Efficient Cost Modeling of Space-filling Curves [Technical Report]

Guanli Liu
The University of Melbourne
guanli@student.unimelb.edu.au

Lars Kulik
The University of Melbourne
lkulik@unimelb.edu.au

Christian S. Jensen
Aalborg University
csj@cs.aau.dk

Tianyi Li
Aalborg University
tianyi@cs.aau.dk

Renata Borovica-Gajic
The University of Melbourne
renata.borovica@unimelb.edu.au

Jianzhong Qi
The University of Melbourne
jianzhong.qi@unimelb.edu.au

ABSTRACT

A *space-filling curve* (SFC) maps points in a multi-dimensional space to one-dimensional points by discretizing the multi-dimensional space into cells and imposing a linear order on the cells. This way, an SFC enables computing a one-dimensional layout for multi-dimensional data storage and retrieval. Choosing an appropriate SFC is crucial, as different SFCs have different effects on query performance. Currently, there are two primary strategies: 1) deterministic schemes, which are computationally efficient but often yield suboptimal query performance, and 2) dynamic schemes, which consider a broad range of candidate SFCs based on cost functions but incur significant computational overhead. Despite these strategies, existing methods cannot efficiently measure the effectiveness of SFCs under heavy query workloads and numerous SFC options.

To address this problem, we propose means of *constant-time* cost estimations that can enhance existing SFC selection algorithms, enabling them to learn more effective SFCs. Additionally, we propose an SFC learning method that leverages reinforcement learning and our cost estimations to choose an SFC pattern efficiently. Experimental studies offer evidence of the effectiveness and efficiency of the proposed means of cost estimation and SFC learning.

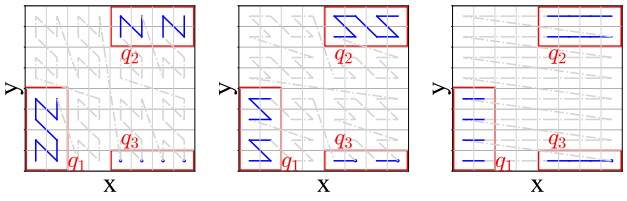
PVLDB Reference Format:
Guanli Liu, Lars Kulik, Christian S. Jensen, Tianyi Li, Renata Borovica-Gajic, and Jianzhong Qi. Efficient Cost Modeling of Space-filling Curves [Technical Report]. PVLDB, 17(1): XXX-XXX, 2024.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://anonymous.4open.science/r/LearnSFC-B6D8.

1 INTRODUCTION

Data layout has a significant impact on the efficiency of querying increasingly massive multidimensional data. In this setting, *space-filling curves* (SFC) are used widely for data ordering and layout computations. For example, *Z-order curves* (ZC, see Figures 1a and 1b) [23] are used in Hudi [2], RedShift [1], and SparkSQL [5]; *C-Curves, which order data points lexicographically by their dimension values* (CC, see Figure 1c), are used in PostgreSQL [25] and SQL Server [16]; and *Hilbert curves* (HC) [6] are used in Google S2 [28].

The range query emerges as an important type of query. The most efficient query processing occurs when the data needed for a query result is stored consecutively, or when the data is stored in a few data blocks. Thus, the storage organization—the order in which the data is stored—affects the cost of processing a query profoundly. When storing data with SFC-based ordering, the choice of which SFC to use for ordering the data is important.



(a) Curve 1 (ZC) (b) Curve 2 (ZC) (c) Curve 3 (CC)

Figure 1: Examples of SFCs (in grey) and queries (in red).

Different range queries benefit differently from different SFCs. In Figure 1, three SFCs on the same data space are shown along with three queries. The fewer disconnected segments of an SFC that need to be accessed to compute a query, the better. To compute q_1, the SFC in Figure 1a is preferable because only a single segment needs to be accessed. Put differently, the data needed may be in a single or in consecutive blocks. In contrast, the SFCs in Figures 1b and 1c map the needed data to two and four segments, respectively.

Next, we observe that no single SFC is optimal for all queries. While the SFC in Figure 1a is good for q_1, it is suboptimal for q_2 and q_3. It is thus critical to select the right SFC. This in turn calls for efficient means of estimating the cost of processing a query using a particular SFC (without query execution) to guide SFC selection.

Existing studies [18, 36] provide *cost estimations* based on counting the number of clusters (continuous curve segments) covered by a query. However, their calculations rely on curve segment scans that require $O(V)$ time, where V is proportional to the size of a query. Given a workload of n queries and m candidate SFCs, $O(n \cdot m \cdot V)$ time is needed to choose an SFC. This is expensive given large n and m (e.g., a $k \times k$ grid can form $m = k^2$! candidate SFCs), thus jeopardizing the applicability of the cost model.

In this paper, we provide efficient means of SFC cost estimation such that a *range query-optimal* SFC can be found efficiently. Specifically, we present algorithms that compute the cost of a query in

Proceedings of the VLDB Endowment, Vol. 17, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

$O(1)$ time. After an $O(n)$-time initialization, the algorithms compute the cost of n queries in $O(1)$ time for each new SFC to be considered. This means that given m candidate SFCs, our algorithms can find the optimal SFC in $O(m)$ time, which is much smaller than $O(n \cdot m \cdot V)$ and thus renders SFC cost estimation practical.

Our algorithms are based on a well-chosen family of SFCs, the *bit-merging curves* (BMC) [8, 20]. A BMC maps d-dimensional ($d \geq 2$) points by merging the bit sequences of the point coordinates (i.e., column indices) from all d dimensions. We consider BMCs for two reasons: (1) BMCs generalize ZC and CC used in real systems [2, 5, 16, 25]. Algorithms to find optimal BMCs can be integrated seamlessly into real systems. (2) The space of BMCs is large. For example, in a 2-dimensional space ($d = 2$), where each dimension uses 16 bits ($\ell = 16$) for a column index, there are $k = 2^\ell$ columns in each dimension of the grid. This yields about 6×10^8 (i.e., $\frac{(d \cdot \ell)!}{(\ell!)^d}$) candidate BMCs. An efficient cost model enables finding a query-efficient SFC in this large space.

Our algorithms model the cost of a range query based on the number and lengths of curve segments covered by the query, which in turn relate to the difference between the curve values of the end points of each curve segment. We exploit the property that the curve values of a BMC come from merging the bits of the column indices. This property enables deriving a closed-form equation to compute the length of a curve segment in $O(d \cdot \ell) = O(1)$ time (given that d and ℓ are constants) for n queries. The property also enables pre-computing d look-up tables that allow computing the number of curve segments in $O(d \cdot \ell) = O(1)$ time. Thus, we achieve constant-time SFC cost estimation. We show the applicability of our cost estimation algorithms by integrating them into the state-of-the-art learned BMC-based structure, the *BMTree* [14], reducing its curve learning time by up to two orders of magnitude.

Furthermore, we develop an SFC learning algorithm named *LBMC* that uses Reinforcement Learning (RL) to find the optimal BMC. Importantly, the reward calculation in RL leverages our efficient cost estimations, thus making the entire learning process extremely fast. This enables the RL agent to converge rapidly to near-optimal solutions while navigating the state space.

In summary, the paper makes the following contributions:

(1) We propose algorithms for efficient range query cost estimation when using BMC-based data layout for multi-dimensional datasets. The algorithms can compute the cost of a range query in $O(1)$ time as well as the cost of a workload of n queries in $O(1)$ time, after a simple scan over the queries. (2) We generalize the applicability of the cost estimation to existing state-of-the-art SFC learning methods based on BMCs, enhancing the learning efficiency of such methods. (3) We propose LBMC, an efficient BMC learning algorithm that leverages the proposed cost estimation. (4) We evaluate the cost estimation and LBMC algorithms on both real and synthetic datasets, finding that (i) our cost estimation outperforms baselines consistently by up to 10^5 times in efficiency, (ii) our cost estimation accelerates the reward calculation of the BMTree by 400x with little impact on query efficiency, and (iii) the LBMC algorithm is applied in a real data lake platform and improves query efficiency by up to 61% compared to other ordering techniques.

2 RELATED WORK

Space-filling curves. SFCs find use in many fields, including in indexing [11, 13, 22, 35], data mining [3], and machine learning [9, 33]. Two popular SFCs, *ZC* [23] and *HC* [6], have been deployed in practical data systems [1, 2, 5]. Bit-merging curves (BMCs) are a family of SFCs, where the curve value of a grid cell is formed by merging the bits of the cell's column indices from all dimensions. To order data points for specific query workloads, *QUILTS* [20] provides a heuristic method to design a series of BMCs and selects the optimal one. The *Bit Merging Tree* (BMTree) [14] learns piecewise BMCs by using a quadtree [7]-like strategy to partition the data space and selecting different BMCs for different space partitions.

SFC cost estimation. To learn an optimal SFC, cost estimation is employed to approximate the query costs without computing the queries. Studies [18, 36] offer theoretical means of estimating the number of curve segments in a query. They do not offer empirical results or guidance on how to construct a query-efficient SFC.

QUILTS formulates the query cost C_t for a BMC-based data layout over a set of queries as $C_t = C_g \cdot C_l$, where C_g is a *global cost* and C_l is a *local cost*. The global cost is the length of a continuous BMC segment that can cover a query range q fully minus the length of the BMC segments in q. The idea is to count the number of segments outside q that may need to be visited to compute the queries. The local cost is the entropy of the relative length of each segment of the BMC curve outside q counted in the global cost, which reflects how uniformly distributed the lengths of such segments are. These two costs rely on the length of the curve segments outside q, which is expensive to compute. Given n queries, it takes $O(n \cdot c_t)$ time to compute C_t, where $O(c_t)$ is the average estimation cost per query.

The BMTree estimates query costs using data points sampled from the target dataset. Such cost estimations are expensive for large datasets and many queries. LMSFC [8], another recent proposal, learns a parameterized SFC (effectively a BMC) using Bayesian optimization [10]. Like the BMTree, LMSFC uses a sampled dataset and a query workload for cost estimation and thus shares the same issues with the BMTree. Our study aims to address these issues by providing a highly effective and efficient cost estimation.

SFC-based data layout. SFCs, especially ZCs and HCs, are used for data layout in data systems to preserve data locality. For example, in Apache Hudi, for each data record, its values at user chosen columns for data layout are each converted into an 8-byte integer (with truncation if necessary), forming a multi-dimensional point of integer coordinates. This point is mapped to a one-dimensional value using a ZC or HC, and all mapped values are used to order the data records to produce a data layout. We aim to learn SFCs to optimize the data layout for more efficient range query processing.

There are other ordering schemes. For example, spectral ordering [4, 19, 30] uses eigenvalues and eigenvectors to order graph vertices based on their connectivity. It aims to reveal graph partitions that minimize the cut size. While it could be adapted to order points in multi-dimensional space, it does not lead to efficient range query processing and hence will not be considered further.

SFC-based indices. The Hilbert R-tree [11, 27] and Z-Rtree [22] use HCs and ZCs to order multi-dimensional data for bulk-loading R-trees. Another index, the *Instance-Optimal Z-Index* [24], uses a

quadtree-like strategy to partition the data space recursively and each sub-space follows a 'Z' or an 'N' shape.

In the recent wave of learned multi-dimensional indices [26, 34], SFCs have been used to map multidimensional points to one-dimensional values, such that one-dimensional learned indices [12] apply. Our cost estimations can be applied to learn SFCs for the mapping step of these indices. However, our work is orthogonal to these studies and we do not aim to propose another learned index.

3 PRELIMINARIES

We start with core concepts underlying BMCs and list frequently used symbols in Table 1.

Table 1: Frequently used symbols.

Symbol	Description
d	The data space dimensionality
ℓ	The number of bits for grid cell numbering in each dimension
D	A multi-dimensional dataset
p	A data point
q	A range query
Q	A set of range queries
B	The block size
p_s, p_e	The start and end points on an SFC of a range query
n	The number of range queries
σ	A bit-merging curve (BMC)
$\mathcal{F}_\sigma$	The curve value calculation function over BMC σ
α_i^j	The jth bit value in dimension i
γ_i^j	The position (0-indexed) of α_i^j in a BMC σ
x_i	A value in dimension i
$[x_{s,i}, x_{e,i}]$	A value range in dimension i

3.1 BMC Definition

A BMC maps multi-dimensional points by merging the *bit sequences* of the coordinates (i.e., column indices) from all d dimensions into a single bit sequence that becomes a one-dimension value [20].

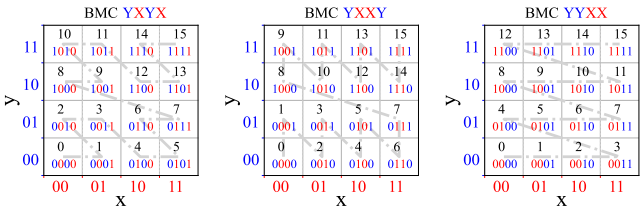


Figure 2: BMC examples ($d = 2$ and $\ell = 2$).

