that $x_i > y_i$ satisfied for all dimension i, we must have the bit is 1 (right) for the $\mathbf{x}$ and 0 (left) for the $\mathbf{x}'$. Therefore, ii) holds. Based on i) and ii), we conclude $T_{\mathsf{T}}(\mathbf{x}) > T_{\mathsf{T}}(\mathbf{x}')$. □

Time Complexity Analysis. We provide time complexities for SFC value computation and MCTS-based BMTree construction. The time complexity for computing the SFC value of $\mathbf{x}$ using the constructed BMTree is $O(M)$, where M is the length of $T_{\mathsf{T}}(\mathbf{x})$. This complexity is comparable to other SFCs described by BMPs.

For BMTree construction, the complexity of Algorithm 3 is $O\left(M \cdot (N + |\mathcal{D}_s|\,(M + \log|\mathcal{D}_s|) + |\mathcal{Q}|)\right)$, where N is the child node size of the policy tree, $|\mathcal{D}_s|$ and $|\mathcal{Q}|$ correspond to the size of sampled data and query workload. It takes at most M actions to construct the BMTree. In each step of choosing an action, the selection step is bounded by child node size $O(N)$; the simulation time corresponds to the computation of ScanRange, which takes $O(M \cdot |\mathcal{D}_s|)$ for SFC value computing, $O(|\mathcal{D}_s| \log(|\mathcal{D}_s|)$ to sort data, and $O(|\mathcal{Q}|)$ to compute ScanRange for each query.

Handling updates. In the presence of insertion, the BMTree can still be applied as the mapping function to compute SFC values for new data points. In the experiment, we observe that the query performance on indexes built on the SFC values from the BMTree is insensitive to moderate shifts in data distribution and query distribution. However, in the case of a sharp change in query distribution, a retraining of the BMTree is recommended to maintain its performance. #3 W1

Optimizing other queries. This paper focuses on optimizing window queries. However, kNN queries can also be included in the optimization objective as part of the workload. We will empirically evaluate if window queries and kNN queries can be optimized together.

4 EVALUATION

The experiment aims to evaluate: (1) proposed piecewise SFC method vs. existing SFCs when applied for SFC-based indexes vs. other indexes, (2) BMTree under different settings (e.g., varying data, query size, distribution shift, dimensionality, aspect ratio), (3) components of BMTree by evaluating different BMTree variants, and (4) suitability of ScanRange (*SR*) as an I/O replacement. #2 D2

4.1 Experimental Setup

Datasets. We conduct experiments on both synthetic and two real datasets. For synthetic datasets, we generate data points in the 2-dimensional data space with a granularity size of $2^{20} \times 2^{20}$, which

follow either uniform distribution (denoted as UNI) or Gaussian distribution (denoted as GAU). Real data OSM-US contains about 100 millions of spatial objects in the U.S. extracted from OpenStreetMap API [1], and TIGER [3] contains 2.3 millions of water areas in north America cleaned by SpatialHadoop [8].

Query workload. We follow the work [27] to generate query workloads. We generate different types of window queries, and each type of queries has a fixed area selected out of $\{2^{30}, 2^{32}, 2^{34}\}$ and a fixed aspect ratio selected out of $\{4, 1, 1/4\}$; Each workload comprises multiple types of queries, which have different combinations of areas and ratios. In addition, we generate query with different distributions by following work like [7, 34], including the uniform distribution (denoted as `UNI`) and the Gaussian distribution (denoted as `GAU`). We also generate the skewed workload (denoted as `SKE`), in which queries follow Gaussian distributions with different μ values.

Index structures. To evaluate the performance of the proposed piecewise SFC compared with the existing SFCs, we integrate piecewise SFC and baseline SFCs into both traditional indexes and learned index structures. First, we integrate piecewise SFC (and baseline SFCs) into the PostgreSQL database system and a built-in B+ Tree variant in PostgreSQL is employed with SFC values as key values. Second, we use a learned spatial index, RSMI [21], to compare the performance of piecewise SFC and baseline SFCs when they are used in RSMI. Here, the B+ Tree of PostgreSQL is a disk based index and the released implementation of RSMI [21] is memory based. We choose them to evaluate the performance of piecewise SFC under different scenarios. We also combine BMTree into ZM [42], another SFC-based learned index, to further demonstrate BMTree's applicability.

SFC Baselines. We choose the following SFC methods as our baselines. (1) Z-curve [24, 38]; (2) Hilbert Curve [25]; (3) QUILTS [27].

Evaluation metrics. For experiments conducted with PostgreSQL, we use the I/O cost (IO) recorded by PostgreSQL system and Query Latency (QL). For experiments under RSMI, we report the node access number of its tree structure and QL for a fair comparison by following [11, 21].

Table 1: Experiment Parameters.

