# Efficient Cost Modeling of Space-filling Curves

Liu, Guanli; Kulik, Lars; Jensen, Christian S.; Li, Tianyi; Qi, Jianzhong

*Publication date:*
2023

# Efficient Cost Modeling of Space-filling Curves

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

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

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

Tianyi Li
Aalborg University
Denmark
tianyi@cs.aau.dk

Jianzhong Qi
The University of Melbourne
Australia
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 the indexing of multi-dimensional data using a one-dimensional index such as a B$^+$-tree. 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 estimation 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.

## KEYWORDS

Space-filling Curves, Cost Model, Reinforcement Learning

**ACM Reference Format:**
Guanli Liu, Lars Kulik, Christian S. Jensen, Tianyi Li, and Jianzhong Qi. 2024. Efficient Cost Modeling of Space-filling Curves. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 13 pages. https://doi.org/XXXXXXX.XXXXXXX

## 1 INTRODUCTION

Indexing is essential to enable efficient query processing on increasingly massive data, including spatial and other low-dimensional data. In this setting, indices based on *space-filling curves* (SFC) are used widely. For example, *Z-order curves* (ZC, see Figures 1a and 1b) [22] are used in Hudi [2], RedShift [1], and SparkSQL [4]; *lexicographic-order curves* (LC, see Figure 1c) are used in PostgreSQL [24] and SQL Server [15]; and *Hilbert curves* (HC) [5] are used in Google S2 [27]. Next, the arguably most important type of query in this setting is the range query that also serves as a foundation for other queries, including *k*NN queries.

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 indexing data using SFC-based indices, the choice of which SFC to use for ordering the data is important.

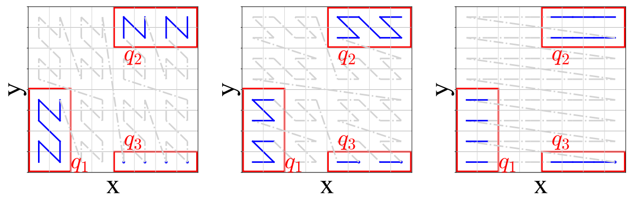


**Figure 1: Examples of SFCs (in grey) and queries (in red).**

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 for a given query (or query workload). This in turn calls for efficient means of estimating the cost of computing a query using a particular SFC (without query execution) to guide SFC selection.

Existing studies [17, 32] 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 query-optimal SFC can be found efficiently. Specifically, we present algorithms that compute the cost of a query in $O(1)$ 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) [7, 19]. A BMC maps multi-dimensional points by merging the bit sequences of the point coordinates (i.e., column indices) from all $d$ dimensions (detailed in Section 3.1). We consider BMCs for two reasons: (1) BMCs generalize ZC and LC used in real systems [2, 4, 15, 24]. 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 $6 \times 10^8$ (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 $\ell$ 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 the cost estimation algorithms by incorporating them into the state-of-the-art learned BMC-based structure, the *BMTree* [13]. The BMTree computes empirical query costs by executing a query workload on the dataset to be indexed. Even with its dataset sampling strategy to reduce the computational costs for query cost estimation, the original SFC learning algorithm of the BMTree takes seven hours (cf. BMTree-SP in Figure 11a) to index a dataset of 100 million points (with only 100,000 sampled points for query cost estimation). Our cost estimation algorithms bring this time down to 57 seconds (cf. BMTree-GC in Figure 11a) with little impact on query efficiency.

Furthermore, we develop an SFC learning algorithm named *LBMC* that uses Reinforcement Learning (RL) techniques to find the optimal BMC. Importantly, the reward calculation in RL leverages our closed-form cost estimation equation and pre-computed look-up tables, thus making the entire learning process extremely efficient. 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 indices on 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 has lower learning and query costs than the competing SFC learning algorithms, including the BMTree.

The rest of the paper is organized as follows. Section 2 covers related work. Section 3 presents preliminaries, and Section 4 details our cost estimations. Section 5 presents LBMC, and Section 6 reports the experimental results. Section 7 concludes the paper.

## 2 RELATED WORK

**Space-filling curves.** SFCs find use in many fields, including in indexing [10, 12, 21, 31], data mining [3], and machine learning [8, 29]. An SFC maps multi-dimensional data values to one-dimensional values, which are then indexed using a one-dimensional index, e.g., the B$^+$-tree.

Two popular SFCs, *ZC* [22] and *HC* [5], are being deployed in practical data systems [1, 2, 4]. Bit-merging curves (BMCs, detailed in Section 3.1) 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 $d$ dimensions. To better order the data points for specific query workloads, *QUILTS* [19] provides a heuristic method to design a series of BMCs and selects the optimal one. A recent technique, the *Bit Merging Tree* (BMTree) [13], learns piece-wise SFCs (i.e., BMCs) by using a quadtree [6]-like strategy to partition the data space and selecting different BMCs for different space partitions.

**Cost estimation for space-filling curves.** To learn an optimal SFC, cost estimation is employed to approximate the query costs without actually computing the queries. Two studies [17, 32] offer theoretical means of estimating the number of curve segments covered by a query range. They do not offer empirical results or guidance on how to construct a query-efficient SFC index.