Figure 2 plots three BMC schemes, which are represented by YXYX, YXXY, and YYXX. Here, the ordering of the X's and Y's specify how the bits from dimensions x and y are combined to obtain a BMC σ. The coordinates from each dimension have two bits, i.e., the *bit length* ℓ of each dimension is 2. The merged bit sequence (i.e., the curve value in binary form) has $d \cdot \ell = 4$ bits.

The bit length ℓ is determined by the grid resolution, which is a system parameter. For simplicity, we use the same ℓ for each dimension (our techniques allow different ℓ's in different dimensions), and we call the column indices of a point p in a cell (or the cell itself) the *coordinates* of p (or the cell).

BMC value calculation. Given a BMC σ, we compute the curve value of a point $p = (x_1, x_2, \ldots, x_d)$ using function $\mathcal{F}_\sigma(p)$:

$$\mathcal{F}_\sigma(p) = \sum_{i=1}^{d} \sum_{j=1}^{\ell} \alpha_i^j \cdot 2^{\gamma_i^j} \tag{1}$$

Rank of a bit in BMC. Let x_i be the dimension-i coordinate of p. In Equation 1, $\alpha_i^j \in \{0, 1\}$ is the jth ($j \in [1, \ell]$) bit of x_i, and γ_i^j is the *rank* of α_i^j in the BMC.

$$\sum_{j=1}^{\ell} \alpha_i^j \cdot 2^{j-1} = x_i \tag{2}$$

Note that the order among the bits from the same dimension does not change when the bits are merged with those from the other dimensions to calculate $\mathcal{F}_\sigma(p)$, i.e., for bits α_i^j and α_i^{j+1}, $\gamma_i^j < \gamma_i^{j+1}$.

For ease of discussion, we use examples with up to three dimensions x, y, and z. Figure 3 calculates $\mathcal{F}_\sigma(p)$ for $p = (2, 1, 7)$ given σ = XYZXYZXYZ. Here, $\alpha_3^1 = 1$ is the first bit value in dimension z, and the rank of the first (i.e., rightmost) Z bit in σ is zero, which means $\gamma_3^1 = 0$. To calculate the curve value of a point for a given σ, we derive each α_i^j and γ_i^j based on x_i and σ, respectively.

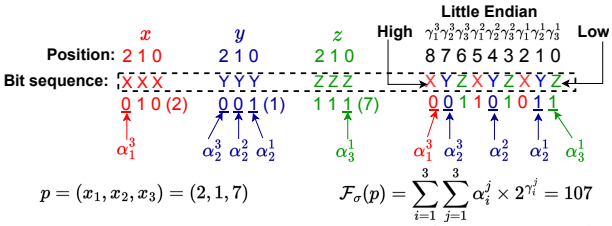


Figure 3: BMC curve value calculation ($d = 3$ and $\ell = 3$).

BMC monotonicity. The BMC value calculation process implies that any BMC is monotonic [14].

Theorem 1 (Monotonicity [14]). *Given $p_1 = (x_{1,1}, \ldots, x_{1,d})$ and $p_2 = (x_{2,1}, \ldots, x_{2,d})$ then $\forall i \in [1, d](x_{1,i} \leq x_{2,i}) \rightarrow \mathcal{F}_\sigma(p_1) \leq \mathcal{F}_\sigma(p_2)$.*

3.2 Range Querying Using a BMC

Next, we present concepts on range query processing with BMCs.

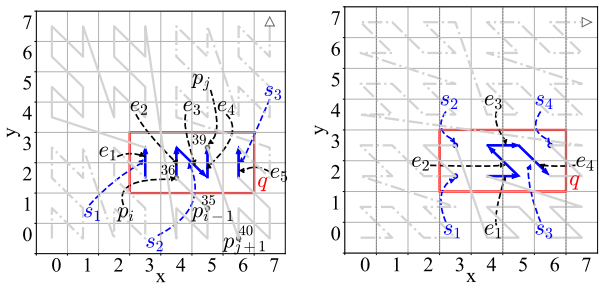
Definition 1 (Range Query). *Given a d-dimensional dataset D and a range query $q = [x_{s,1}, x_{e,1}] \times [x_{s,2}, x_{e,2}] \times \ldots \times [x_{s,d}, x_{e,d}]$, where $[x_{s,i}, x_{e,i}]$ denotes the query range in dimension i, query q returns all points $p = (x_1, x_2, ..., x_d) \in D$ that satisfy: $\forall i \in [1, d](x_{s,i} \leq x_i \leq x_{e,i})$.*

As mentioned earlier, computing a query q using different BMCs can lead to different costs. To simplify the discussion for determining the cost of a query, we use the following corollary.

Corollary 1. *Given $p_s = (x_{s_1}, \ldots, x_{s_d})$ and $p_e = (x_{e_1}, \ldots, x_{e_d})$, any query q is bounded by the curve value range $[\mathcal{F}_\sigma(p_s), \mathcal{F}_\sigma(p_e)]$.*

Corollary 1 follows directly from the monotonicity of BMCs. To simplify the discussion, we use a point p and the cell that encloses p interchangeably and rely on the context for disambiguation.

Query section [20]. A continuous curve segment in a query q is called a *query section*. We denote a query section s with end points p_i and p_j by $[\mathcal{F}_\sigma(p_i), \mathcal{F}_\sigma(p_j)]$, which translates to a data scan over the range $[\mathcal{F}_\sigma(p_i), \mathcal{F}_\sigma(p_j)]$. The number of query sections in $[\mathcal{F}_\sigma(p_s), \mathcal{F}_\sigma(p_e)]$ determines the cost of q.



(a) BMC XYXYXY (b) BMC YXYXYX

Figure 4: Query sections and directed edges in BMCs.

Example 1. *In Figure 4a, there are three query sections s_1, s_2, and s_3, with $s_2 = [\mathcal{F}_\sigma(p_i), \mathcal{F}_\sigma(p_j)] = [36, 39]$. By definition, a point (cell) immediately preceding p_i or succeeding p_j must be outside q; otherwise, it is part of the query section. For example, p_{i-1} ($\mathcal{F}_\sigma(p_{i-1}) = 35$) and p_{j+1} ($\mathcal{F}_\sigma(p_{j+1}) = 40$) in Figure 4a are outside q. The number of query sections in q varies across different BMCs, e.g., the same q as in Figure 4a has four query sections in Figure 4b.*

Directed edge [36]. A pair of two consecutive points p_i and p_j forms a *directed edge* e if the curve values of p_i and p_j differ by one under a given σ, i.e., $\mathcal{F}_\sigma(p_j) - \mathcal{F}_\sigma(p_i) = 1$. As each point is represented through a binary value, the difference occurs because $\mathcal{F}_\sigma(p_i) = \underbrace{\ldots}_{prefix}\underline{0}\underbrace{1\ldots1}_{K\ 1s}$ and $\mathcal{F}_\sigma(p_j) = \underbrace{\ldots}_{prefix}\underline{1}\underbrace{0\ldots0}_{K\ 0s}$, where the last $K \geq 0$ bits are changed from 1 to 0 and the $(K+1)$st bit from 0 to 1.

Example 2. *We use two examples to illustrate this concept, one for $K > 0$ and the other for $K = 0$. First, suppose that the binary representations of $\mathcal{F}_\sigma(p_i) = 15$ and $\mathcal{F}_\sigma(p_j) = 16$ are **0**$\underline{0}$**1111** and **0**$\underline{1}$**0000**, respectively. Four bits starting from the right (i.e., $K = 4$) are changed from 1 to 0, and the fifth bit from 0 to 1. The last bit **0** is the shared prefix. Second, if the binary forms of $\mathcal{F}_\sigma(p_i) = 16$ and $\mathcal{F}_\sigma(p_j) = 17$ are **01000**$\underline{0}$ and **01000**$\underline{1}$, respectively, only the first bit (from the right) is changed from 0 to 1, i.e., no bits ($K = 0$) are changed from 1 to 0, and the shared prefix is **01000**.*

The number of directed edges (denoted by $\mathcal{E}_\sigma(q)$) plus the number of query sections (denoted by $\mathcal{S}_\sigma(q)$) in a given query q yields the number of distinct points (denoted by $\mathcal{V}(q)$) in q:

$$\mathcal{E}_\sigma(q) + \mathcal{S}_\sigma(q) = \mathcal{V}(q) \tag{3}$$

Here, $\mathcal{V}(q)$ is independent of σ, while the values of $\mathcal{E}_\sigma(q)$ and $\mathcal{S}_\sigma(q)$ depend on σ. The intuition is that if q consists of a single section s, i.e., the curve stays completely inside s and $\mathcal{S}_\sigma(q) = 1$ then there are $\mathcal{V}(q) - 1$ directed edges connecting a given start point p_s and end point p_e of s. In other words, we obtain $\mathcal{E}_\sigma(q) + \mathcal{S}_\sigma(q) = \mathcal{V}(q) - 1 + 1 = \mathcal{V}(q)$. Further, each time a curve exits a query section s_i and enters the next section s_{i+1}, the last point in s_i becomes disconnected (minus one directed edge) but one new query section is added (plus 1 for the query section) when the curve reenters s_{i+1}. For example, in Figure 4a, there are 3 query sections ($\mathcal{S}_\sigma(q)$) and 5 directed edges ($\mathcal{E}_\sigma(q)$); in Figure 4b, there are 4 query sections and 4 directed edges in q. Both figures have $\mathcal{V}(q) = 8$ points in q.

4 EFFICIENT BMC COST ESTIMATIONS

When conducting a range query q with start point p_s and end point p_e, over a dataset D ordered by a BMC σ, a straightforward query method retrieves all data points within the range $[\mathcal{F}_\sigma(p_s), \mathcal{F}_\sigma(p_e)]$ and then filters out any false positives that fall outside q. The efficiency of this method relies on the number of data points in $[\mathcal{F}_\sigma(p_s), \mathcal{F}_\sigma(p_e)]$ and the clustering of data points within this range to eliminate false positives.

To measure BMCs effectively without executing real queries, we introduce two metrics: (i) the *global cost*, $C^g_\sigma(q)$, which measures the length of the range $[\mathcal{F}_\sigma(p_s), \mathcal{F}_\sigma(p_e)]$, reflecting the total span of data covered by q; and (ii) the *local cost*, $C^l_\sigma(q)$, which quantifies the number of query sections within q, reflecting the effectiveness of data clustering. These metrics are crucial for evaluating the impact of data organization on query efficiency.

Next, we present efficient algorithms for computing the global and local costs in Sections 4.1 and 4.2, respectively.

4.1 Global Cost Estimation for BMC

As mentioned above, we define the global cost of query q as the length of $[\mathcal{F}_\sigma(p_s), \mathcal{F}_\sigma(p_e)]$.

Definition 2 (Global Cost). *The global cost $C^g_\sigma(q)$ of query q under BMC σ is the length of the curve segment from p_s to p_e:*

$$C^g_\sigma(q) = \mathcal{F}_\sigma(p_e) - \mathcal{F}_\sigma(p_s) + 1 = \sum_{j=1}^{d}\sum_{k=1}^{\ell}(\alpha^k_{e,j} - \alpha^k_{s,j}) \cdot 2^{\gamma^k_j} + 1 \tag{4}$$

Efficient computation. Following the definition, given a set Q of n queries, their total global cost can be calculated by visiting every query $q \in Q$ and adding up $C^g_\sigma(q)$. This naive approach takes time proportional to the number of queries to compute. To reduce the time cost without loss of accuracy, we rewrite the global cost as a closed-form function for efficient computation.

$$\begin{aligned} C^g_\sigma(Q) &= \sum_{i=1}^{n} C^g_\sigma(q_i) = \sum_{i=1}^{n}\sum_{j=1}^{d}\sum_{k=1}^{\ell}\underbrace{(\alpha^k_{i,e,j} - \alpha^k_{i,s,j})}_{\text{BMC independent}} \cdot \underbrace{2^{\gamma^k_j}}_{\text{BMC dependent}} + n \\ &= \sum_{j=1}^{d}\sum_{k=1}^{\ell}\sum_{i=1}^{n}(\alpha^k_{i,e,j} - \alpha^k_{i,s,j}) \cdot 2^{\gamma^k_j} + n = \sum_{j=1}^{d}\sum_{k=1}^{\ell} A^k_j \cdot 2^{\gamma^k_j} + n \end{aligned} \tag{5}$$

Here, $q_i \in Q$; $\alpha^k_{i,s,j}$ and $\alpha^k_{i,e,j}$ denote the kth bits of the coordinates of the lower and the upper end points of q_i in dimension j, respectively; $A^k_j = \sum_{i=1}^{n}(\alpha^k_{i,e,j} - \alpha^k_{i,s,j})$, which is BMC independent and can be calculated once by scanning the n range queries in Q to compute the gap between p_e and p_s on the kth bit of the jth dimension, for any BMC. Only the term $2^{\gamma^k_j}$ is BMC dependent and must be calculated for each curve because γ^j_i represents the rank of the jth bit from dimension i of a BMC (cf. Section 3.1). If the BMC σ is changed, e.g., from XYXY**XY** to XYXY**YX**, then $\gamma^1_1 = 1$ and $\gamma^1_2 = 0$ are changed to $\gamma^1_1 = 0$ and $\gamma^1_2 = 1$, respectively.