Parameters	Value
Data	GAU **UNI** OSM-US TIGER
Query	`GAU` **`SKE`** `UNI`
Sampling rate	0.01 0.025 **0.05** 0.075 0.1
# Training Q	100 500 **1000** 1500 2000
Max depth	1 5 **10** 15 20

Parameter settings. Table 1 lists the parameters used in our experiments, and the default settings are in bold. We set the rollout number in MCTS at 10 by default. The max depth is the depth of BMTree built via the RL model; the sampling rate (0.05 by default) is the rate of sampling training data for computing the ScanRange.

Evaluation platform. We train the BMTree with PyTorch 1.9, Python 3.8. The experiments are conducted on an 80-cores server with an Intel(R) Xeon(R) Gold 6248 CPU@2.50GHz 64.0GB RAM, no GPU resource is leveraged to train the model. The datasets and code are available via the link[2] to reproduce our work.

[2]https://github.com/gravesprite/Learned-BMTree

4.2 Experimental Results

4.2.1 Effectiveness study. This experiment is to compare the effectiveness of the learned piecewise SFC in query processing with other SFCs under both PostgreSQL and RSMI environments. We also compare optimized SFC-based index with other indexes. For each experiment, we use 1000 windows queries, which are randomly generated by following respective distributions for training, and another 2000 different window queries, which are generated by following the same distribution, for evaluation.

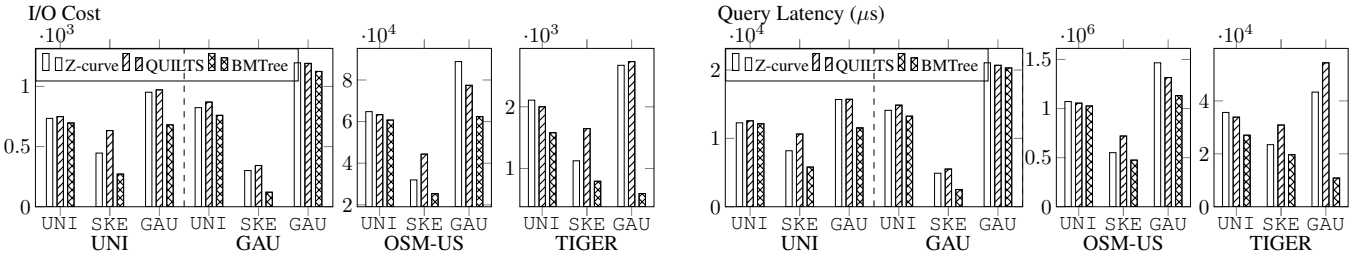
Results on PostgreSQL. Figure 6a and Figure 6b show the IO and QL on window queries. To ensure PostgreSQL conducts `indexscan` during querying, both the `bitmapscan` and `seqscan` in PostgreSQL are disabled. We do not include the Hilbert curve for this experiment since the Hilbert curve requires additional structure and dedicated algorithm for returning accurate results for window queries, and PostgreSQL does not support them for the Hilbert curve.

We observe that the proposed piecewise SFC modeled by BMTree consistently outperforms the baselines in all the combinations of data and query distributions in terms of both IO and QL. Between the two baselines, QUILTS performs worse for `SKE` workload and performs similarly as the Z-curve for `UNI` workload and `GAU` workload. This is because our query workload contains queries with very different aspect ratios (e.g., 4 and 1/4), which does not meet the requirement of QUILTS' algorithm. QUILTS will only choose queries with particular ratio to optimize and thus results in bad performance among other queries. We also conducted experiments where the queries are with similar ratio, and QUILTS performs better than Z-curve. BMTree outperforms Z-curve by 5.2%–39.1% (resp. 7.7%–59.8%, 6.3%–29.8% and 25.1%–77.8%) in terms of I/O cost on UNI (resp. GAU, OSM-US and TIGER) dataset across different types of workloads. The results in terms of QL are consistent with those of IO. BMTree's superior performance is because (1) BMTree is able to generate piecewise SFCs to handle distinct query distribution , and (2) BMTree is equipped with effective learning technique to generate BMPs and subspaces. We notice that under the `UNI` workload BMTree outperforms Z-curve by 25.1% on TIGER while it only outperforms Z-curve slightly on the other three datasets. This is as expected: Under a uniform query workload, BMTree can only make use of data distribution, but not query distribution, to optimize the performance; TIGER is very skewed and BMTree can capture the skewed data feature of TIGER.