QUILTS formulates the query cost $C_t$ for a BMC index 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 is able to cover a query range $q$ fully minus the length of the BMC segments in $q$, for each query. 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. However, computing these two costs relies on the accumulated length of the curve segments outside $q$, which is expensive to compute. Given $n$ range queries, it takes $O(n \cdot c_t)$ time to compute $C_t$, where $O(c_t)$ is the average estimation cost per query. Further, they can only be used to estimate the query costs of a given BMC index and do not enable an efficient search for a query-efficient BMC index.

The BMTree estimates query costs using data points sampled from the target dataset. Such cost estimations are expensive for large datasets and many queries. For example, BMTree curve learning over a dataset of 100 million points (with 100,000 sampled points) and 1,000 queries can take more than seven hours (cf. BMTree-SP in Figure 11a). While using a smaller sampled dataset and fewer queries may reduce the learning time, the resulting curve may cause suboptimal query performance (cf. BMTree-SP-6/8/10 in Figure 13). LMSFC [7], another recent proposal, learns a parameterized SFC (which is effectively a BMC) using Bayesian optimization [9]. Like the BMTree, LMSFC uses a sampled dataset and a query workload for query cost estimation and thus has the same issues as the BMTree. Our study aims to address these issues by providing a highly effective and efficient cost estimation.

**Space-filling curve-based indices.** The Hilbert R-tree [10] is a classic index structure based on SFC. It uses an HC to map and order multi-dimensional data, based on which an R-tree is built on the data. This simple structure has been shown to be competitive in many follow-up studies. A recent study further achieves worst-case optimal range query processing by adding an extra *rank space*-based mapping step over the input data before the Hilbert R-tree is built [26]. Another index, the *Instance-Optimal Z-Index* [23], uses a quadtree-like strategy to recursively partition the data space. It creates four sub-spaces of a (sub-)space, which may be of different sizes. The four sub-spaces are each ordered by ZCs of different sizes and follow a 'Σ' or an 'N' shape. At the bottom level of the space partitioning hierarchy, the ZCs of sub-spaces that come from different parent sub-spaces are connected following the order of the ZC that traverses the parent sub-spaces. This way, a curve is formed that traverses all bottom-level sub-spaces, and the data points are indexed in that order.

In the recent wave of machine learning-based optimization for indices [11, 18], SFCs have been used to order and map multi-dimensional data points to one-dimensional values, such that one-dimensional learned indices (e.g., RMI [11]) can be applied. ZM [30] and RSMI [25] are representative proposals. As the BMTree [13] work shows, different learned SFCs can be plugged into these index structures to (possibly) improve their query performance. Our cost estimations can be applied to further enhance the SFC learning process as discussed above, which are orthogonal to these studies.

## 3 PRELIMINARIES

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

### 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 [19].

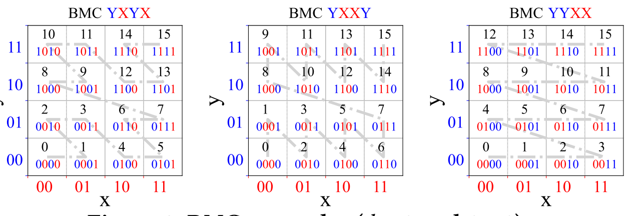


**Figure 2: BMC examples ($d$ = 2 and $\ell$ = 2).**

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 $\sigma$. The coordinates from each dimension have two bits, i.e., the *bit length* $\ell$ of each dimension is 2. The merged bit sequence (i.e., the curve value in binary form) has $d \cdot \ell = 4$ bits.

**Table 1: Frequently used symbols.**

| Symbol | Description |
|---|---|
| $d$ | The data space dimensionality |
| $\ell$ | 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 |
| $\sigma$ | A bit-merging curve (BMC) |
| $\mathcal{F}_\sigma$ | The curve value calculation function over BMC $\sigma$ |
| $\alpha_i^j$ | The $j$th bit value in dimension $i$ |
| $\gamma_i^j$ | The position (0-indexed) of $\alpha_i^j$ in a BMC $\sigma$ |
| $x_i$ | A value in dimension $i$ |
| $[x_{s,i}, x_{e,i}]$ | A value range in dimension $i$ |

The bit length $\ell$ is determined by the grid resolution, which is a system parameter. We use the same $\ell$ for each dimension to simplify the discussion, and we use the little endian bit order, i.e., the rightmost bit has the lowest rank (cf. Figure 3). For simplicity, 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 $\sigma$, we compute the curve value of a point $p = (x_1, x_2, \ldots, x_d)$ using function $\mathcal{F}_\sigma(p)$:

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

Let $x_i$ be the dimension-$i$ coordinate of $p$. In the equation, $\alpha_i^j \in \{0, 1\}$ is the $j$th ($j \in [1, \ell]$) bit of $x_i$, and $\gamma_i^j$ is the rank of $\alpha_i^j$ 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 $\alpha_i^j$ and $\alpha_i^{j+1}$, $\gamma_i^j < \gamma_i^{j+1}$.