Algorithm costs. The above property helps reduce the cost of computing the global cost when given multiple candidate BMCs, to learn the best BMC from a large volume of candidate BMCs. Without an efficient cost modeling, the global cost takes $O(m \cdot n \cdot d \cdot \ell)$ time for m candidate BMCs over n queries (based on Equation 4). Based on our proposed closed form (Equation 5), after an initial $O(n)$-time scan over the n queries (to compute A^k_j), the holistic global cost

over n queries can be calculated in $O(m \cdot d \cdot \ell)$ time to examine the "goodness" of a candidate BMC, i.e., $O(m)$ time given constant number of dimensions d and number of bits ℓ in each dimension.

4.2 Local Cost Estimation for BMC

The local cost measures the degree of segmentation of the curve in $[\mathcal{F}_\sigma(p_s), \mathcal{F}_\sigma(p_e)]$, which indicates the number of false positive data blocks that are retrieved unnecessarily and need to be filtered. We define the local cost as the number of query sections, following existing studies [18, 36] that use the term "*number of clusters*" for the same concept.

Definition 3 (Local Cost). *The local cost $C^l_\sigma(q)$ of query q under BMC σ is the number query sections in q, i.e., $\mathcal{S}_\sigma(q)$.*

Intuition. Recall that $\mathcal{V}(q)$ is the number of distinct points in q. We assume one data point per cell and that every B data points are stored in a block. A point is a true positive if it (and its cell) is in query q and a false positive if it is outside q but is retrieved by the query. If q has only one query section, the largest number of block accesses is $\lfloor(\mathcal{V}(q) - 2)/B\rfloor + 2$, i.e., only the first and last blocks can contain false positives (at least one true positive point in each block). In this case, the precision of the query process is at least $\frac{\mathcal{V}(q)}{\mathcal{V}(q)+2\cdot(B-1)}$. Following the same logic, if there are n_s query sections, in the worst case, each query section incurs two excess block accesses, each for a block containing only one true positive point. The largest number of block accesses is $\lfloor(\mathcal{V}(q) - 2 \cdot n_s)/B\rfloor + 2 \cdot n_s$, and the precision is $\frac{\mathcal{V}(q)}{\mathcal{V}(q)+2\cdot n_s\cdot(B-1)}$. The excess block accesses grows linearly with n_s. Thus, we use n_s to define the local cost.

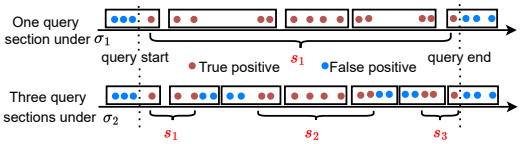


Figure 5: Query sections vs. block accesses

Example 3. *In Figure 5, we order points based on BMCs σ_1 and σ_2 and place the points in blocks where $B = 4$. There are 14 true positives (i.e., $\mathcal{V}(q) = 14$). There is only one query section under σ_1, which leads to a precision of $\frac{14}{5\times4} = 70\%$ for 5 block accesses, whereas σ_2 has three query sections (due to a different curve). The number of block accesses is 7, and the precision drops to $\frac{14}{7\times4} = 50\%$.*

Efficient computation. A simple way to compute the local cost of an arbitrary range query is to count the number of query sections by traversing the curve segment from p_s to p_e, but this is also time-consuming. To reduce the cost, we rewrite Equation 3 as:

$$\mathcal{S}_\sigma(q) = \mathcal{V}(q) - \mathcal{E}_\sigma(q) \tag{6}$$

Given a query q and the grid resolution of the data space, it is straightforward (i.e., taking $O(d) = O(1)$ time) to compute the number of cells in q (i.e., $\mathcal{V}(q)$). Then, our *key insight* is that $\mathcal{S}_\sigma(q)$ can be computed by counting the number of directed edges, i.e., $\mathcal{E}_\sigma(q)$, which can be done efficiently in $O(1)$ time as detailed below. Thus, $\mathcal{S}_\sigma(q)$ can be computed in $O(1)$ time.

4.2.1 Rise and Drop Patterns. To compute $\mathcal{E}_\sigma(q)$ efficiently, we analyse how the bit sequence of a BMC changes from one point to another following a directed edge. A directed edge is formed by two consecutive points with (binary) curve values that share the same *prefix*, while the remaining bits are changed. We observe that different directed edges have the same shape when they share the same pattern in their changed bits, even if their prefixes are different. In Figure 6a, consider edges $e_1 = (5, 6) = [\mathbf{000101}, \mathbf{000110}]$ and $e_2 = (13, 14) = [\mathbf{001101}, \mathbf{001110}]$. Both edges are in query q as indicated by the red rectangle, and they share the same '\' shape because the two rightmost bits in both cases change from "01" to "10". However, in Figure 6a, edge $(1, 2) = [\mathbf{000001}, \mathbf{000010}]$ is not in q, and the prefix ("0000") differs from that of e_1 and e_2 above.

A query q can only contain directed edges of a few different shapes. In Figure 6a, edge $(31, 32) = [011111, 100000]$ is not in q, and the pattern of the changed bits differs from that of e_1 and e_2.

Note that the bits of the curve values come from the coordinates (i.e., column indices) of the two end points of a directed edge. By analyzing the bit patterns of the column indices spanned by a query q in each dimension, we can count the number of directed edges that can appear in q.

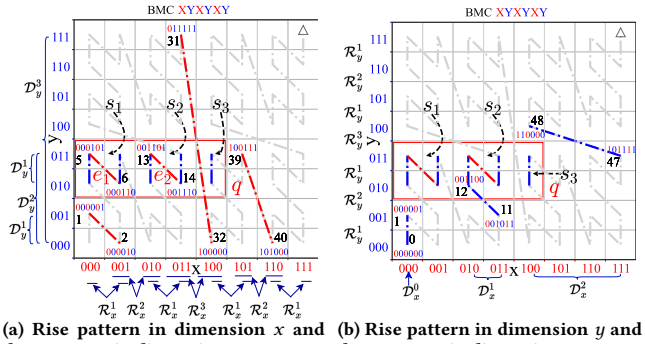
To generalize, recall that given a directed edge from p_i to p_j, $\mathcal{F}_\sigma(p_i) = \underbrace{\ldots}_{prefix}\,0\,\underbrace{1\ldots1}_{K\ 1s}$ and $\mathcal{F}_\sigma(p_j) = \underbrace{\ldots}_{prefix}\,1\,\underbrace{0\ldots0}_{K\ 0s}$ $(K \geq 0)$ must exist where the K rightmost bits are changed from 1 to 0, while the $(K+1)$st rightmost bit is changed from 0 to 1. The bits of $\mathcal{F}_\sigma(p_i)$ and $\mathcal{F}_\sigma(p_j)$ come from those of the column indices of p_i and p_j. Thus, the $K+1$ rightmost bits changed from $\mathcal{F}_\sigma(p_i)$ to $\mathcal{F}_\sigma(p_j)$ must also come from those of the column indices. In particular, there must be one dimension, where the column index has contributed k $(1 \leq k \leq K)$ changed bits and one of the bits has changed from 0 to 1, while the rest dimensions contribute bits changing from 1 to 0.

Our key observation is that the bit-changing patterns across the column indices in a dimension only depend on the column indices themselves, making them *BMC independent*. By pre-computing the number of bit-changing patterns that can form the $(K+1)$-bit change of a directed edge, we can derive efficiently the number of directed edges given a query q and a BMC.

We summarize the bit-changing patterns to form a directed edge with two basic patterns: a *rise pattern* and a *drop pattern*. A rise pattern $\mathcal{R}^k_b$ of a directed edge from p_i to p_j represents a k-bit $(k \geq 1)$ change in the dimension-b coordinate of p_i (i.e., $x_{i,b}$) to that of p_j (i.e., $x_{j,b}$), where the rightmost $k-1$ bits are changed from 1 to 0 and the kth bit (from the right) is changed from 0 to 1, i.e., $x_{i,b} = \underbrace{\ldots}_{prefix}\,0\,\underbrace{1\ldots1}_{(k-1)\ 1s}$ and $x_{j,b} = \underbrace{\ldots}_{prefix}\,1\,\underbrace{0\ldots0}_{(k-1)\ 0s}$. A drop pattern $\mathcal{D}^k_b$ of a directed edge from p_i to p_j represents a rightmost k-bit $(k \geq 0)$ 1-to-0 change in the dimension-b coordinate of p_i (i.e., $x_{i,b}$) to that of p_j (i.e., $x_{j,b}$), i.e., $x_{i,b} = \underbrace{\ldots}_{prefix}\,\underbrace{1\ldots1}_{k\ 1s}$ and $x_{j,b} = \underbrace{\ldots}_{prefix}\,\underbrace{0\ldots0}_{k\ 0s}$.

Given a dimension where the coordinates use ℓ bits, there can be ℓ different rise patterns, i.e., $k \in [1, \ell]$, and there can be $\ell + 1$ different drop patterns, i.e., $k \in [0, \ell]$. Note the *special case* where $k = 0$, i.e., $\mathcal{D}^0_b$, indicating no bit value drop in dimension b.

Example 4. *In Figure 6a, consider the directed edge from p_i to p_j, where $\mathcal{F}_\sigma(p_i) = 1$ (000001) and $\mathcal{F}_\sigma(p_j) = 2$ (000010), i.e., the '\' segment at the bottom left. The x-coordinate of p_i changes from 000 to 001 to that of p_j (i.e., rise pattern $\mathcal{R}^1_x$). The y-coordinate of p_i changes from 001 to 000 to that of p_j (i.e., drop pattern $\mathcal{D}^1_y$). Thus, this directed*



(a) Rise pattern in dimension x and drop pattern in dimension y.

(b) Rise pattern in dimension y and drop pattern in dimension x.

Figure 6: Example of forming a directed edge with rise and drop patterns: for BMC XYXYXY ($d = 2$ and $\ell = 3$), each directed edge is formulated by a rise and a drop pattern.

edge can be represented by a combination of $\mathcal{R}^1_x$ and $\mathcal{D}^1_y$, denoted as $\mathcal{R}^1_x \oplus \mathcal{D}^1_y$. This same combination also applies in other directed edges, such as that from $\mathcal{F}_\sigma(p_i) = 13$ to $\mathcal{F}_\sigma(p_j) = 14$, which is another '\'-shaped segment. Other directed edges may use a different combination, e.g., $\mathcal{R}^3_x \oplus \mathcal{D}^3_y$ for the one from $\mathcal{F}_\sigma(p_i) = 31$ to $\mathcal{F}_\sigma(p_j) = 32$, and $\mathcal{R}^2_x \oplus \mathcal{D}^2_y$ for the one from $\mathcal{F}_\sigma(p_i) = 39$ to $\mathcal{F}_\sigma(p_j) = 40$.

Figure 6a shows the rise patterns $\mathcal{R}^k_x$ in dimension-x and the drop patterns $\mathcal{D}^k_y$ in dimension-y, where a directed edge is in red. Similarly, Figure 6b shows the rise patterns $\mathcal{R}^k_y$ in dimension-y and the drop patterns $\mathcal{D}^k_x$ in dimension-x, where a directed edge is in blue. The *pattern combination operator* '$\oplus$' applied on rise and drop patterns means that a directed edge is formed by the two patterns.

Note also that while the rise and the drop patterns on a dimension are BMC independent, which ones that can be combined to form a directed edge is BMC dependent because different BMCs order the bits from different dimensions differently. For example, consider $\sigma = \mathrm{X}^3\mathrm{Y}^3\mathrm{X}^2\mathrm{Y}^2\mathrm{X}^1\mathrm{Y}^1$ (i.e., XYXYXY). From the right to the left of σ, the first rise pattern is $\mathcal{R}^1_x$. It can only be combined with drop pattern $\mathcal{D}^1_y$, as there is just one bit Y^1 from dimension-y to the right of X^1. Similarly, $\mathcal{R}^2_x$ and $\mathcal{R}^3_x$ can each be combined with $\mathcal{D}^2_y$ and $\mathcal{D}^3_y$, respectively, i.e., all 1-bits to the right of X^2 and X^3 must be changed to 0, according to the bit-changing pattern of a directed edge. In general, for each dimension, there are only ℓ valid combinations of a rise and a drop pattern, and this number generalizes to $d \cdot \ell$ in a d-dimensional space given a BMC.