Results on RSMI. The original RSMI [34] uses the Hilbert curve, and we include it as a baseline for this experiment as RSMI returns approximate results for all curves. All the curves achieve comparable recall (99.5% or above) using RSMI's algorithms for window queries. Figures 7a and 7b show the node access number and QL for all the curves using RSMI. We observe that BMTree consistently outperforms all the baselines. For example, BMTree outperforms the Z-curve by 18.2%–29.0% (resp. 13.7%–28.4%, 13.5%–26.5%, and 2.8%–25.3%) in terms of node access number on UNI (resp. GAU, US-OSM, and TIGER) dataset. We also observe that the Hilbert curve achieves similar performance with BMTree on GAU dataset, which could be attributed to its good toleration to the skewness [43].

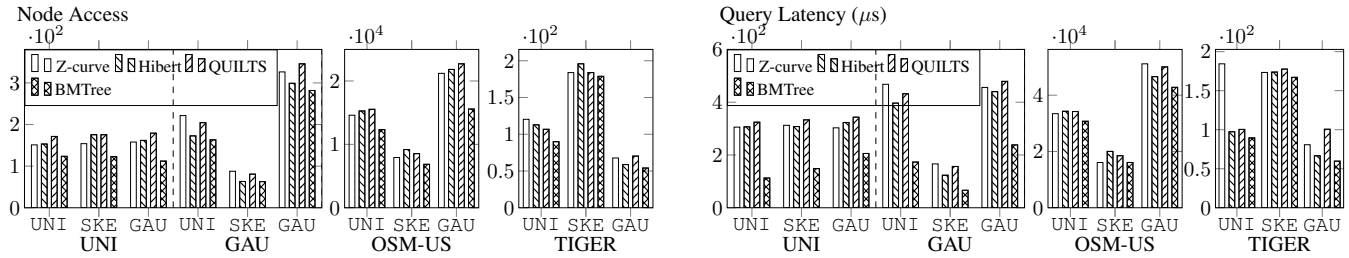
Comparing with other indexes. We compare the performance of two SFC-based indexes, RSMI and ZM, with baseline indexes including (1) two R tree variants: STR [19] and R* Tree [5]; and (2)

#1 R1W2D7
#2 R2O3
#3 E2



(a) I/O Cost (b) Query Latency

Figure 6: Results under PostgreSQL, where the first and the second lines under the x bar denote the query and data distribution.



(a) Node Access (b) Query Latency

Figure 7: Results using RSMI learned index structure.

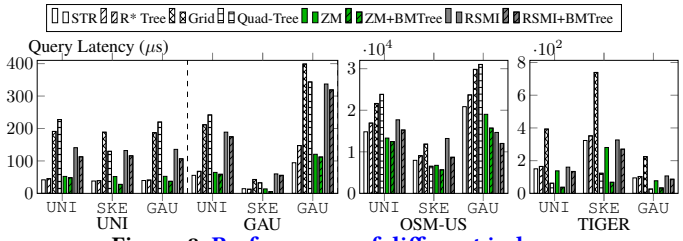
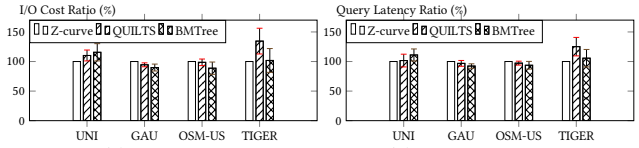


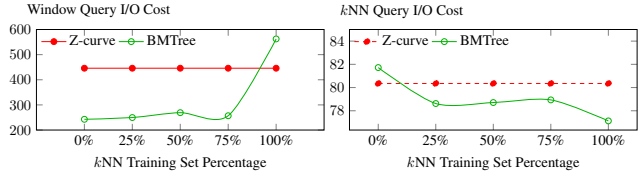
Figure 8: Performance of different indexes.

two partition-based methods: Grid-File and Quad-Tree. For RSMI and ZM, we consider both its original version and its combination with BMTree. We adopt the same setting (in memory, and with the same node size) for all indexes for a fair comparison. Figure 8 reports the results of QL. We observe that the original ZM (called "ZM") performs comparably with STR and R* Tree and outperforms Grid-File and Quad-Tree for most of the data-query scenarios. The ZM with BMTree (called "ZM+BMTree") outperforms the R tree variants and partition-based methods in most cases, especially under the SKE workload.



(a) I/O Cost (b) Query Latency

Figure 9: Performance of kNN queries.



(a) Window Query I/O (b) kNN Query I/O

Figure 10: Window query kNN query joint optimization.

Effect on kNN queries. The piecewise SFC is learned to optimize window queries. To see whether it has negative influence on the performance of the kNN queries. We generate 1,000 kNN query points following the data distribution, and we apply the kNN algorithm [34] in PostgreSQL, with k set at 25. We report the IO and QL ratios in Figure 9a and 9b, which are the ratio of results of different curves divided by the result of the Z-curve. We observe BMTree is comparable with the baselines: BMTree performs slightly better than the baselines on GAU and OSM-US while the Z-curve is slightly better on UNI and TIGER. Therefore, although the piecewise SFC is optimized for window queries, the performance of the kNN query is not compromised.