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 $\sigma$ = 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 $\sigma$ is zero, which means $\gamma_3^1 = 0$. To calculate the curve value of a point for a given $\sigma$, we derive each $\alpha_i^j$ and $\gamma_i^j$ based on $x_i$ and $\sigma$, respectively.

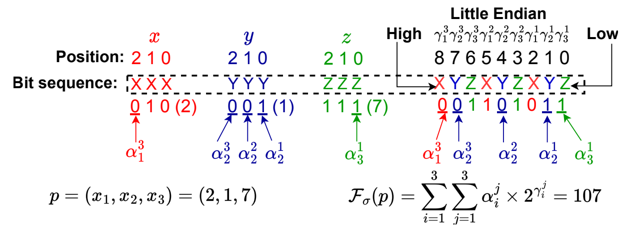


**Figure 3: BMC curve value calculation ($d$ = 3 and $\ell$ = 3).**

**BMC monotonicity.** The BMC value calculation process implies that any BMC is monotonic.

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

Proof. Given $x_{1,i} \le x_{2,i}$, we have $\sum_{j=1}^{\ell} \alpha_{1,i}^j \cdot 2^{j-1} \le \sum_{j=1}^{\ell} \alpha_{2,i}^j \cdot 2^{j-1}$ based on Equation 2. The order among the bits from $x_{1,i}$ and $x_{2,i}$ do not change when they are used to calculate $\mathcal{F}_\sigma(p_1)$ and $\mathcal{F}_\sigma(p_2)$, respectively. Thus, $\sum_{j=1}^{\ell} \alpha_{1,i}^j \cdot 2^{\gamma_{1,i}^j} \le \sum_{j=1}^{\ell} \alpha_{2,i}^j \cdot 2^{\gamma_{2,i}^j}$. Since this holds for any $i \in [1, d]$, we have $\sum_{i=1}^{d} \sum_{j=1}^{\ell} \alpha_{1,i}^j \cdot 2^{\gamma_{1,i}^j} \le \sum_{i=1}^{d} \sum_{j=1}^{\ell} \alpha_{2,i}^j \cdot 2^{\gamma_{2,i}^j}$, i.e., $\mathcal{F}_\sigma(p_1) \le \mathcal{F}_\sigma(p_2)$. □

## 3.2 Range Querying Using a BMC

Next, we present concepts on range query processing with BMCs that will be used later to formulate query cost estimation.

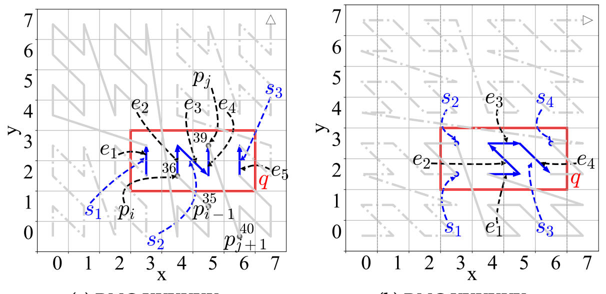
Definition 1 (Range Query). *Given a $d$-dimensional dataset $D$ and a range query $q = [x_{s,1}, x_{e,1}] \times [x_{s,2}, x_{e,2}] \times \ldots \times [x_{s,d}, x_{e,d}]$, where $[x_{s,i}, x_{e,i}]$ denotes the query range in dimension i, query $q$ returns all points $p = (x_1, x_2, ..., x_d) \in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

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 (Theorem 1). 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 [19].** 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 $[\mathcal{F}_\sigma(p_i), \mathcal{F}_\sigma(p_j)]$. Intuitively, each query section translates to a one-dimensional range query $[\mathcal{F}_\sigma(p_i), \mathcal{F}_\sigma(p_j)]$ on a B$^+$-tree index on dataset $D$. Thus, 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 [32].** Query sections are composed by connecting a series of points (cells). The pair of two consecutive points $p_i$ and $p_j$ forms a *directed edge* (denoted by $e$) if the curve values of $p_i$ and $p_j$ differ by one under a given $\sigma$, 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 $K$ ($K \ge 0$) bits are changed from 1 to 0 and the ($K + 1$)st bit is changed from 0 to 1.

Example 2. *We use the binary form of two pairs of integers that form directed edges 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. In this case, four bits starting from the right (i.e., $K = 4$) are changed from 1 to 0, and the fifth bit is changed 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**.*

We explain now why 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 $q$. 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 $\mathcal{E}_\sigma(q) + \mathcal{S}_\sigma(q) = \mathcal{V}(q) - 1 + 1 = \mathcal{V}(q)$. This is because 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 $s_{i+1}$. This leads to the following equation:

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

While $\mathcal{V}(q)$ is independent of $\sigma$, the values for $\mathcal{E}_\sigma(q)$ and $\mathcal{S}_\sigma(q)$ depend on $\sigma$. For example, in Figure 4a ($\sigma$ =XYXYXY), there are 3 query sections ($\mathcal{S}_\sigma(q)$) and 5 directed edges ($\mathcal{E}_\sigma(q)$) in $q$; in Figure 4b ($\sigma$ =YXYXYX), there are 4 query sections and 4 directed edges in $q$. Both figures have $\mathcal{V}(q) = 8$ points in $q$. Equation 3 is key in computing the local cost (Section 4.2) of a query.