Next, $\mathcal{E}_\sigma(q)$ can be calculated by counting the number of valid rise and drop patterns in q. For example, when $d = 2$:

$$\mathcal{E}_\sigma(q) = \sum_{i=1}^{\ell} \Big(\mathcal{N}(\mathcal{R}^i_x) \cdot \mathcal{N}(\mathcal{D}^{r_y}_y) + \mathcal{N}(\mathcal{R}^i_y) \cdot \mathcal{N}(\mathcal{D}^{r_x}_x) \Big) \tag{7}$$

Here, $\mathcal{N}(\cdot)$ counts the number of times that a pattern occurs in q, and r_x (r_y) is a parameter depending on the drop patterns that can be combined with $\mathcal{R}^i_x$ ($\mathcal{R}^i_y$). In Figure 6, for $q = ([0,4] \times [2,3])$, there are two $\mathcal{R}^1_x$, one $\mathcal{R}^2_x$, and one $\mathcal{R}^3_x$, i.e., $\mathcal{N}(\mathcal{R}^1_x) = 2$, $\mathcal{N}(\mathcal{R}^2_x) = 1$, and $\mathcal{N}(\mathcal{R}^3_x) = 1$. Next, there is one $\mathcal{D}^1_y$, zero $\mathcal{D}^2_y$, and zero $\mathcal{D}^3_y$ that are valid to match with these rise patterns, i.e., $\mathcal{N}(\mathcal{D}^1_y) = 1$, $\mathcal{N}(\mathcal{D}^2_y) = 0$, and $\mathcal{N}(\mathcal{D}^3_y) = 0$. Similarly, $\mathcal{N}(\mathcal{R}^1_y) = 1$, and $\mathcal{R}^1_y$ can be matched with $\mathcal{D}^0_x$, where $\mathcal{N}(\mathcal{D}^0_x) = 5$. Recall that $\mathcal{D}^0_x$ is the special case with no bit value drop. It is counted as the length of the query range in dimension x. Overall, $\mathcal{E}_\sigma(q) = 2 \times 1 + 1 \times 5$. Thus, there are $10 - 7 = 3$ query sections in q according to Equation 6, which is consistent with the figure.

Efficient counting of rise and drop patterns. A rise pattern $\mathcal{R}^k_b$ represents a change in the dimension-b coordinate from $x_{i,b} = a \cdot 2^k + (2^{k-1} - 1)$ to $x_{j,b} = a \cdot 2^k + 2^{k-1}$ ($a \geq 0 \wedge a \in \mathbb{N}$). Here, $a \cdot 2^k$ is the prefix, while $2^{k-1} - 1$ (i.e., $\underline{0}\ \underbrace{1\ldots1}_{(k-1)\ 1\mathrm{s}}$) and 2^{k-1} (i.e., $\underline{1}\ \underbrace{0\ldots0}_{(k-1)\ 0\mathrm{s}}$) represent the changed bits. Then, given the data domain $[x_{s,b}, x_{e,b}]$ of dimension b, each pattern can be counted by calculating $\lfloor (x_{e,b} - 2^{k-1})/2^k \rfloor - \lceil (x_{s,b} - (2^{k-1} - 1))/2^k \rceil + 1$, i.e., a bound on the different values of a, which takes $O(1)$ time. Similarly, a drop pattern $\mathcal{D}^k_b$ represents a change from $x_{i,b} = a \cdot 2^k + 2^k - 1$ to $x_{j,b} = a \cdot 2^k + 0$ ($a \geq 0 \wedge a \in \mathbb{N}$). Here, $2^k - 1$ (i.e., $\underbrace{1\ldots1}_{k\ 1\mathrm{s}}$) and 0 (i.e., $\underbrace{0\ldots0}_{k\ 0\mathrm{s}}$) represent the changed bits. We can count each pattern by calculating $\lfloor (x_{e,b} + 1)/2^k \rfloor - \lceil x_{s,b}/2^k \rceil$, again in $O(1)$ time.

Drop pattern collection. As mentioned a directed edge can be decomposed into a rise pattern and a drop pattern. To generalize to d dimensions, we call the set of all drop patterns in the $d - 1$ dimensions a *drop pattern collection* $\mathcal{D}^{k'}$, which represents the bit combination over $d - 1$ drop patterns: $\mathcal{D}^{\sum_{i=1,i\neq b}^{d-1} k_i} = \uplus_{i=1,i\neq b}^{d} \mathcal{D}^{k_i}_i$ ($k' = \sum_{i=1,i\neq b}^{d} k_i = K - k$), where b is the dimension with a rise pattern. Here, '$\uplus$' is a pattern combination operator (like $\oplus$ above). We note that $\mathcal{D}^{k'}$ and $\mathcal{D}^k_b$ are interchangeable if $d = 2$. For simplicity, we call $\mathcal{D}^{k'}$ a drop pattern when the context eliminates any ambiguity.

Now, in a d-dimensional data space, a directed edge can be formed by combining one rise pattern and $d - 1$ drop patterns, i.e., $\mathcal{R}^k_b \oplus \mathcal{D}^{\sum_{i=1,i\neq b}^{d} k_i} = \mathcal{R}^k_b \oplus (\uplus_{i=1,i\neq b}^{d} \mathcal{D}^{k_i}_i)$ where $k' = \sum_{i=1,i\neq b}^{d} k_i$. Equation 7 is then rewritten as:

$$\mathcal{E}_\sigma(q) = \sum_{j=1}^{d} \sum_{i=1}^{\ell} \mathcal{N}(\mathcal{R}^i_j) \cdot \mathcal{N}(\mathcal{D}^r) \tag{8}$$

Here, the value of parameter r depends on the number of drop patterns that can be combined with $\mathcal{R}^i_j$.

4.2.2 Pattern Tables. We have shown how to compute the local cost of a query efficiently. Given a set Q of n range queries ($q_i \in Q$), their total local cost based on Definition 3 is:

$$C^l_\sigma(Q) = \sum_{i=1}^{n} C^l_\sigma(q_i) = \sum_{i=1}^{n} \mathcal{V}(q_i) - \sum_{i=1}^{n} \mathcal{E}_\sigma(q_i) \tag{9}$$

This cost takes $O(n)$ time to compute. Given m BMCs, computing their respective total local costs $C^l_\sigma(Q)$ takes $O(m \cdot n)$ time. As $\sum_{i=1}^{n} \mathcal{V}(q_i)$ is independent of the BMCs, it can be computed once by performing an $O(n)$-time scan over Q. The computational bottleneck for m BMCs is then the computation of $\sum_{i=1}^{n} \mathcal{E}_\sigma(q_i)$.

We eliminate this bottleneck by introducing a look-up table called a *pattern table* that stores pre-computed numbers of rise-and-drop pattern combinations to form the directed edges at different locations, which are BMC-independent. As shown in Table 2, the pattern table for dimension b, denoted by $Table^b$, contains ℓ rows, each corresponding to a rise pattern in the dimension, and $\ell \cdot (d - 1) + 1$ columns, each corresponding to a drop pattern in the other $d - 1$ dimensions. The value in row i and column j is the product of the numbers of rise pattern $\mathcal{R}^i_b$ and drop pattern $\mathcal{D}^j$.

Table 2: Pattern table $Table^b$ for dimension b.

	$\mathcal{D}^0$	$\mathcal{D}^1$	$\cdots$	$\mathcal{D}^{\ell\cdot(d-1)}$
$\mathcal{R}_b^1$	$\mathcal{N}(\mathcal{R}_b^1)\cdot\mathcal{N}(\mathcal{D}^0)$	$\mathcal{N}(\mathcal{R}_b^1)\cdot\mathcal{N}(\mathcal{D}^1)$	$\cdots$	$\mathcal{N}(\mathcal{R}_b^1)\cdot\mathcal{N}(\mathcal{D}^{\ell\cdot(d-1)})$
$\mathcal{R}_b^2$	$\mathcal{N}(\mathcal{R}_b^2)\cdot\mathcal{N}(\mathcal{D}^1)$	$\mathcal{N}(\mathcal{R}_b^2)\cdot\mathcal{N}(\mathcal{D}^2)$	$\cdots$	$\mathcal{N}(\mathcal{R}_b^2)\cdot\mathcal{N}(\mathcal{D}^{\ell\cdot(d-1)})$
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
$\mathcal{R}_b^\ell$	$\mathcal{N}(\mathcal{R}_b^\ell)\cdot\mathcal{N}(\mathcal{D}^0)$	$\mathcal{N}(\mathcal{R}_b^\ell)\cdot\mathcal{N}(\mathcal{D}^1)$	$\cdots$	$\mathcal{N}(\mathcal{R}_b^\ell)\cdot\mathcal{N}(\mathcal{D}^{\ell\cdot(d-1)})$

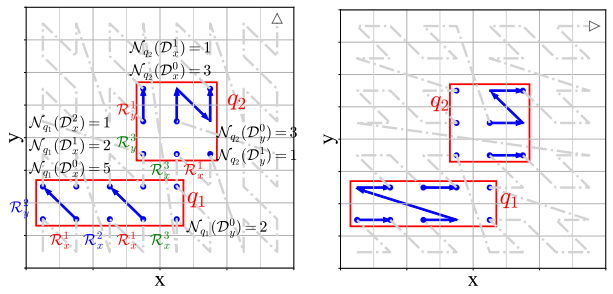
There is a total of $\ell\cdot(d-1)+1$ drop patterns in the $d-1$ dimensions because there are $\ell\cdot(d-1)$ bits in those dimensions, i.e., $k'\in[0,\ell\cdot(d-1)]$ for $\mathcal{D}^{k'}$. Further, since the rise and drop patterns correspond to only the bit sequences in each dimension and not the curve values, the values in the pattern tables can be computed once given a set of queries Q and can then be reused across local cost estimation for different BMCs. Algorithm 1 summarizes the steps to compute pattern table $Table^b$ based on its definition.

Algorithm 1: Generate pattern table (GPT)

Input: Query set Q, target dimension b, data dimensionality d, number of bits per dimension ℓ
Output: Pattern table $Table^b$

```
Initialize an ℓ × (ℓ · (d − 1) + 1) table Table^b;
for q ∈ Q do
    for i ∈ [1, ℓ] do
        for j ∈ [0, ℓ · (d − 1)] do
            N(R_b^i) ← count the number of R_b^i in q;
            N(D^j) ← count the number of D^j in q;
            Table^b[i][j] ← Table^b[i][j] + N(R_b^i) · N(D^j);
return Table^b;
```

Example 5. *In Figure 7a, we show two queries q_1 and q_2, and the pattern tables $Table^x$ and $Table^y$ are shown in Tables 3 and 4, respectively. In the tables, we use '+' to denote summing up the pattern table cell values (i.e., $\mathcal{N}(\mathcal{R}_b^i)\cdot\mathcal{N}(\mathcal{D}^j)$, and $\mathcal{N}(\mathcal{D}^j)$ is $\mathcal{N}(\mathcal{D}_x^j)$ or $\mathcal{N}(\mathcal{D}_y^j)$) computed for q_1 and q_2. For example, in q_1, $\mathcal{N}(\mathcal{R}_x^1)=2$ (the two $\mathcal{R}_x^1$ are labeled for q_1 in Figure 7a) and $\mathcal{N}(\mathcal{D}_y^0)=2$ (the value range of q_1 in dimension y is 2). Meanwhile, in q_2, $\mathcal{N}(\mathcal{R}_x^1)=1$ (one $\mathcal{R}_x^1$ is labeled for q_2 in Figure 7a) and $\mathcal{N}(\mathcal{D}_y^0)=3$ (the value range of q_2 in dimension y is 3). Thus, in $Table^x$, the cell $Table^x[1][0]$ (corresponding to $\mathcal{R}_x^1\oplus\mathcal{D}_y^0$) is the sum of $\mathcal{N}(\mathcal{R}_x^1)\cdot\mathcal{N}(\mathcal{D}_y^0)$ in q_1 and q_2, i.e., $4+3$.*



(a) Six directed edges (σ = XYXYXY) (b) Nine directed edges (σ = YXYXYX)

Figure 7: Example of pattern counting ($d=2$, $\ell=3$).

Table 3: $Table^x$

	$\mathcal{D}_y^0$	$\mathcal{D}_y^1$	$\mathcal{D}_y^2$	$\mathcal{D}_y^3$
$\mathcal{R}_x^1$	4 + 3	0 + 1	0 + 0	0 + 0
$\mathcal{R}_x^2$	2 + 0	0 + 0	0 + 0	0 + 0
$\mathcal{R}_x^3$	2 + 3	0 + 1	0 + 0	0 + 0

Table 4: $Table^y$

	$\mathcal{D}_x^0$	$\mathcal{D}_x^1$	$\mathcal{D}_x^2$	$\mathcal{D}_x^3$
$\mathcal{R}_y^1$	0 + 3	0 + 1	0 + 0	0 + 0
$\mathcal{R}_y^2$	5 + 0	2 + 0	1 + 0	0 + 0
$\mathcal{R}_y^3$	0 + 3	0 + 1	0 + 0	0 + 0

4.2.3 Local Cost Estimation with Pattern Tables. Next, we describe how to derive the number of directed edges (and hence compute the total local cost) given the d pattern tables for n queries.

Algorithm 2 shows how to compute the local cost using the pattern tables. Each dimension j is considered for the rise patterns (Line 2). Then, we consider each rise pattern in the dimension, i.e., each row i in $Table^j$ (Line 3). We locate the corresponding drop pattern (i.e., the table column index) based on i and a given BMC σ, which is done by the `get_col` function (Line 4). Then, we add the cell value to the number of directed edges $\mathcal{E}_\sigma$ (Line 5). Note that all ℓ rise patterns in each dimension are considered because a BMC has ℓ bits on each dimension, which can all be the bit that changes from 0 to 1. We return the total local cost by subtracting the total number of directed edges from the total number of cells in Q.