Optimizing window query and kNN query. We evaluate the performance when window query and kNN query are optimized together. To optimize our BMTree method for kNN queries, we convert kNN queries into window queries by following [34] and include them in the training workload. We then vary the weight of the objective based on kNN queries relative to that based on the window queries from 0% to 100% during training. Figure 10a reports the window query IO and Figure 10b reports the kNN query IO. We observe that as the weight increases, the window query IO tends to increase while the kNN query IO tends to decrease. We also observe that when weight is between 25% and 75%, the performance of the window query only mildly degrades, while the performance of kNN query is better than that based on Z-curve. The results show the potential of our method to optimzie two types of queries together.

#2 R1O1 #3 W2

Table 2: Analysis on the generated BMTree.

Dataset	UNI			GAU			OSM-US			TIGER		
Workload	UNI	SKE	GAU	UNI	SKE	GAU	UNI	SKE	GAU	UNI	SKE	GAU
#P	47	26	39	26	28	51	48	29	32	36	19	28
top-1 %$\mathcal{D}$	6.2	17.2	6.2	17.2	12.1	8.1	9.2	12.1	18.7	39.6	40.1	44.0
top-10 %$\mathcal{D}$	37.5	73.4	44.9	73.4	77.9	54.7	52.8	72.3	73.9	99.6	99.9	99.9
Z-curve (%)	0	0	0	0	0	0	0	0	0	0	0	0
C-curve (%)	0	0	2.0	0	0	0	3.2	9.1	15.2	0	0	0

4.2.2 Understanding piecewise SFCs. To understand the piecewise SFCs and their effectiveness, we perform the following analysis.

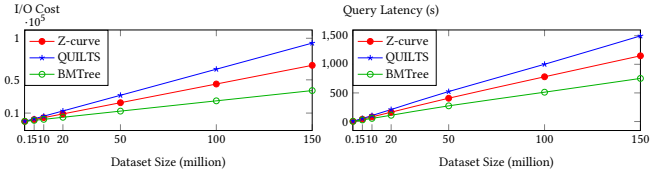
Analysis of generated BMTree. Each generated BMTree has at most 1024 paths decided by the trained RL agent when we set the Max Depth at 10 during training. Therefore, each BMTtree encodes at most 1024 distinct BMPs. However, multiple paths may share the same BMP. Table 2 shows the number of distinct BMPs (#P). It also shows the percentage of the data that the top-k BMPs cover (top-k %$\mathcal{D}$, we consider top-1 and top10). We also report the percentage of two typical patterns, Z-curve and C-curve, among the BMPs in the BMTree. We observe: (1) For each dataset, the number of patterns (#P) in BMTrees varies significantly across the different workloads. Usually no single pattern can dominate the data space. (2) The popular space-filling curves Z-curve(%) and C-curve(%) take a small portion among the returned BMPs in BMTrees. (3) On the SKE workload, our algorithm tends to generate more distinct BMPs for different subspaces on UNI, GAU, and US-OSM datasets, while no single BMP dominates.

Effectiveness of the piecewise design. To evaluate the effectiveness of our piecewise design, we vary the split depth from 0 to 10. When it is 0, MCTS returns a single BMP, but not a piecewise SFC. Table 3 gives the result under default settings. We observe that a single BMP performs much worse than piecewise SFCs, and ScanRange further drops when the splitting granularity of subspaces increases, i.e., more piecewise BMPs are generated.

Table 3: The effect of split depths.

Split depth	0	2	5	8	10
ScanRange	939.6	562.3	524.6	520.7	519.1
Training Time (min)	16.3	21.3	35.7	105.4	133.9

4.2.3 Varying settings. We evaluate BMTree's performance under various settings: dataset/query size, distribution shifting, dimensionality, and window aspect ratio.



(a) I/O Cost **(b) Query Latency**

Figure 11: Performance vs dataset size.

Scalability of learned SFCs. To evaluate the scalability of BMTree, we evaluate the performance of the SFCs by varying data size from 0.1 to 150 million. We construct the BMTree based on the 1 million data, and the others follow the default settings. The result is shown in Figure 11. We observe that the BMTree displays a linear trend for the IO and QL when data size increases. We observe similar trends for baselines.

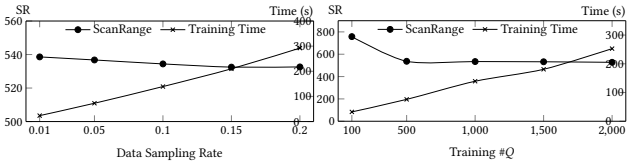
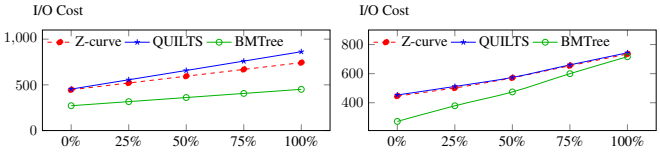


Figure 12: Varying data sampling size.