# 4 EFFICIENT BMC COST ESTIMATION

Consider a range query $q$ with start point $p_s$ and end point $p_e$ and assume that dataset $D$ has been indexed with a B$^+$-tree using BMC $\sigma$. A simple query algorithm accesses the range $[\mathcal{F}_\sigma(p_s), \mathcal{F}_\sigma(p_e)]$ using the B$^+$-tree, and filters any false positives not included in $q$. The query cost of $q$ then relates to the length of $[\mathcal{F}_\sigma(p_s), \mathcal{F}_\sigma(p_e)]$ and the number of false positives in the range. The number of false positives in turn relates to the number of query sections in $q$. Thus, we define the cost of $q$ (when using BMC $\sigma$), denoted by $C_\sigma(q)$, as a combination of the length of $[\mathcal{F}_\sigma(p_s), \mathcal{F}_\sigma(p_e)]$ (called the *global cost*, $C_\sigma^g(q)$) and the number of query sections (called the *local cost*, $C_\sigma^l(q)$) in $q$. Empirically, we find that the product of the global and the local costs best differentiates the query performance of different BMC indices, which helps identify query-optimal BMC indices (i.e., the goal of our study). Hence, we define $C_\sigma(q)$ as:

$$C_\sigma(q) = C_\sigma^g(q) \cdot C_\sigma^l(q) \tag{4}$$

Note that a commonly used alternative query algorithm is to break $q$ into query sections and perform a range query on the B$^+$-tree for each such section. In this case, the local cost applies directly. The global cost, on the other hand, applies implicitly, because a larger range of $[\mathcal{F}_\sigma(p_s), \mathcal{F}_\sigma(p_e)]$ implies a higher cost to examine and uncover the query sections in the range.

Note also that the cost model of QUILTS [19] uses the product of a global and a local cost. However, its definitions of global and local costs, described in Section 2 are different from ours.

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 $\sigma$ is the length of the curve segment from $p_s$ to $p_e$:*

$$C^g_\sigma(q) = \mathcal{F}_\sigma(p_e) - \mathcal{F}_\sigma(p_s) + 1 = \sum_{j=1}^{d}\sum_{k=1}^{\ell}(\alpha^k_{e,j} - \alpha^k_{s,j}) \cdot 2^{\gamma^k_j} + 1 \tag{5}$$

**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.

$$C^g_\sigma(Q) = \sum_{i=1}^{n} C^g_\sigma(q_i) = \sum_{i=1}^{n}\sum_{j=1}^{d}\sum_{k=1}^{\ell}\underbrace{(\alpha^k_{i,e,j} - \alpha^k_{i,s,j})\cdot}_{\text{BMC independent}} \quad \underbrace{2^{\gamma^k_j}}_{\text{BMC dependent}} \quad +n$$
$$= \sum_{j=1}^{d}\sum_{k=1}^{\ell}\underbrace{\sum_{i=1}^{n}(\alpha^k_{i,e,j} - \alpha^k_{i,s,j})}_{\text{BMC independent}} \cdot 2^{\gamma^k_j} + n = \sum_{j=1}^{d}\sum_{k=1}^{\ell} A^k_j \cdot 2^{\gamma^k_j} + n \tag{6}$$

Here, $q_i \in Q$; $\alpha^k_{i,s,j}$ and $\alpha^k_{i,e,j}$ denote the $k$th 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 $k$th bit of the $j$th dimension, for any BMC. Only the term $2^{\gamma^k_j}$ is BMC dependent and must be calculated for each curve because $\gamma^j_i$ represents the rank of the $j$th bit from dimension $i$ of a BMC (cf. Section 3.1). If the BMC $\sigma$ 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.

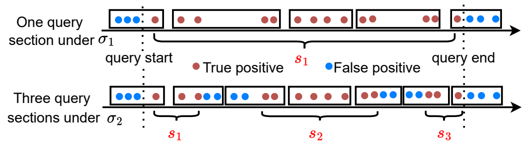
**Algorithm costs.** The above property helps reduce the cost of computing the global cost when given multiple candidate BMCs. For example, when learning the best BMC from a large volume of candidate BMCs (see Section 5), each BMC is evaluated individually in each iteration (Algorithm 3). Without an efficient cost modeling, the global cost is $O(m \cdot n \cdot d \cdot \ell)$ for $m$ candidate BMCs over $n$ queries (based on Equation 5). Based on our proposed closed form method (Equation 6), 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, i.e., $O(m)$ time given constant number of dimensions $d$ and number of bits $\ell$ 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 [17, 32] 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 $\sigma$ 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.