Algorithm 2: Compute local cost with pattern tables

Input: BMC σ, data dimensionality d, number of bits per dimension ℓ, all pattern tables $Table^j$, total number of cells in the queries $\mathcal{V}$
Output: Total local cost of n queries

```
1 E_σ ← 0;
2 for j ∈ [1, d] do
3     for i ∈ [1, ℓ] do
4         col ← get_col(σ, i, j);
5         E_σ ← E_σ + Table^j[i][col];
6 return V − E_σ;
```

Example 6. *Based on Example 5, given BMC XYXYXY, from $Table^x$, we read cells $(\mathcal{R}_1^1,\mathcal{D}_2^1)$, $(\mathcal{R}_1^2,\mathcal{D}_2^2)$, and $(\mathcal{R}_1^3,\mathcal{D}_2^3)$, i.e., the cells with "wavy" lines. Similarly, we read the cells with "wavy" lines from $Table^y$. These cells sum up to 6, which is the number of directed edges (segments with arrows) in Figure 7a. Similarly, the cells relevant to BMC YXYXYX are underlined, which yields a total of nine directed edges in Figure 7b.*

Algorithm costs. In general, for each rise pattern, the total number of possible drop pattern combinations is $(\ell+1)^{d-1}$ based on drop pattern collection. The time complexity of generating the d pattern tables is $O(d\cdot\ell\cdot(\ell+1)^{d-1})$, where d denotes the number of dimensions, ℓ denotes the number of rows, and $(\ell+1)^{d-1}$ denotes the accumulated number of drop patterns (equal to $(\ell+1)$ when $d=2$). After initialization, the retrieval time complexity of pattern tables is $O(d\cdot\ell)=O(1)$, i.e., we retrieve ℓ cells from each table.

We generate d pattern tables, each with $\ell\cdot(\ell+1)^{d-1}$ keys. Thus, the space complexity for the pattern tables is $O(d\cdot\ell\cdot(\ell+1)^{d-1})$. For example, when $d=3$ and $\ell=32$, all the tables take 1.6 MB (1.2 MB for keys and 0.4 MB for values).

Scalability to larger d. The value of d represents the number of coordinate dimensions used for data record ordering (e.g., data columns for data layout), which is typically not too large, e.g., $d=2$ for the Amazon reviews data storage [2].

As d increases, the storage space for our pattern tables could grow substantially. To use storage space more efficiently, when d becomes large (e.g., when $d\geq 5$ in our experiments), we fall back to aggregating the local cost of each single query (instead of using pre-computed look-up tables) to compute the local cost of multiple queries. Using this approach, for a d-dimensional query, the local cost computation time is $O(d\cdot\ell)$, i.e., to go over ℓ bits per dimension and calculate the combination of d patterns (one rise pattern and $d-1$ drop patterns) (see Section 4.2.1). The overall local cost computation time for n queries is then $O(n\cdot d\cdot\ell)$, without extra

storage space costs. Compared with the naive local cost computation process, which takes $O(n \cdot V)$ time, the above approach is still more efficient since V is proportional to ℓ^d (see Section 1).

5 COST ESTIMATION-BASED BMC LEARNING

Next, powered by our efficient cost estimations, we aim to find the optimal BMC σ_{opt} that minimizes the costs of a set of queries Q on a dataset D. While using BMCs reduces the number of curve candidates from $(2^\ell)^d!$ to $\frac{(d\cdot\ell)!}{(\ell!)^d}$ (Section 1), it is still non-trivial to find the optimal BMC from the $\frac{(d\cdot\ell)!}{(\ell!)^d}$ candidates. We present an efficient learning-based algorithm named *LBMC* for this search.

Problem transformation. Starting from any random BMC σ, the process to search for σ_{opt} can be seen as a bit-swapping process, until every bit falls into its optimal position, assuming an oracle to guide the bit-swapping process.

To reduce the search space, we impose two constraints: (a) we only swap two adjacent bits each time, and (b) two bits from the same dimension cannot be swapped (to obey the BMC definition). Any bit then takes at most $(d-1)\cdot\ell$ swaps to reach its optimal position if this position is known. Given $d\cdot\ell$ bits, at most $d\cdot(d-1)\cdot\ell^2$ swaps are needed to achieve the optimal BMC guided by an oracle.

In practice, an ideal oracle is unavailable. Now the problem becomes how to run the bit swaps without an ideal oracle. There are two approaches: (a) run a random swap (i.e., *exploration*) each time and keep the result if it reduces the query cost, and (b) select a position that leads to the largest query cost reduction each time (i.e., *exploitation*). Using either approach yields local optima. We integrate both approaches by leveraging *deep reinforcement learning* (DRL) to approach a global optimum, since DRL aims to maximize a long-term objective [15] and balance exploration and exploitation.

BMC learning formulation. We formulate BMC learning as a DRL problem: (1) **State space** $\mathcal{S}$, where a state (i.e., a BMC) $\sigma_t \in \mathcal{S}$ at time step t is a vector $\langle\sigma_t[d\cdot\ell], \sigma_t[d\cdot\ell-1], \ldots, \sigma_t[1]\rangle$, and $\sigma_t[i]$ is the ith bit. For example, if σ_t =XYZ, $\sigma_t[3]$=X, $\sigma_t[2]$=Y, and $\sigma_t[1]$=Z. (2) **Encoding function** $\phi(\cdot)$, which encodes a BMC to fit the model input. We use one-hot encoding. For example, X, Y, and Z can be encoded into $[0,0,1]$, $[0,1,0]$, and $[1,0,0]$, respectively, and XYZ by $[0,0,1,0,1,0,1,0,0]$. (3) **Action space** $\mathcal{A}$, where an action $a \in \mathcal{A}$ is the position of a bit to swap. When the ath bit is chosen, we swap it with the $(a+1)$st bit (if $a+1 \le d\cdot\ell$). Thus, $\mathcal{A} = \{a \in \mathbb{Z} : 1 \le a \le d\cdot\ell-1\}$. (4) **Reward** r: $\mathcal{S}\times\mathcal{A}\times\mathcal{S} \to r$, which is the query cost reduction when reaching a new BMC σ_{t+1} from σ_t. The reward r_t at step t is calculated as $r_t = (C_{\sigma_t} - C_{\sigma_{t+1}})/C_{\sigma_1}$, where $C_{\sigma_t} = C^g_{\sigma_t}(Q)\cdot C^l_{\sigma_t}(Q)$ is the cost of σ_t estimated by Equation 5 and Algorithm 2. Following QUILTS [20], our estimation of query cost for every state utilizes the product of global and local costs, but we differ in these two cost definitions (see Section 2). (5) **Parameter** ϵ, which balances exploration and exploitation to avoid local optima.

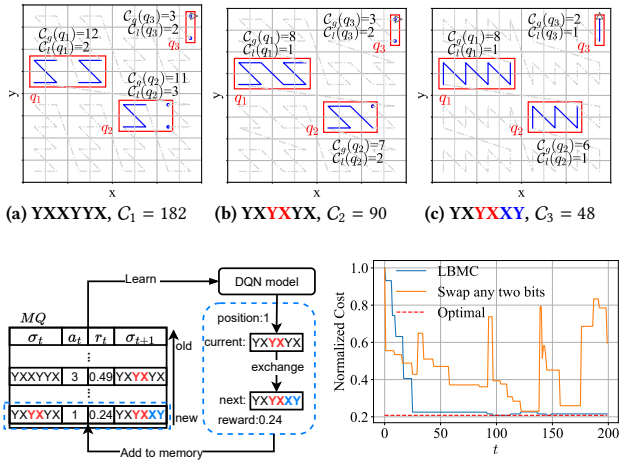
The LBMC algorithm. We summarize LBMC in Algorithm 3 where the input σ_1 can be any initial BMC, e.g., a ZC. The key idea is to learn a policy $\pi : \mathcal{S} \to \mathcal{A}$ that guides the position selection for a bit swap given a status, to maximize a value function $\mathbb{Q}^*(\phi(\sigma_t), a)$ (i.e., the reward) at each step t. Such a policy π can be learned by training a model (a *deep Q-network*, DQN [17]) with parameters θ over existing "*experience*" (previously observed state transitions and their rewards), which is used to predict the position a to maximize the value function (i.e., $\max_a \mathbb{Q}^*(\phi(\sigma_t), a; \theta)$). After a number of iterations, the learned BMC σ^*_{opt} is expected to approach σ_{opt}, which is returned as the algorithm output.

We use MQ to store the latest N_{MQ} swapping records (the experience, Line 1). We learn to approach σ_{opt} with M episodes and T steps per episode (Lines 2 and 3). In each episode, we start with σ_1 encoded by $\phi(\cdot)$. To select a swap position a_t at step t, we generate a random number in $[0,1]$, if it is greater than ϵ, we randomly select a position a_t, otherwise, we set a_t as the position with the highest probability to maximize the reward, i.e., $\max_a \mathbb{Q}^*(\phi(\sigma_t), a; \theta)$ (Line 4). The prediction is based on the current state σ_t and model weights θ. We execute a_t ($\mathsf{E}(\sigma_t, a_t)$) and compute reward r_t using our cost model (Line 5). We record the new transition in MQ and train the DQN (update θ) over sampled data in MQ (Lines 6 and 7). The training uses gradient descent to minimize a loss function $L_t(\theta_t) = \mathbb{E}_{\phi(\sigma),a\sim\rho(\cdot)}\left[(y_t - Q(\phi(\sigma), a; \theta_t))^2\right]$ where y_t is the target from iteration t and $\rho(\cdot)$ is the action distribution [17]. We use σ^*_{opt} to record the new BMC (Line 8), to be returned in the end (Line 9).

Algorithm 3: Learn BMC (LBMC)

Input: Initial BMC σ_1
Output: A query-efficient BMC σ^*_{opt}

1 Initialize replay memory MQ with capacity N_{MQ};
2 **for** *episode* $\in [1, M]$ **do**
3 **for** $t \in [1, T]$ **do**
4 With probability ϵ select a random position a_t, or $a_t \leftarrow \max_a \mathbb{Q}^*(\phi(\sigma_t), a; \theta)$;
5 $\sigma_{t+1} \leftarrow \mathsf{E}(\sigma_t, a_t)$, Compute reward r_t;
6 Store transition $(\phi(\sigma_t), a_t, r_t, \phi(\sigma_{t+1}))$ in MQ;
7 Train model θ with sampled transitions from MQ;
8 $\sigma^*_{opt} \leftarrow \sigma_{t+1}$;
9 **return** σ^*_{opt};



(a) YXXYYX, C_1 = 182 (b) YXYXYX, C_2 = 90 (c) YXYXXY, C_3 = 48

(d) Learning through LBMC (e) Normalized costs vs. # steps

Figure 8: A BMC learning example.

Example 7. *Figure 8 illustrates LBMC with $\ell = 3$ and three queries q_1, q_2, and q_3. The initial BMC σ_1 = YXXYYX has an (estimated) query cost of $C_1 = 182 = (12+11+3)\times(2+3+2)$ (Figure 8a). We select position $a_1 = 3$ and swap the 3rd and 4th bits to get σ_2 = YXYXYX with cost $C_2 = 90$ (Figure 8b). Next, we select position $a_2 = 1$ and swap the 1st and 2nd bits to get σ_3 = YXYXXY with cost $C_3 = 48$ (Figure 8c). We store the intermediate results in MQ for learning the*

DQN (Figure 8d, the BMCs are shown without encoding). Figure 8e shows the normalized costs, C_t/C_1, which decrease as t increases (Figures 8a to 8c are three of the steps) to approach the optimum.

We note that swapping adjacent bits rather than random pairs of bits can enhance the stability of the learning process, as the resulting curve patterns (and query costs) change gradually. Swapping random pairs of bits can lead to drastic changes in the curve patterns disrupting the locality structure, thereby hindering the model convergence. This is confirmed by Figure 8e that plots the estimated costs of the resulting curves learned with these two different strategies. The costs of LBMC decrease much more steadily.

Algorithm cost. LBMC involves $T \cdot M$ iterations that each involves three key operations: bit-swap position prediction, reward calculation (cost estimation), and model training. Their costs are $O(1)$, $O(C_t)$, and $O(\mathbb{T}_\theta)$, respectively. The total time cost is then $O(T \cdot M \cdot (1 + C_t + \mathbb{T}_\theta))$. Here, $T \cdot M$ is a constant, while $O(\mathbb{T}_\theta)$ is determined by the model structure. Our cost estimation results in $O(C_t) = O(1)$, thus enabling an efficient BMC search.

6 EXPERIMENTS

We evaluate: (i) our cost estimation efficiency and effectiveness in Experiment 1 (Section 6.2) and Experiment 2 (Section 6.3), respectively; (ii) the query efficiency of LBMC and BMTree i.e., the start of the art (Section 6.4) in Experiment 3; and (iii) the query efficiency achieved with the curves learned by LBMC in Experiment 4 (Section 6.5).