Figure 13: Varying training #Q.

Effect of training data and query size. We evaluate the effect of the sizes of training data and query on SR and training time. (1) Varying data sampling rate. To vary the training data size, we vary the data sampling rate from 0.01 to 0.2. Figure 12 shows that SR decreases as the data sampling rate increases. This is because a higher sampling rate would result in a more reliable reward, and thus our model is expected to perform better. In addition, with the increase of the sampling rate, the training time per episode grows as expected. There is a trade-off between accuracy and training efficiency. (2) Varying training query size (#Q). We vary #Q from 100 to 2,000. Figure 13 shows that with the increase of #Q, BMTree tends to achieve better SR. However, SR is relatively stable after #Q is greater than 1,000. Training time increases linearly with #Q, which is consistent with the time complexity analysis.

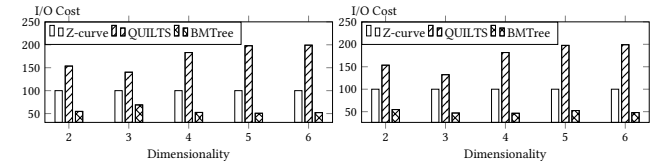


(a) Data distribution shift **(b) Query distribution shift**

Figure 14: Performance under data & query distribution shift.

Effect of distribution shifting. We evaluate the effect of data distribution and query distribution shifts on the performance of the SFCs. For the data distribution shift, we insert different percentage of data points following the GAU distribution to the dataset following the UNI distribution, and report the IO in Figure 14a. For the query distribution shift, we train the model by using queries following the SKE distribution and then test the model with different percentages of queries following the GAU distribution, and report the IO in Figure 14b. We observe that with shift of data distribution, BMTree still perform much better than Z-curve and QUILTS. Although its performance deteriorates under the shift of query distribution, BMTree still outperforms Z-curve and QUILTS, especially when the shift is mild. #1 R3W1

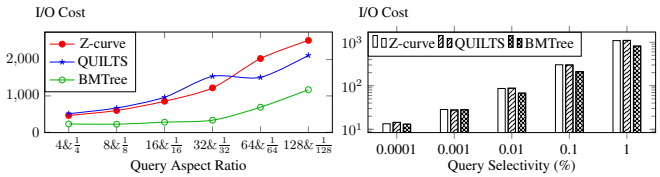
Effect of higher dimensionality. To evaluate the effect of dimensionality on the effectiveness of the learned SFC, we vary the dimensionality from 2 to 6 on the dataset following both uniform and normal distributions. We report the IO in Figure 15. BMTree consistently outperforms the baselines and saves up to 54% of I/O cost compared with the best baseline Z-curve. This demonstrates that our method generalizes well on data with more than 2 dimensions.



(a) Uniform Data **(b) Normal Data**

Figure 15: I/O cost vs dimensionality.

Effect of varying query aspect ratio and selectivity. (1) We evaluate the performance of BMTree by varying query aspect ratios from $\{4, \frac{1}{4}\}$ to $\{128, \frac{1}{128}\}$, and the results are reported in Figure 16a. We observe that BMTree works consistently better than other SFCs across different aspect ratios including very wide ones. (2) We #1 R3D6



(a) Varying aspect ratio **(b) Varying selectivity**

Figure 16: Varying query aspect ratio and selectivity.

vary query selectivity from 0.0001% to 1% and report the results in Figure 16b. We observe that the improvement of BMTree is subtle under very small query range. The reason could be because BMTree fed with queries of very small range overfits easily, and requires more queries to be effective.

Effect of the max depths. We evaluate the effect of the max depth parameter which is the depth of BMTree that are built using RL. We vary the max depth from 1 to 20. The results are reported in Table. 4. The result shows that as the max depth increases, *SR* drops and then tends to be stable after 10.

Table 4: The effect of the max depths.

Max depth	1	5	10	15	20
ScanRange	967.6	671.1	532.9	527.8	527.6
Training Time (min)	2.0	8.8	100.1	254.8	409.5

#2 R4O2 R3O5

4.2.4 Evaluating BMTree variants. We analyze four BMTree variants: BMTree-Data Driven (with dataset only), BMTree-noGAS (no GAS algorithm included), BMTree-greedy (pure greedy), and BMTree-LMT (with limited BMPs). Results are in Figure 17.