One query section under $\sigma_1$
query start
True positive
$s_1$
False positive
query end
Three query sections under $\sigma_2$
$s_1$
$s_2$
$s_3$
**Figure 5: Query sections vs. block accesses**

Example 3. *In Figure 5, we order points based on BMCs $\sigma_1$ and $\sigma_2$ 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 $\sigma_1$, which leads to a precision of $\frac{14}{5\times4} = 70\%$ for 5 block accesses, whereas $\sigma_2$ 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:

$$\mathcal{S}_\sigma(q) = \mathcal{V}(q) - \mathcal{E}_\sigma(q) \tag{7}$$

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 $e_1 = (5, 6) = [\mathbf{0001}\underline{01}, \mathbf{0001}\underline{10}]$ and $e_2 = (13, 14) = [\mathbf{0011}\underline{01}, \mathbf{0011}\underline{10}]$. 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 $(1, 2) = [\mathbf{0000}\underline{01}, \mathbf{0000}\underline{10}]$ 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 $(31, 32) = [\underline{011111}, \underline{100000}]$ 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 $p_j$, $\mathcal{F}_\sigma(p_i) = \underbrace{\ldots}_{prefix}\,\underline{0}\,\underbrace{1\ldots1}_{K\ 1s}$ and $\mathcal{F}_\sigma(p_j) = \underbrace{\ldots}_{prefix}\,\underline{1}\,\underbrace{0\ldots0}_{K\ 0s}$ $(K \geq 0)$ 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.

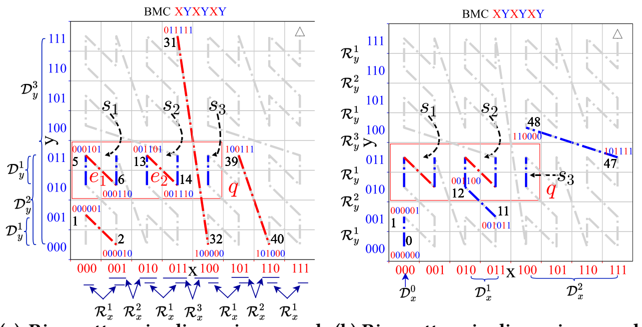
We summarize the bit-changing patterns to form a directed edge with two basic patterns: a *rise pattern* and a *drop pattern*.

Definition 4 (Rise Pattern). *A rise pattern $\mathcal{R}_b^k$ 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 $k$th bit (from the right) is changed from 0 to 1, i.e., $x_{i,b} = \underbrace{\ldots}_{prefix}\,\underline{0}\,\underbrace{1\ldots1}_{(k-1)\ 1s}$ and $x_{j,b} = \underbrace{\ldots}_{prefix}\,\underline{1}\,\underbrace{0\ldots0}_{(k-1)\ 0s}$.*

Definition 5 (Drop Pattern). *A drop pattern $\mathcal{D}_b^k$ 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 $\ell$ bits, there can be $\ell$ 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., $\mathcal{D}_b^0$, indicating no bit value drop in dimension $b$.

Example 4. *In Figure 6a, consider the directed edge from $p_i$ to $p_j$, where $\mathcal{F}_\sigma(p_i) = 1$ $(\underline{000}001)$ and $\mathcal{F}_\sigma(p_j) = 2$ $(\underline{000}010)$, i.e., the '\' segment at the bottom left. The $x$-coordinate of $p_i$ changes from $\underline{000}$ to $\underline{001}$ to that of $p_j$ (i.e., rise pattern $\mathcal{R}_x^1$). The $y$-coordinate of $p_i$ changes from 001 to 000 to that of $p_j$ (i.e., drop pattern $\mathcal{D}_y^1$). Thus, this directed edge can be represented by a combination of $\mathcal{R}_x^1$ and $\mathcal{D}_y^1$, denoted as $\mathcal{R}_x^1 \oplus \mathcal{D}_y^1$. 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., $\mathcal{R}_x^3 \oplus \mathcal{D}_y^3$ for the one from $\mathcal{F}_\sigma(p_i) = 31$ to $\mathcal{F}_\sigma(p_j) = 32$, and $\mathcal{R}_x^2 \oplus \mathcal{D}_y^2$ for the one from $\mathcal{F}_\sigma(p_i) = 39$ to $\mathcal{F}_\sigma(p_j) = 40$.*


BMC XYXYXY
(a) panel: 111, 110, 101, 100, 011, 010, 001, 000
$\mathcal{D}_y^3$, $\mathcal{D}_y^1$, $\mathcal{D}_y^2$, $\mathcal{D}_y^1$
$s_1$ $s_2$ $s_3$
011111, 31, 000101, 5, 6, 000110, 001101, 13, 14, 001110, 100111, 39, 000001, 1, 2, 000010, 32, 100000, 40, 101000
$e_1$ $e_2$ $q$
x, y
$\mathcal{R}_x^1$ $\mathcal{R}_x^2$ $\mathcal{R}_x^1$ $\mathcal{R}_x^3$ $\mathcal{R}_x^1$ $\mathcal{R}_x^2$ $\mathcal{R}_x^1$
(b) panel: $\mathcal{R}_y^1$ $\mathcal{R}_y^2$ $\mathcal{R}_y^1$ $\mathcal{R}_y^3$ $\mathcal{R}_y^1$ $\mathcal{R}_y^2$ $\mathcal{R}_y^1$
48, 110000, 101111, 47, 001100, 12, 11, 001011, 000001, 1, 0, 000000
$s_1$ $s_2$ $s_3$ $q$
$\mathcal{D}_x^0$ $\mathcal{D}_x^1$ $\mathcal{D}_x^2$