6.1 Experimental Settings

Our cost estimation algorithms (i.e., GC and LC) and BMC learning algorithm (i.e., LBMC) are implemented in Python, with TensorFlow facilitating the BMC learning. Additionally, we integrate BMC into Apache Hudi v0.14.0 using Java. We run experiments on a desktop computer equipped with 64-bit Ubuntu 20.04 with a 3.60 GHz Intel i9 CPU, 64 GB RAM, and a 500 GB SSD.

Datasets. We use three real datasets: **OSM** [21], **NYC** [31] and **TPC-H** [32], and one synthetic dataset **SKEW** [14]. OSM contains 100 million 2D location points (2.2 GB). NYC contains 150 million yellow taxi transactions (10.5 GB). TPC-H is generated by `dbgen`. We use the `lineitem` table (0.74 ~ 12.5 GB). SKEW contains 100 million points in skewed distribution (skewness 4, 1.7 GB) [27].

Experiment 2 (E2) and Experiment 3 (E3) leverages OSM and SKEW to replicate the conditions of the BMtree [14], assessing performance under skewed data distributions. Experiment 4 (E4) uses NYC and TPC-H to process real-world queries, facilitating an assessment in practical application contexts. Notably, Experiment 1 (E1) focuses on the efficiency of cost estimation, without the direct need for a dataset.

Queries. In E1, we evaluate performance using synthetic queries, varying both their number and ranges to simulate different workload scenarios. E2 extends this approach, employing 1,000 synthetic range queries for SFC learning and an additional 2,000 for testing purposes. The queries are of uniform size and follow the distributions of their respective datasets. E3 uses real query evaluation to assess the performance under practical queries. For TPC-H, queries are automatically generated alongside data tables. For NYC, we generate five query workloads (QW): large queries (QW1) spanning 1/6 (of the data range, same below) in each dimension; thin queries (QW2 and QW3) spanning 1/3 and 1/30 in the two dimensions, respectively, small queries (QW4) spanning 1/15 in each dimension, and skewed queries (QW5) with bottom left corners at that of the data space and random upper right corners.

Parameter settings. Table 5 summarizes the parameter values used, with default values in **bold**. In the table, n denotes the number of queries; δ denotes the edge length of a query; d denotes the data dimensionality; N denotes the dataset cardinality – we use sampling to obtain datasets of different cardinalities; and s indicates the `dbgen` scale factor, affecting the size of TPC-H (i.e., 6 million records for $s = 1$, and 96 million for $s = 16$).

A key parameter is the number of bits ℓ, which impacts the curve value mapping efficiency substantially. In E1, we restrict ℓ to 18 to suit a naive local cost baseline. In E2, we set default $\ell = 20$ following the BMTree to balance the computational costs of curve value mapping and cost estimation.

Table 5: Parameter settings.

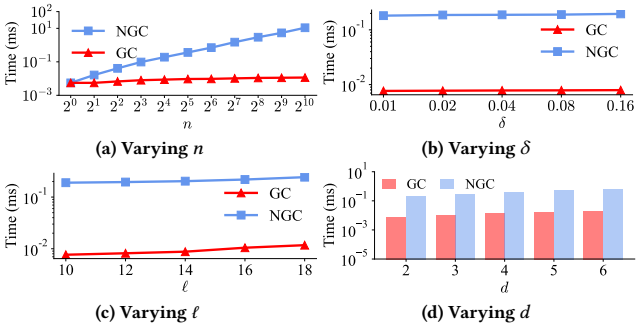
Experiments	Parameter	Values
E1	n	2^0, 2^1, 2^2, 2^3, **2^4**, 2^5, 2^6, 2^7, 2^8, 2^9, 2^{10}
	$\delta(\times 2^4)$	**1**, 2, 4, 8, 16
	ℓ	**10**, 12, 14, 16, 18
	d	**2**, 3, 4, 5, 6
	$\rho(\times 10^{-3})$	0.1, 0.5, 1, 5, 10
E2	N	10^4, 10^5, 10^6, **10^7**, 10^8
	$\rho(\times 10^{-3})$	0.1, 0.25, 0.5, 0.75, **1**, 2.5, 5, 7.5, 10
	h	5, 6, 7, 8, 9, **10**
	n	1, 5, 10, 50, 100, 500, **1000**, 1500, 2000
E3	s	1, 2, 4, 8, **16**
	ℓ	16, 20, 24, 28, 32, **64**
	d	**2**, 3, 4, 5, 6

6.2 E1: Efficiency of Cost Estimation

We first evaluate the efficiency of our algorithms (excluding initialization) to compute the global cost **GC** and the local cost **LC** (Algorithm 2), which are based on Equations 5 and 8. We use **IGC** and **ILC** to denote the initialization steps of the two costs, respectively. As there are no existing efficient algorithms to compute these costs, we compare with baseline algorithms based on Equations 4 and 9, denoted by **NGC** and **NLC**.

We vary the number of queries n, the query size (via δ), and the number of bits ℓ. We run experiments for 2- to 6-dimensional spaces. Due to page limits, we focus on the 2-dimensional space. As the cost estimation is data independent, a dataset is not needed to study their efficiency. The queries are generated at random locations.

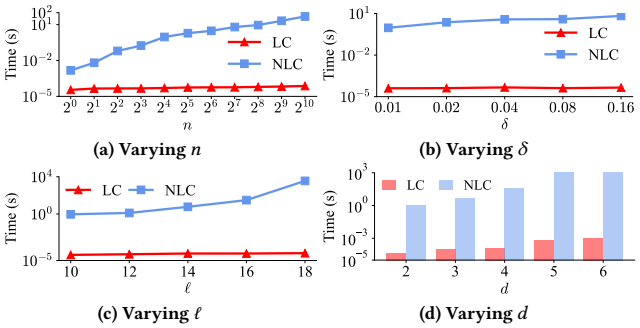
6.2.1 Efficiency of GC. Figures 9a and 9b show the impact of n and δ, respectively. Since GC takes $O(d \cdot \ell)$ time to compute (after the initialization step), its running time is unaffected by n and δ. NGC takes $O(n \cdot d \cdot \ell)$ time. Its running time grows linearly with n and is unaffected by δ as shown in the figures. Figure 9c shows that the running times of GC and NGC both increase with ℓ, which is consistent with their time complexities. Since the relative performance of our algorithm and the baseline is stable when ℓ is varied, we use a default value of 10 instead of the maximum value 18 as mentioned earlier, to streamline this set of experiments. Figure 9d shows the impact of d. The running times of both GC and NGC increase with d, which is also expected.



(a) Varying n (b) Varying δ (c) Varying ℓ (d) Varying d

Figure 9: Running times of global cost estimation.

Overall, GC is consistently faster than NGC, with up to more than an order of magnitude performance gain.



(a) Varying n (b) Varying δ (c) Varying ℓ (d) Varying d

Figure 10: Running times of local cost estimation.

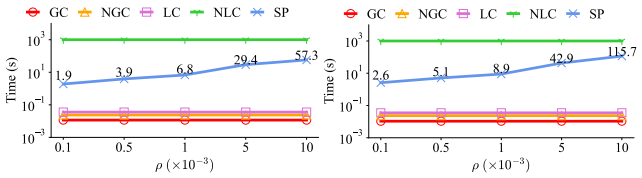
6.2.2 Efficiency of LC. Figure 10 shows the running times of computing local costs. The performance patterns of LC and NLC are similar to those observed above for GC and NGC, and they are consistent with the cost analysis in Section 4.2. The performance gains of LC are even larger, as its pre-computed pattern table enables extremely fast local-cost estimation. As Figure 10d shows, LC outperforms NLC by more than four orders of magnitude for all $d \in [2, 6]$. When $d \geq 5$, we aggregate the local cost of each query instead of using look-up tables as discussed in Section 4.2, which is still much faster than NLC, while avoiding a high space cost.

6.2.3 Initialization Costs of GC and LC. Table 6 shows the running times of IGC and ILC, which increase with n, because the initialization steps need to visit all range queries to compute a partial global cost and prepare the pattern tables, respectively. These running times are smaller than those of NGC and NLC, confirming the efficiency of the proposed cost estimation algorithms. Similar patterns are observed when varying δ, ℓ, and d, which are omitted for brevity. We do not report the result when $n = 2^0$ (i.e., $n = 1$) as no initialization is needed for a single query.

Table 6: ICs of GC and LC (Varying n).

n	2^1	2^2	2^3	2^4	2^5	2^6	2^7	2^8	2^9	2^{10}
IGC (ms)	**0.03**	**0.05**	**0.08**	**0.15**	**0.27**	**0.52**	**1.06**	**1.93**	**4.07**	**7.79**
NGC (ms)	0.03	0.05	0.10	0.18	0.36	0.70	1.50	2.96	5.37	10.86
ILC (s)	**0.01**	**0.01**	**0.02**	**0.06**	**0.12**	**0.23**	**0.48**	**0.95**	**1.83**	**3.63**
NLC (s)	0.01	0.06	0.18	0.93	1.93	3.03	6.31	9.21	20.98	48.22

6.2.4 Comparison with Sampling-based Cost Estimation. BMTree uses a data sampling-based (**SP**) strategy for fast query cost estimation. We compare our cost estimation algorithms with SP, varying the sampling rate ρ (using the datasets and default settings for the experiments on the BMTree). The running time of SP is measured by executing all queries after an intermediate BMTree is built. As Figure 11 shows, the running times of SP grow with ρ, as there are larger result sets to compute. The other cost estimation algorithms are dataset independent. Their running times stay unchanged. Compared with our LC and GC, SP is at least 53x and 164x slower when ρ is as small as 0.0001 on the SKEW dataset.



(a) SKEW (b) OSM

Figure 11: Comparison of SP with cost estimations as ρ varie.

6.3 E2: Effectiveness of Cost Estimation

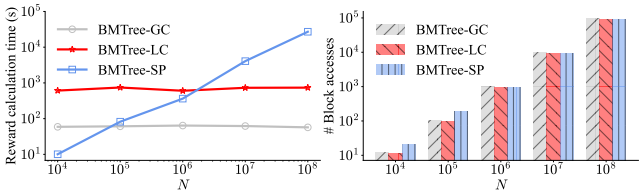
We next explore the applicability and effectiveness of our GC and LC cost estimations within a live database environment, specifically PostgreSQL, by comparing with the state-of-the-art SFC learning algorithm, the BMTree [14]. The BMTree originally uses a data sampling-based empirical cost estimation, referred to as **BMTree-SP**. We replace this built-in cost estimation by GC and LC, and denote the resulting variants by **BMTree-GC** and **BMTree-LC**, respectively. We note that, the BMTree has reported [14] superior query performance over traditional indices including ZC, HC, R-trees, R*-trees, STR-trees, Grid-files, quadtrees, and QUILTS [20] as well as learned indices ZM [34] and RSMI [26]. We repeated the experiments and observed that BMTree-GC and BMTree-LC also outperform these indices, which are omitted for brevity.

We report the reward calculation time during the learning process, as the other steps of the three variants are the same. For query execution, data is loaded into PostgreSQL, and we report the average number of block accesses as recorded by PostgreSQL.

6.3.1 Varying the Dataset Cardinality. We start by varying the dataset cardinality N from 10^4 to 10^8. Figure 12 shows the results on the OSM dataset (the results on the other datasets show similar patterns and are omitted for brevity; same below). BMTree-GC and BMTree-LC have constant reward calculation times, since GC and LC are computed in constant times. In comparison, the reward calculation time of BMTree-SP increases linearly with the dataset cardinality, as BMTree-SP builds intermediate index structures based on sampled data points for query cost estimation. When N increases, the number of sampled data points also increases. At $N = 10^8$ (the default sampling rate is $\rho = 0.001$, i.e., BMTree-SP is run on a sampled set of 10^5 points), the reward calculation time of BMTree-SP (more than 7 hours) is 36x and 474x higher than those of BMTree-LC (737 s) and BMTree-GC (57 s).

In terms of the query costs, learning the BMTree with all three algorithms requires more block accesses as N increases, which is expected. Importantly, all three algorithms incur similar numbers

of block accesses given the same N value. This suggests that the GC and LC cost estimations can be applied to improve the curve learning efficiency of the BMTree without adverse effects on the query efficiency. In general, BMTree-LC offers lower query costs than BMTree-GC. Thus, applications that are more sensitive to query costs may use BMTree-LC, while those that are more sensitive to learning costs may use BMTree-GC.



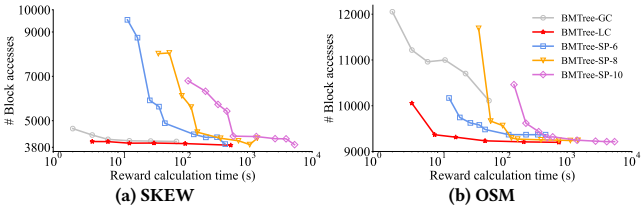
(a) Reward calculation time **(b) Query processing cost**

Figure 12: Varying the dataset cardinality (OSM).

6.3.2 Varying the Sampling Rate and the Depth of the BMTree. Two alternative approaches to improve the curve learning efficiency of the BMTree are (1) to reduce its data sampling rate ρ and (2) to reduce the depth of its space partitioning h.