(1) BMTree-DD. We evaluate the performance of BMTree when query workload is not available. We generate training queries for BMTree by following the dataset's distribution. In Figure 17, we observe that BMTree-DD performs comparably with BMTree on the UNI workloads for all datasets. However, on the SKE workload, BMTree performs much better in general. **(2) BMTree-noGAS.** We evaluate the effectiveness of GAS algorithm. We observe a performance drop compared with the MCTS using GAS and this demonstrates the usefulness of GAS. **(3) BMTree-greedy.** We apply GAS for all action selections and build a purely greedy based BMTree. We observe that MCTS with GAS outperforms both BMTree-noGAS and BMTree-greedy. This indicates a synergistic improvement of MCTS over GAS. **(4) BMTree-LMT.** To evaluate the superiority of considering all BMPs, we designed a baseline BMTree in which only Z-curve and C-curve are allowed to be assigned to the subspaces, denoted as BMTree-LMT. We observe a significant improvement of BMTree over using Z-curve and C-curve alone, which demonstrates the necessity of considering all BMPs.

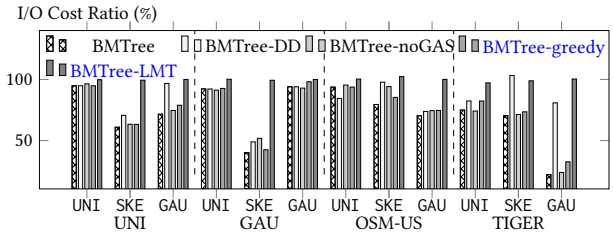


Figure 17: I/O Cost on BMTree Variants.

4.2.5 Effectiveness of SR. We propose *SR* to replace the actual IO for training, aiming to accelerate the training. We report that computing *SR* is 20× faster than executing queries under PostgreSQL and 100× faster than RSMI since RSMI needs to train before executing queries, demonstrating the efficiency advantage of *SR*. We then evaluate if the *SR* metric is consistent with query performance of different indexes. Table 5 shows the results of *SR* with the SKE query workload and UNI (or GAU) data. We report the corresponding I/O cost (IO_B) under PostgreSQL as well as node access number (NA_R) under RSMI. We observe that the *SR* value is consistent with IO_B and NA_R, showing that optimizing *SR* is a good replacement for IO_B and NA_R in optimizing BMTree.

Table 5: SR vs IO_B (resp. NA_R)

Query + Data	SKE + UNI			SKE + GAU		
Metric	IO_B	NA_R	*SR*	IO_B	NA_R	*SR*
Z-Curve	453.9	154.0	1020	301.0	87.7	657
Hilbert curve	/	175.5	944	/	63.1	656
QUILTS	633.1	175.5	857	342.4	80.8	690
BMTree	**271.5**	**123.1**	**569**	**120.8**	**62.8**	**182**

5 RELATED WORK

Space-Filling Curves. Many SFCs have been developed. C-curve [12] organizes the data points dimension-by-dimension. Z-curve [28–30] and Hilbert curves [12, 13, 24, 25] are widely applied in index design. Despite the success of these SFCs in many applications, they do not consider data distribution and query workload. QUILTS [27] is proposed to consider data distribution and query workload in designing the mapping function of SFCs. All these SFCs, including QUILTS, adopt a single mapping scheme, which may not always be suitable for the whole data space and query workload as illustrated in Section 1. In this paper, we propose the first piecewise SFC, in which we design different mapping functions for different data subspaces by considering both data distribution and query workload. Furthermore, we propose a method based on reinforcement learning to learn SFCs to directly optimize the performance.

Space-Filling Curves based Index Structures. SFCs can be used for indexing multi-dimensional data points with the mapped values, and this is widely adopted by DBMS. SFCs are also essential for recent works on learned multi-dimensional indexes [34, 42]. Specifically, ZM [42] combines a Z-curve with a learned index, namely RMI [15]. RSMI [34] applies the Hilbert curve together with a learned index structure for spatial data. Pai et al. [31] present preliminary results on the instance-optimal Z-index based on the Z-curve that adapts to the data and workload. SFC-based indexes can also be applied for data skipping [35, 40, 44], which aims to partition and organize data into data pages such that querying algorithms only access pages that are relevant to a query. SFC based approach [26] maps a multidimensional data point to a scalar value based on a SFC, and then uses the B+-tree or range-partitioned key value store (e.g., H-base) for partitioning and organizing data.

Analysis on Space-Filling Curves. There are studies [22–25, 27, 43] that evaluate SFCs. Mokbel et al. [22–24] discuss the characteristics of good SFCs. Moon et al. [25] propose clustering number, which represents the number of disk seeks during query processing. Xu et al. [43] prove that the Hilbert curve is a preferable SFC with

a low clustering number. Nishimura et al. [27] propose cohesion cost, which evaluates how good SFCs could cluster data.