**(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.**

*Figure 6a has shown the rise patterns $\mathcal{R}_x^k$ in dimension-$x$ and the drop patterns $\mathcal{D}_y^k$ in dimension-$y$. Combining a rise and a drop pattern from these patterns forms a directed edge in red in the figure.*

*Similarly, we show in Figure 6b the rise patterns $\mathcal{R}_y^k$ in dimension-$y$ and the drop patterns $\mathcal{D}_x^k$ in dimension-$x$. Combining a rise and a drop pattern from these patterns forms a black directed edge.*

The *pattern combination operator* '$\oplus$' applied on two (rise or drop) patterns means that the $(K+1)$-bit change of 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 $\sigma = X^3Y^3X^2Y^2X^1Y^1$ (i.e., XYXYXY). From the right to the left of $\sigma$, the first rise pattern is $\mathcal{R}_x^1$. It can only be combined with drop pattern $\mathcal{D}_y^1$, as there is just one bit $Y^1$ from dimension-$y$ to the right of $X^1$. Similarly, $\mathcal{R}_x^2$ and $\mathcal{R}_x^3$ can each be combined with $\mathcal{D}_y^2$ and $\mathcal{D}_y^3$, 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 $\ell$ 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 $d = 2$:

$$\mathcal{E}_\sigma(q) = \sum_{i=1}^{\ell}\Big(\mathcal{N}(\mathcal{R}_x^i)\cdot\mathcal{N}(\mathcal{D}_y^{r_y}) + \mathcal{N}(\mathcal{R}_y^i)\cdot\mathcal{N}(\mathcal{D}_x^{r_x})\Big) \tag{8}$$

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 $\mathcal{R}_x^i$ ($\mathcal{R}_y^i$). In Figure 6, for $q = ([0, 4] \times [2, 3])$, there are two $\mathcal{R}_x^1$, one $\mathcal{R}_x^2$, and one $\mathcal{R}_x^3$, i.e., $\mathcal{N}(\mathcal{R}_x^1) = 2$, $\mathcal{N}(\mathcal{R}_x^2) = 1$, and $\mathcal{N}(\mathcal{R}_x^3) = 1$. Next, there is one $\mathcal{D}_y^1$, zero $\mathcal{D}_y^2$, and zero $\mathcal{D}_y^3$ that are valid to match with these rise patterns, i.e., $\mathcal{N}(\mathcal{D}_y^1) = 1$, $\mathcal{N}(\mathcal{D}_y^2) = 0$, and $\mathcal{N}(\mathcal{D}_y^3) = 0$. Similarly, $\mathcal{N}(\mathcal{R}_y^1) = 1$, and $\mathcal{R}_y^1$ can be matched with $\mathcal{D}_x^0$ , where $\mathcal{N}(\mathcal{D}_x^0) = 5$. Recall that $\mathcal{D}_x^0$ 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 7, which is consistent with the figure.

**Efficient counting of rise and drop patterns.** A rise pattern $\mathcal{R}_b^k$ 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., $\underline{0}\ \underbrace{1...1}_{(k-1)\ \text{1s}}$) and $2^{k-1}$ (i.e., $\underline{1}\ \underbrace{0...0}_{(k-1)\ \text{0s}}$) 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 $\mathcal{D}_b^k$ 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, $a \cdot 2^k$ is the prefix, while $2^k - 1$ (i.e., $\underbrace{1...1}_{k\ \text{1s}}$) and 0 (i.e., $\underbrace{0...0}_{k\ \text{0s}}$) 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.

**Generalizing to $d$ dimensions.** As mentioned at the beginning of the subsection, a directed edge can be decomposed into a rise pattern in one dimension and drop patterns in the remaining $d - 1$ dimensions. We call the set of all drop patterns in the $d - 1$ dimensions a *drop pattern collection.*

Definition 6 (Drop Pattern Collection). *For a directed edge in $d$-dimensional space, a drop pattern collection $\mathcal{D}^{k'}$ represents the bit combination over $d - 1$ drop patterns: $\mathcal{D}^{\sum_{i=1, i \neq b}^{d-1} k_i} = \biguplus_{i=1, i \neq b}^{d} \mathcal{D}_i^{k_i}$ ($k' = \sum_{i=1, i \neq b}^{d} k_i = K - k$), 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 $\mathcal{D}_b^k$ 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., $\mathcal{R}_b^k \oplus \mathcal{D}^{\sum_{i=1, i \neq b}^{d} k_i} = \mathcal{R}_b^k \oplus (\biguplus_{i=1, i \neq b}^{d} \mathcal{D}_i^{k_i})$ where $k' = \sum_{i=1, i \neq b}^{d} k_i$. Equation 8 is then rewritten as:

$$\mathcal{E}_\sigma(q) = \sum_{j=1}^{d} \sum_{i=1}^{\ell} \mathcal{N}(\mathcal{R}_j^i) \cdot \mathcal{N}(\mathcal{D}^r) \tag{9}$$

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

### 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:

$$C_\sigma^l(Q) = \sum_{i=1}^{n} C_\sigma^l(q_i) = \sum_{i=1}^{n} \mathcal{V}(q_i) - \sum_{i=1}^{n} \mathcal{E}_\sigma(q_i) \tag{10}$$

This cost takes $O(n)$ time to compute. Given $m$ BMCs, computing their respective total local costs $C_\sigma^l(Q)$ 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. Since each directed edge is a combination of a rise pattern in some dimension $b$ and $d - 1$ drop patterns, we proceed to show how to pre-compute $d$ pattern tables, each recording the rise patterns of a dimension.