In this set of experiments, we study how these two parameters impact the reward calculation time and the query cost of the resulting SFCs. In particular, we vary ρ from 10^{-4} to 10^{-2} (a total of 9 values, cf. Table 5), and we vary h from 5 to 10. Figure 13 plots the results on the SKEW and OSM datasets. BMTree-SP has three result polylines: BMTree-SP-6, BMTree-SP-8, and BMTree-SP-10, each of which uses a different h value, while the points on each polyline represent the results of different ρ values (points on the right come from larger ρ values).

BMTree-GL and BMTree-LC are plotted with one polyline each, as they are not impacted by ρ. The points on these polylines represent the results of different values of h (points on the right correspond to larger h values). We see that a larger h value tends to lead to lower query costs, while it also yields a longer reward calculation time. Powered by the LC cost estimation algorithm, BMTree-LC reduces the reward calculation time by at least an order of magnitude while achieving the same level of query costs (i.e., its curve lies at the bottom left of the figure). BMTree-GC can also be very fast at reward calculation, while it may suffer at query performance.



(a) SKEW **(b) OSM**

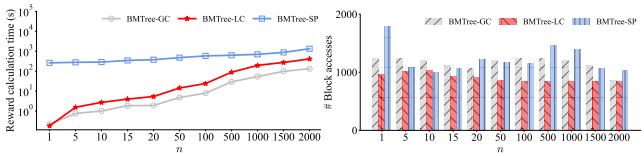
Figure 13: Varying the sampling rate and the space partitioning depth of the BMTree.

6.3.3 Varying the Number of Queries for Curve Learning. Next, we study the impact of the number of queries used in curve learning (the "*training queries*" for short), varying n from 1 to 2,000. To add diversity to the queries, we now randomly generate the query side length between 1/16384 and 1/64 of the data range.

Figure 14a shows the reward calculation times. All three algorithms take more time as n grows, because their reward calculations all need to go through all n queries at least once. BMTree-GC and BMTree-LC are consistently faster than BMTree-SP. The performance gap shrinks with n, because BMTree-SP takes extra time to build an intermediate index, which gets amortized as n grows.

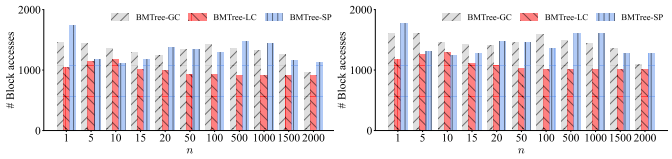
Figure 14b shows the average number of block accesses over 2,000 test queries (following the distribution of the training queries). The algorithms show different performance patterns. (1) BMTree-LC shows a steadily decreasing pattern starting from $n = 10$, with its query costs being lower than those of BMTree-SP from $n \geq 50$. This shows that our LC can be highly effective to guide learning query-efficient curves given just a small n. (2) Both BMTree-GC and BMTree-SP fluctuate more, with a more substantial drop only when n reaches 1,500, which is less desirable.

6.3.4 Varying the Distribution of the Test Queries. In Figure **??**, we show the query costs when the test queries follow a different distribution from that of the training queries. The centroids of the training queries are sampled from the dataset, while those of the test queries are sampled from a normal distributions for 50% and all of the test queries in Figures 15b and 15d, respectively. We see that the query costs increase as the test queries become more different from the training queries. For example, at $n = 2,000$, the number of block accesses for BMTree-LC are 1,013 and 1,179 in the two sub-figures, respectively. This is expected, as the curves learned suit the test queries less and less. Importantly, the relative performance among the three algorithms is stable across the sub-figures, i.e., LC is consistently at least as effective as the sampling-based cost estimator proposed by the BMTree.

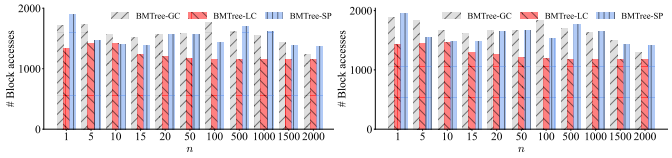


(a) Reward calculation time. **(b) Query processing cost**

Figure 14: Varying the number of training queries (OSM).



(a) 75% test queries from training distribution **(b) 50% test queries from training distribution**



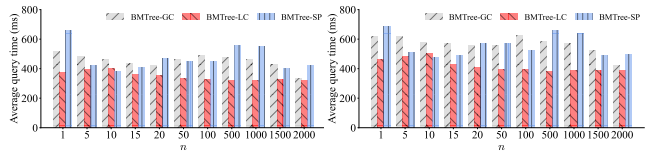
(c) 25% test queries from training distribution **(d) No test queries from training distribution**

Figure 15: Varying the number of queries (OSM).

6.3.5 Extension to In-memory System. We further conduct experiments on a in-memory database system, Redis. This system supports a data structure named sorted sets, which maintain the set elements in a certain order based on their *scores*. Naturally, we can use the

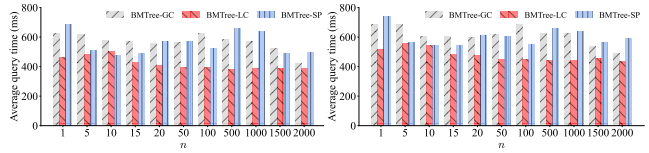
curve values as the scores and run in-memory experiments. As there is no block access in Redis, we report the average query time.

Figure 16 shows the results. We observe patterns in query times that are consistent with those observed under disk-based implementations. Using our LC algorithm, BMTree-LC again yields the lowest query costs. BMTree-GC can also learn to optimize the query costs of the curves, although it takes more training queries (e.g., 2,000). These results confirms the effectiveness of our cost estimation algorithms under in-memory settings.



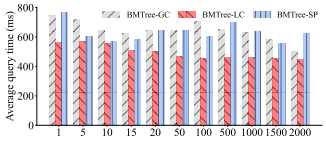
(a) All test queries from training distribution

(b) 75% test queries from training distribution



(c) 50% test queries from training distribution

(d) 25% test queries from training distribution



(e) No test queries from training distribution

Figure 16: Varying the number of queries on Redis (OSM).

6.4 E3: Comparison between LBMC and BMTree

We proceed to compare the query performance of **LBMC** and the BMTree, both of which use a learning based technique. Additionally, we compare these techniques with traditional baselines such as QUILTS, CC, ZC, and HC. For all techniques, we use the curves obtained to order the data points and build B^+-trees in PostgreSQL for query processing, and we report the average number of block accesses as before (cf. Section 6.3).

6.4.1 Varying the Dataset Cardinality. We further study the impact of dataset cardinality N. Figure 17 shows the results. Like before, the average number of block accesses increases with N, which is expected. LBMCis again the most efficient in terms of query costs, needing at least 39% fewer block accesses than the BMTree (4.0 vs. 6.6 when $N = 10^4$), and the advantage is up to 74% (1,044 vs. 4,131 when $N = 10^7$).

We report the SFC learning times of the BMTree and LBMC when varying N in Table 8. We see that LBMC is much faster than the BMTree at SFC learning and that the advantage grows with N. This is because the cost estimation (i.e., reward calculation) in the BMTree is much slower than that in LBMC, as shown in the last subsection. The cost estimation time dominates when there are more data points for the BMTree, while the cost estimation time of LBMC remains constant when varying N.

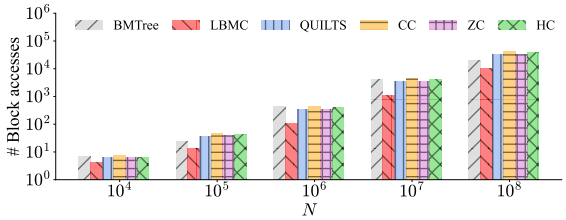


Figure 17: Varying cardinality (OSM).

CC, ZC, and HC are not learned, and they do not take any learning time. QUILTS takes less than 1 second, as it only considers a few curve candidates (which are generated based on query shapes) using a cost model. We have used our cost estimation algorithms in our implementation of QUILTS, as the original cost model is prohibitively expensive.

Table 7: SFC learning time (seconds).

N	10^4	10^5	10^6	10^7	10^8
BMTree	54	55	61	99	551
LBMC	15	15	15	15	15
QUILTS (with our cost estimation)	0.2	0.2	0.2	0.2	0.2

6.4.2 Varying the Aspect Ratio of Queries. Figure 18 shows the query costs when varying the query aspect ratio. Here, LBMC shows a stronger advantage over the competitors on queries that are "stretched", while CC also better suits the queries that are long and thin (16:1) which is intuitive. When the aspect ratio is 1 : 1, LBMC, QUILTS, and ZC share almost the same query performance because they all tend to form a 'Σ' shape to fit square queries. The BMTree is again outperformed by LBMC, because of its less flexible learning scheme (i.e., learning for only up to h bits), while LBMC can learn a BMC scheme with all ℓ bits ($\ell = 20$ by default).

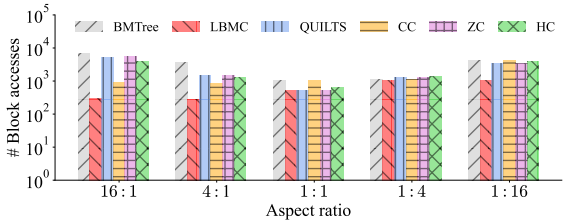


Figure 18: Varying the query aspect ratio (OSM).

6.4.3 Varying the Edge Length of Queries. Figure 19 shows that the average number of block accesses grows with the query edge length, as expected. Here, LBMC again outperforms the competitors consistently, further showing the robustness of LBMC.

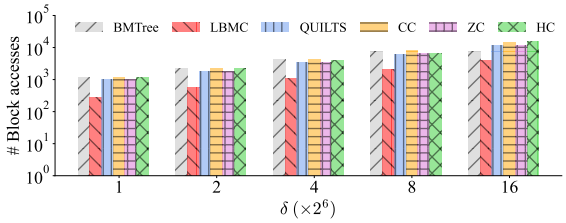


Figure 19: Varying the query edge length (OSM).

6.5 E4: Improvement on Query Efficiency

We investigate the query efficiency of using the BMCs learned by **LBMC** to order data points, comparing with other ordering techniques within Hudi [2]. Using Hudi is motivated by its inherent support for a variety of data ordering techniques to compute the data layout: **ZC** [22], **HC** [11], and **CC** [14, 20]. In Hudi, values in data columns chosen to compute the data layout are each converted into an 8-byte integer (with truncation if needed). The converted values of each data record are mapped to a one-dimensional value using an SFC, which is then used for data ordering and layout.

We introduce BMC (i.e., the output of LBMC or a baseline algorithm QUILTS [20]) into Hudi by adding a BMC-based ordering to Hudi. The BMTree is excluded because it cannot be easily integrated into Hudi due to its complex structure with multiple BMCs. cannot compare with the recent learned SFC, LMSFC [8], because its source code and some implementation details are unavailable

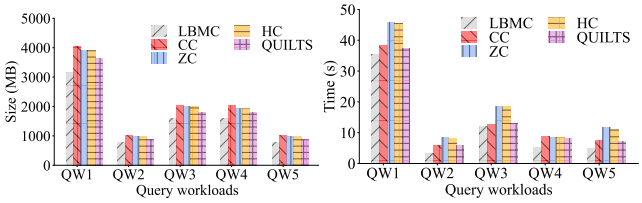
Our evaluation involves five distinct query workloads on TPC-H and NYC datasets, comprising 1,000 queries each. These queries, while uniform in SQL structure, vary in their query ranges and conditions specified within the `WHERE` clause.

We use Spark WebUI APIs [29] to measure *the average size of data files scanned* per query that indicates the direct benefits of data ordering and *the average CPU time* per query.

6.5.1 Results on TPC-H. We sort the records using the `commitdate` and `receiptdate` columns of the `lineitem` table in TPC-H. This boosts the efficiency of queries that often use these columns.

Overall Results. Figure 20 shows the results. In terms of the average size of data files scanned, LBMC outperforms all competitors consistently. It saves at least 17% of the data scans compared with those of ZC and HC. Compared with QUILTS, LBMC also improves by at least 11%. CC has the highest data volume scanned, exceeding that of LBMC by over 21%. This is because the linear ordering intersects with a broader range than the others.

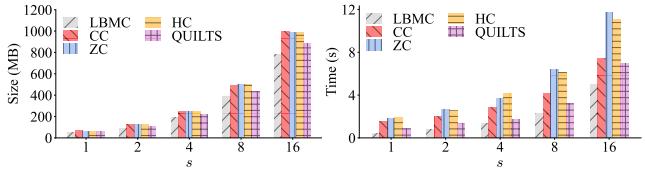
Regarding the average CPU time, LBMC is also consistently the lowest. It saves more than 21% time comparing wit ZC and HC. Compared with QUILTS (8.14 s), LBMC (5.13 s) saves up to 36% of the CPU time on QW4. Here, the CPU time of CC is no longer the worst – CC enables more efficient aggregation (e.g., "`ORDER BY`") after the data scans, as data is already fully sorted in a column. While ZC and HC may reduce the data scans, they may need more expensive re-ordering on specific column(s) during aggregation.