Reinforcement Learning for building trees. Our method of generating SFC is based on reinforcement learning techniques [6, 41]. There are several recent studies [11, 20, 44] on applying the RL techniques to generate tree structures. Yang et al. [44] construct the Qd-tree for partitioning data into blocks on storage with Proximal Policy Optimization (PPO) networks [37]. Gu et al. [11] propose to utilize RL to construct the R-tree for answering spatial queries, and Neurocuts [20] constructs a decision tree based on the RL agent. These RL designs are not suitable for our task of learning piecewise SFCs and our design of RL models is based on MCTS and is different from the designs in these studies.

6 CONCLUSION

In this paper, we study the Space-Filling Curve Design problem and propose constructing piecewise SFCs that adopt different mapping schemes for different data subspaces. Specifically, we propose the BMTree for maintaining multiple bit merging patterns, in which every path corresponds to a BMP. We further propose to construct the BMTree in a data-driven manner via reinforcement learning. We conduct extensive experiments on both synthetic and real datasets with different query workloads. The results verify that our piecewise SFCs are consistently superior over existing SFCs, especially when data or (and) queries have a certain degree of skewness.

REFERENCES

[1] n.d.. OpenStreetMap. http://www.openstreetmap.org/.

[2] n.d.. PostGis. https://postgis.net/docs/using_postgis_dbmanagement.html.

[3] n.d.. TIGER/Line Shapefiles. http://www.census.gov/geo/www/tiger/.

[4] Rudolf Bayer. 1997. The universal B-tree for multidimensional indexing: General concepts. In *International Conference on Worldwide Computing and Its Applications*. Springer, 198–209.

[5] 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 *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*. 322–331.

[6] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. 2012. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games* 4, 1 (2012), 1–43.

[7] Jialin Ding, Vikram Nathan, Mohammad Alizadeh, and Tim Kraska. 2020. Tsunami: a learned multi-dimensional index for correlated data and skewed workloads. *Proceedings of the VLDB Endowment* 14, 2 (2020), 74–86.

[8] Ahmed Eldawy and Mohamed F Mokbel. 2015. Spatialhadoop: A mapreduce framework for spatial data. In *2015 IEEE 31st international conference on Data Engineering*. IEEE, 1352–1363.

[9] Christos Faloutsos. 1988. Gray codes for partial match and range queries. *IEEE Transactions on Software Engineering* 14, 10 (1988), 1381–1393.

[10] Christos Faloutsos and Shari Roseman. 1989. Fractals for secondary key retrieval. In *Proceedings of the eighth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. 247–252.

[11] Tu Gu, Kaiyu Feng, Gao Cong, Cheng Long, Zheng Wang, and Sheng Wang. 2021. The RLR-tree: a reinforcement learning based R-tree for spatial data. *arXiv e-prints* (2021), arXiv–2103.

[12] Hosagrahar V Jagadish. 1990. Linear clustering of objects with multiple attributes. In *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*. 332–342.

[13] Andreas Kipf, Harald Lang, VN Pandey, Raul Alexandru Persa, Christoph Anneser, Eleni Tzirita Zacharatou, Harish Doraiswamy, Peter A Boncz, Thomas Neumann, and Alfons Kemper. 2020. Adaptive main-memory indexing for high-performance point-polygon joins. (2020).

[14] Levente Kocsis and Csaba Szepesvári. 2006. Bandit based monte-carlo planning. In *European conference on machine learning*. Springer, 282–293.

[15] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*. 489–504.

[16] Jonathan K. Lawder and Peter J. H. King. 2001. Querying multi-dimensional data indexed using the Hilbert space-filling curve. *ACM Sigmod Record* 30, 1 (2001), 19–24.

[17] Ken CK Lee, Wang-Chien Lee, Baihua Zheng, Huajing Li, and Yuan Tian. 2010. Z-SKY: an efficient skyline query processing framework based on Z-order. *The VLDB Journal* 19, 3 (2010), 333–362.

[18] Ken CK Lee, Baihua Zheng, Huajing Li, and Wang-Chien Lee. 2007. Approaching the skyline in Z order.. In *VLDB*, Vol. 7. 279–290.

[19] Scott T. Leutenegger, J. M. Edgington, and Mario Alberto López. 1997. STR: A Simple and Efficient Algorithm for R-Tree Packing. In *Proceedings of the Thirteenth International Conference on Data Engineering, April 7-11, 1997, Birmingham, UK*, W. A. Gray and Per-Åke Larson (Eds.). IEEE Computer Society, 497–506. https://doi.org/10.1109/ICDE.1997.582015

[20] Eric Liang, Hang Zhu, Xin Jin, and Ion Stoica. 2019. Neural packet classification. In *Proceedings of the ACM Special Interest Group on Data Communication*. 256–269.

[21] Guanli Liu. 2020. RSMI Code Released. https://github.com/Liuguanli/RSMI.