**Table 2: Pattern table *Table$^b$* for dimension $b$ using $\ell$ bits on each dimension.**

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

Definition 7 (Pattern Table). *The pattern table for dimension $b$, denoted by Table$^b$, contains $\ell$ 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. As shown in Table 2, the value in row $i$ and column $j$ is the product of the numbers of rise pattern $\mathcal{R}_b^i$ and drop pattern $\mathcal{D}^j$.*

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 $\ell$
Output: Pattern table $Table^b$
1 Initialize an $\ell \times (\ell \cdot (d-1) + 1)$ table $Table^b$;
2 for $q \in Q$ do
3     for $i \in [1, \ell]$ do
4         for $j \in [0, \ell \cdot (d-1)]$ do
5             $\mathcal{N}(\mathcal{R}_b^i) \leftarrow$ count the number of $\mathcal{R}_b^i$ in $q$;
6             $\mathcal{N}(\mathcal{D}^j) \leftarrow$ count the number of $\mathcal{D}^j$ in $q$;
7             $Table^b[i][j] \leftarrow Table^b[i][j] + \mathcal{N}(\mathcal{R}_b^i) \cdot \mathcal{N}(\mathcal{D}^j)$;
8 return $Table^b$;
```

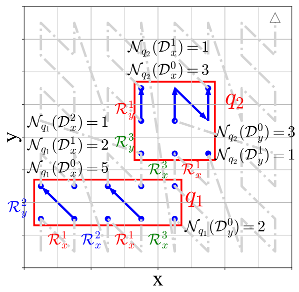
Example 5. *In Figure 7a, we show two queries $q_1$ and $q_2$, and the pattern tables Table$^x$ and Table$^y$ are shown in Tables 3 and 4, respectively. In the tables, we use '+' to denote summing up the pattern table cell values (i.e., $\mathcal{N}(\mathcal{R}_b^i) \cdot \mathcal{N}(\mathcal{D}^j)$, 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., $4 + 3$.*

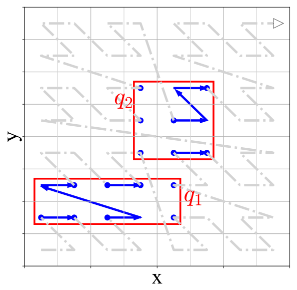
**Table 3:** *Table$^x$*

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

**Table 4:** *Table$^y$*

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


$\mathcal{N}_{q_2}(\mathcal{D}_x^1) = 1$
$\mathcal{N}_{q_2}(\mathcal{D}_x^0) = 3$
$\mathcal{R}_y^1$
$q_2$
$\mathcal{N}_{q_1}(\mathcal{D}_x^2) = 1$
$\mathcal{N}_{q_1}(\mathcal{D}_x^1) = 2$
$\mathcal{N}_{q_1}(\mathcal{D}_x^0) = 5$
$\mathcal{R}_y^3$
$\mathcal{N}_{q_2}(\mathcal{D}_y^0) = 3$
$\mathcal{N}_{q_2}(\mathcal{D}_y^1) = 1$
$\mathcal{R}_x^3$
$\mathcal{R}_x^1$
$\mathcal{R}_y^2$
$q_1$
$\mathcal{N}_{q_1}(\mathcal{D}_y^0) = 2$
$\mathcal{R}_x^1$ $\mathcal{R}_x^2$ $\mathcal{R}_x^1$ $\mathcal{R}_x^3$
x
y
(a) Six directed edges ($\sigma$ = XYXYXY)



(b) Nine directed edges ($\sigma$ = YXYXYX)

**Figure 7: Rise and drop pattern counting example ($d = 2$, $\ell = 3$). The results are shown in pattern tables in Tables 3 and 4.**

### 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 $\sigma$, 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 $\ell$ rise patterns in each dimension are considered because a BMC has $\ell$ 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 $\sigma$, data dimensionality $d$, number of bits per
       dimension $\ell$, all pattern tables $Table^j$, total number of cells
       in the queries $\mathcal{V}$
Output: Total local cost of $n$ queries
1 $\mathcal{E}_\sigma \leftarrow 0$;
2 for $j \in [1, d]$ do
3     for $i \in [1, \ell]$ do
4         $col \leftarrow$ get_col($\sigma, i, j$);
5         $\mathcal{E}_\sigma \leftarrow \mathcal{E}_\sigma + Table^j[i][col]$;
6 return $\mathcal{V} - \mathcal{E}_\sigma$;
```

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 Definition 6. 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, $\ell$ 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 $\ell$ 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).