(a) Average size of data files scanned **(b) Average CPU time**

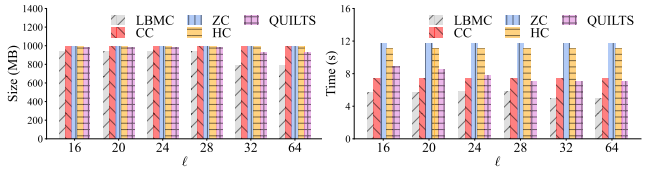
Figure 20: Querying TPC-H.

Varying dataset cardinality. We vary N by varying the scale factor s of `dbgen`. Figure 21 shows the results on QW5 (similar results are observed on other query workloads, same below). As s increases, the query costs increase as expected. LBMC again has the lowest query costs due to its strong clustering capability.



(a) Average size of data files scanned **(b) Average CPU time**

Figure 21: Varying the scale of TPC-H (QW5).



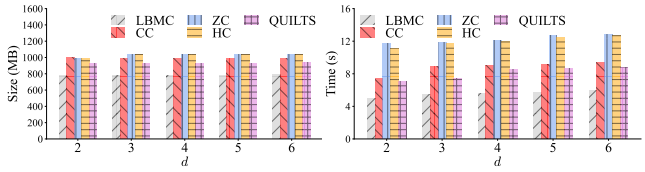
(a) Average size of data files scanned **(b) Average CPU time**

Figure 22: Varying the number of bits on TPC-H (QW5).

Varying the number of bits. We vary the number of bits ℓ used in LBMC and QUILTS. CC, HC, and ZC use $\ell = 64$ which is inbuilt and fixed by Hudi. Figure 22 shows the results (for workload QW5). We observe a query cost reduction for LBMC and QUILTS when $\ell = 32$. This is the point when the use of more bits substantially reduces duplicates in the curve values, and hence fewer false positives are retrieved at query time. When ℓ increases further to 64, the query costs have not reduced further. This suggests that $\ell = 32$ is sufficient for curve construction over the TPC-H dataset.

Varying data dimensionality. To test the impact of d, we use the `lineitem` table in TPC-H and workload QW5 like above. We now use up to six columns: `commitdate`, `receiptdate`, `shipdate`, `linenumber`, `quantity`, and `discount`, starting from `commitdate` and `receiptdate` (i.e., $d = 2$). As d increases by one, we append an extra random query range to every query in QW5.

Figure 23 shows the query costs of the computed curves. As d increases, the average size of data files scanned, as well as the performance gap among the algorithms, is stable. The average CPU times, on the other hand, grow with d due to the increased time costs in filtering the data records. LBMC has the lowest query costs consistently, confirming its scalability with d.



(a) Average size of data files scanned **(b) Average CPU time**

Figure 23: Varying the data dimensionality on TPC-H.

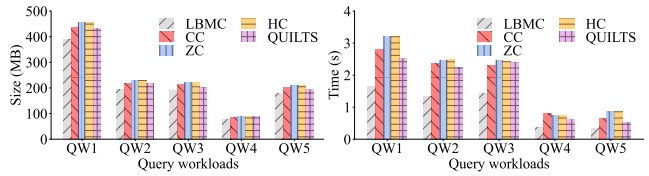
BMC optimization time. Table 8 lists the times taken by QUILTS and LBMC to compute their optimized curves (ZC, HC, and CC do not need this time). QUILTS has a varying computation time, because its local cost is computed based on the query sizes which vary across the workloads. LBMC has a much lower and constant time, which is consistent with our $O(1)$-time promise.

Table 8: BMC optimization time (seconds).

Query workloads	QW1	QW2	QW3	QW4	QW5
QUILTS	1142	266	413	484	257
LBMC	13	13	13	13	13

6.5.2 Results on NYC. Next, we run queries on NYC sorted based on the `pickup_location` and `dropoff_location` columns.

Overall results. As Figure 24 shows, LBMC consistently outperforms all the other sorting techniques, achieving data scan reductions by more than 10%, 14%, 14%, and 6% over CC, ZC, HC, and QUILTS, respectively. Here, CC outperforms ZC and HC on four workloads (except for QW4). This is because queries in the four workloads each spans quite a large range in at least one dimension. Such queries retrieve data in a large consecutive range. CC-based data layout suits this retrieval pattern nicely and hence CC performs the best for these workloads. LBMC also reduces the CPU time by at least 35% (1.63 s for LBMC vs. 2.53 s for QUILTS on QW3) and up to 61% (0.34 s for LBMC vs. 0.86 s for HC on QW5). Here, the CPU time of CC is also slightly better since it is more efficient for aggregations (e.g., "ORDER BY" a dimension) after the data scans, while ZC and HC may take extra time to re-order on a dimension during aggregation.



(a) Average size of data files scanned **(b) Average CPU time**

Figure 24: Querying NYC.

7 EXTEND TO OTHER CURVES AND QUERIES

We discuss extensions to other types of curves and queries.

Other types of curves. The idea of counting the number of times that a curve enters and leaves a query range to derive the number of curve segments in the query range (i.e., the local cost) could apply to any SFC. The challenge lies in how to count efficiently. Our LC algorithm exploits the property that the BMC curve value of a cell is determined by the bits of the coordinates (column indices) of the cell. This property ensures that curve moving patterns can be derived and counted efficiently based on the changes in the bits of the coordinates of the boundary of a query range.

Such a property does *not* hold for Hilbert curves, which are formed by a (recursive) combination of pattern '⊓' and its rotations. To count the curve segments in a query, we need to count the number of times that pattern '⊓' and its different rotations intersect the query boundary, which is not directly computable from the cell coordinates. This challenge necessitates another study to find efficient algorithms to compute the number of intersections.

Other types of queries. We focused on range queries following the literature [14, 20, 36] and for that they are a basic query type. We discuss how to extend to k nearest neighbor (kNN) and spatial join queries which are also commonly used.

KNN queries are typically processed as a series of range queries with growing ranges, until no new kNN objects can be found. The challenge lies in estimating the number of range queries needed and the ranges of such queries, which could be done based on the data density around the query point (e.g., using the cumulative distribution functions). For spatial join queries, consider range joins over two datasets D_A and D_B. Then, data points (extended by the join range) from D_A can be seen as a set of range queries to be run on D_B, such that our cost estimation algorithms apply. Given more advanced join algorithms or predicates, e.g., *R-tree-join* or kNN join, more complex filtering may apply to reduce the query costs. In this case, our algorithms may be used to derive a cost upper bound.

8 CONCLUSIONS

We studied efficient cost estimation for a family of SFCs, i.e., the BMCs. Our cost algorithms can compute the global and the local query costs of BMCs in constant time given n queries. We extended these algorithms to the state-of-the-art curve learning algorithm, the BMTree, which originally measured the effectiveness of SFCs. Experimental results show that our algorithms can reduce the cost estimation time of the BMTree by over an order of magnitude with little or no impact on the query efficiency of the learned curves. We further proposed a reinforcement learning-based curve learning algorithm. The resulting learned BMCs are shown to achieve lower query costs than other baselines in a real data lake system.

REFERENCES

[1] Amazon AWS. 2016. *https://aws.amazon.com/blogs/big-data/amazon-redshift-engineerings-advanced-table-design-playbook-compound-and-interleaved-sort-keys.* Accessed: 2024-02-27.

[2] Apache Hudi. 2021. *https://hudi.apache.org/blog/2021/12/29/hudi-zorder-and-hilbert-space-filling-curves.* Accessed: 2024-02-27.

[3] Christian Böhm. 2020. Space-filling Curves for High-performance Data Mining. *CoRR* abs/2008.01684 (2020).

[4] Fan R. K. Chung. 1997. *Spectral Graph Theory.* CBMS Series in Mathematics No. 92, American Mathematical Society.

[5] Databricks Engineering Blog. 2018. *https://databricks.com/blog/2018/07/31/processing-petabytes-of-data-in-seconds-with-databricks-delta.html.* Accessed: 2024-02-27.

[6] Christos Faloutsos and Shari Roseman. 1989. Fractals for Secondary Key Retrieval. In *PODS.* 247–252.

[7] 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.

[8] Jian Gao, Xin Cao, Xin Yao, Gong Zhang, and Wei Wang. 2023. LMSFC: A Novel Multidimensional Index based on Learned Monotonic Space Filling Curves. *PVLDB* 16, 10 (2023), 2605–2617.

[9] Claire E. Heaney, Yuling Li, Omar K. Matar, and Christopher C. Pain. 2020. Applying Convolutional Neural Networks to Data on Unstructured Meshes with Space-Filling Curves. *CoRR* abs/2011.14820 (2020).

[10] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. 2011. Sequential Model-Based Optimization for General Algorithm Configuration. In *International Conference on Learning and Intelligent Optimization.* 507–523.

[11] Ibrahim Kamel and Christos Faloutsos. 1994. Hilbert R-tree: An Improved R-tree using Fractals. In *VLDB.* 500–509.

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

[13] Warren M. Lam and Jerome M. Shapiro. 1994. A Class of Fast Algorithms for the Peano-Hilbert Space-Filling Curve. In *International Conference on Image Processing.* 638–641.

[14] Jiangneng Li, Zheng Wang, Gao Cong, Cheng Long, Han Mao Kiah, and Bin Cui. 2023. Towards Designing and Learning Piecewise Space-Filling Curves. *PVLDB* 16, 9 (2023), 2158–2171.

[15] Stephen McAleer, Forest Agostinelli, Alexander Shmakov, and Pierre Baldi. 2019. Solving the Rubik's Cube Without Human Knowledge. In *ICLR.*

[16] Microsoft. 2023. *https://learn.microsoft.com/en-us/sql/relational-databases/indexes/indexes?view=sql-server-ver16.* Accessed: 2024-02-27.

[17] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. 2013. Playing Atari with Deep Reinforcement Learning. *CoRR* abs/1312.5602 (2013).

[18] Bongki Moon, H. 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.

[19] Andrew Ng, Michael Jordan, and Yair Weiss. 2001. On Spectral Clustering: Analysis and an algorithm. In *NIPS*, T. Dietterich, S. Becker, and Z. Ghahramani (Eds.), Vol. 14.

[20] Shoji Nishimura and Haruo Yokota. 2017. QUILTS: Multidimensional Data Partitioning Framework Based on Query-Aware and Skew-Tolerant Space-Filling Curves. In *SIGMOD.* 1525–1537.

[21] OpenStreetMap. 2018. *OpenStreetMap North America data dump. https://download.geofabrik.de.* Accessed: 2024-02-27.

[22] Jack A. Orenstein. 1986. Spatial Query Processing in an Object-Oriented Database System. In *SIGMOD*. 326–336.

[23] Jack A. Orenstein and T. H. Merrett. 1984. A Class of Data Structures for Associative Searching. In *PODS*. 181–190.

[24] Sachith Pai, Michael Mathioudakis, and Yanhao Wang. 2022. Towards an Instance-Optimal Z-Index. In *AIDB@VLDB*.

[25] PostgreSQL. 2023. *https://www.postgresql.org/docs/current/indexes-multicolumn.html.* Accessed: 2024-02-27.

[26] Jianzhong Qi, Guanli Liu, Christian S. Jensen, and Lars Kulik. 2020. Effectively Learning Spatial Indices. *PVLDB* 13, 11 (2020), 2341–2354.

[27] Jianzhong Qi, Yufei Tao, Yanchuan Chang, and Rui Zhang. 2018. Theoretically Optimal and Empirically Efficient R-trees with Strong Parallelizability. *PVLDB* 11, 5 (2018), 621–634.

[28] S2 Geometry. 2023. *http://s2geometry.io.* Accessed: 2024-02-27.

[29] Spark Web UI. 2024. *https://spark.apache.org/docs/latest/web-ui.html.* Accessed: 2024-02-27.

[30] Daniel A. Spielman and Shang-Hua Teng. 2007. Spectral partitioning works: Planar graphs and finite element meshes. *Linear Algebra Appl.* 421, 2 (2007), 284–305.

[31] TLC Trip Record Data. 2022. *https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page.* Accessed: 2024-02-27.

[32] TPC. 2022. *TPC-H. http://www.tpc.org/tpch/.* Accessed: 2024-02-27.

[33] Panagiotis Tsinganos, Bruno Cornelis, Cornelis Jan, Bart Jansen, and Athanassios Skodras. 2021. The Effect of Space-filling Curves on the Efficiency of Hand Gesture Recognition Based on sEMG Signals. *International Journal of Electrical and Computer Engineering Systems* 12, 1 (2021), 23–31.

[34] Haixin Wang, Xiaoyi Fu, Jianliang Xu, and Hua Lu. 2019. Learned Index for Spatial Queries. In *MDM*. 569–574.

[35] Pan Xu, Cuong Nguyen, and Srikanta Tirthapura. 2018. Onion Curve: A Space Filling Curve with Near-Optimal Clustering. In *ICDE*. 1236–1239.

[36] Pan Xu and Srikanta Tirthapura. 2014. Optimality of Clustering Properties of Space-Filling Curves. *ACM Transactions on Database Systems* 39, 2 (2014), 10:1–27.