[22] Mohamed F Mokbel and Walid G Aref. 2001. Irregularity in multi-dimensional space-filling curves with applications in multimedia databases. In *Proceedings of the tenth international conference on Information and knowledge management*. 512–519.

[23] Mohamed F Mokbel and Walid G Aref. 2003. On query processing and optimality using spectral locality-preserving mappings. In *International Symposium on Spatial and Temporal Databases*. Springer, 102–121.

[24] Mohamed F Mokbel, Walid G Aref, and Ibrahim Kamel. 2003. Analysis of multi-dimensional space-filling curves. *GeoInformatica* 7, 3 (2003), 179–209.

[25] Bongki Moon, Hosagrahar V Jagadish, Christos Faloutsos, and Joel H. Saltz. 2001. Analysis of the clustering properties of the hilbert space-filling curve. *IEEE Transactions on knowledge and data engineering* 13, 1 (2001), 124–141.

[26] Shoji Nishimura, Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. 2011. MD-HBase: A scalable multi-dimensional data infrastructure for location aware services. In *2011 IEEE 12th International Conference on Mobile Data Management*, Vol. 1. IEEE, 7–16.

[27] Shoji Nishimura and Haruo Yokota. 2017. Quilts: Multidimensional data partitioning framework based on query-aware and skew-tolerant space-filling curves. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1525–1537.

[28] Jack A Orenstein. 1986. Spatial query processing in an object-oriented database system. In *Proceedings of the 1986 ACM SIGMOD international conference on Management of data*. 326–336.

[29] Jack A Orenstein. 1989. Redundancy in spatial databases. *ACM SIGMOD Record* 18, 2 (1989), 295–305.

[30] JA Orenstem and TH Merett. 1984. A Class of Data Structures for Assoclattve Searchmg. *Proc Thrrd SIGACT News SIGMOD Symposwn on the Prmclples of Database Systems* (1984), 181–190.

[31] Sachith Gopalakrishna Pai, Michael Mathioudakis, and Yanhao Wang. 2022. Towards an Instance-Optimal Z-Index. In *4th International Workshop on Applied AI for Database Systems and Applications (AIDB@ VLDB2022)*.

[32] Giuseppe Peano. 1890. Sur une courbe, qui remplit toute une aire plane. *Math. Ann.* 36, 1 (1890), 157–160.

[33] Martin L Puterman. 2014. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons.

[34] Jianzhong Qi, Guanli Liu, Christian S Jensen, and Lars Kulik. 2020. Effectively learning spatial indices. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2341–2354.

[35] Vijayshankar Raman, Gopi Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M Lohman, et al. 2013. DB2 with BLU acceleration: So much more than just a column store. *Proceedings of the VLDB Endowment* 6, 11 (2013), 1080–1091.

[36] Hanan Samet. 2006. *Foundations of multidimensional and metric data structures*. Morgan Kaufmann.

[37] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347* (2017).

[38] Tomáš Skopal, Michal Krátkỳ, Jaroslav Pokornỳ, and Václav Snášel. 2006. A new range query algorithm for Universal B-trees. *Information Systems* 31, 6 (2006), 489–511.

[39] Zack Slayton. 2017. Z-Order Indexing for Multifaceted Queries in Amazon DynamoDB. https://aws.amazon.com/blogs/database/z-order-indexing-for-multifaceted-queries-in-amazon-dynamodb-part-1/.

[40] Liwen Sun, Michael J Franklin, Sanjay Krishnan, and Reynold S Xin. 2014. Fine-grained partitioning for aggressive data skipping. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 1115–1126.

[41] Richard S Sutton and Andrew G Barto. 2018. *Reinforcement learning: An introduction*. MIT press.

[42] Haixin Wang, Xiaoyi Fu, Jianliang Xu, and Hua Lu. 2019. Learned index for spatial queries. In *2019 20th IEEE International Conference on Mobile Data Management (MDM)*. IEEE, 569–574.

[43] Pan Xu and Srikanta Tirthapura. 2014. Optimality of clustering properties of space-filling curves. *ACM Transactions on Database Systems (TODS)* 39, 2 (2014), 1–27.

[44] 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 *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 193–208.

[45] Man Lung Yiu, Yufei Tao, and Nikos Mamoulis. 2008. The B dual-Tree: indexing moving objects by space filling curves in the dual space. *The VLDB Journal* 17, 3 (2008), 379–400.

[46] Liang Zhou, Chris R Johnson, and Daniel Weiskopf. 2020. Data-driven space-filling curves. *IEEE Transactions on Visualization and Computer Graphics* 27, 2 (2020), 1591–1600.

[47] Xuanhe Zhou, Guoliang Li, Chengliang Chai, and Jianhua Feng. 2021. A learned query rewrite system using monte carlo tree search. *Proceedings of the VLDB Endowment* 15, 1 (2021), 46–58.