## 5 COST ESTIMATION-BASED BMC LEARNING

Next, powered by our efficient cost estimations, we aim to find the optimal BMC $\sigma_{opt}$ 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 $\sigma$, the process to search for $\sigma_{opt}$ 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 on the bit swaps: (a) we only swap two adjacent bits each time, and (b) two bits from the same dimension cannot be swapped (which guarantees valid BMCs after swaps, cf. Section 3.1). Any bit then takes at most $(d - 1) \cdot \ell$ swaps to reach its optimal position, when such a 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 [14] 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 $i$th bit. For example, if $\sigma_t$ =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 $a$th 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 $r\colon \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow r$, which is the query cost reduction when reaching a new BMC $\sigma_{t+1}$ from $\sigma_t$. Since an oracle is unavailable, we use our cost model to estimate the query cost of a BMC. 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 $\sigma_t$ estimated by Equation 6 and Algorithm 2. (5) Parameter $\epsilon$, which balances exploration and exploitation to avoid local optima.

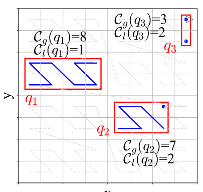
Based on this formulation, we use *deep Q-learning* [16] in our LBMC algorithm to learn a query-efficient BMC index.

**The LBMC algorithm.** We summarize LBMC in Algorithm 3 where the input $\sigma_1$ can be any initial BMC, e.g., a ZC. The key idea of LBMC is to learn a policy $\pi : \mathcal{S} \rightarrow \mathcal{A}$ that guides the position selection for a bit swap given a status, to maximize a value function $\mathrm{Q}^*(\phi(\sigma_t), a)$ (i.e., the reward) at each step $t$. Such a policy $\pi$ can be learned by training a model (a *deep Q-network*, DQN) with parameters $\theta$ 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., $\max_a \mathrm{Q}^*(\phi(\sigma_t), a; \theta)$). After a number of iterations, the learned BMC $\sigma^*_{opt}$ is expected to approach $\sigma_{opt}$, which is returned as the algorithm output.

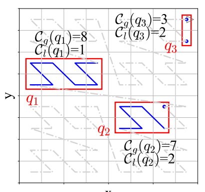
We initialize a storage $MQ$ to store the latest $N_{MQ}$ bit-swapping records (i.e., the experience, Line 1). We learn to approach $\sigma_{opt}$ with $M$ episodes and $T$ steps per episode (Lines 2 and 3). In each episode, we start with $\sigma_1$ encoded by $\phi(\cdot)$. To select a swap position $a_t$ at step $t$, we generate a random number in $[0, 1]$, if the number is greater than $\epsilon$, we randomly select a position $a_t$, otherwise, we set $a_t$ as the position with the highest probability to obtain a maximal reward, i.e., $\max_a \mathsf{Q}^*(\phi(\sigma_t), a; \theta)$ (Line 4). The prediction is based on the current state $\sigma_t$ and model weights $\theta$. We execute $a_t$ ($\mathsf{E}(\sigma_t, a_t)$ at Line 5) and compute reward $r_t$ using our cost model (Line 6). We record the new transition in $MQ$ and train the DQN (i.e., update $\theta$) over sampled data in $MQ$ (Lines 7 and 8). 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 [16]. We use $\sigma^*_{opt}$ to record the new BMC from each swap (Line 9), which is returned in the end (Line 10).

**Algorithm 3:** Learn BMC (LBMC)

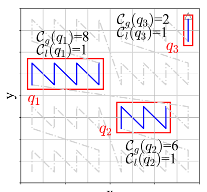
```
   Input: Initial BMC $\sigma_1$
   Output: A query-efficient BMC $\sigma^*_{opt}$
1  Initialize replay memory $MQ$ with capacity $N_{MQ}$;
2  for $episode \in [1, M]$ do
3      for $t \in [1, T]$ do
4          With probability $\epsilon$ select a random position $a_t$, or
             $a_t \leftarrow \max_a \mathsf{Q}^*(\phi(\sigma_t), a; \theta)$;
5          $\sigma_{t+1} \leftarrow \mathsf{E}(\sigma_t, a_t)$;
6          Compute reward $r_t$;
7          Store transition $(\phi(\sigma_t), a_t, r_t, \phi(\sigma_{t+1}))$ in $MQ$;
8          Train model $\theta$ with sampled transitions from $MQ$;
9          $\sigma^*_{opt} \leftarrow \sigma_{t+1}$;
10 return $\sigma^*_{opt}$;
```



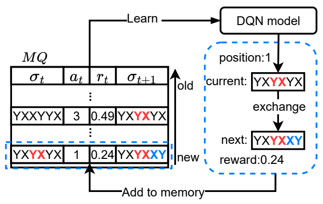
**(a) YXXYYX, $C_1$ = 175**



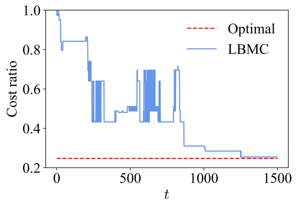
**(b) YXYXYX, $C_2$ = 90**



**(c) YXYXXY, $C_3$ = 48**



**(d) Learning through LBMC**



**(e) Cost ratio vs. number of 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 $\sigma_1$ = YXXYYX has an (estimated) query cost of $C_1$ = 175 (Figure 8a). We select position $a_1$ = 3 and swap the 3rd and the 4th bits to get $\sigma_2$ = YXYXYX such that the cost is decreased to $C_2$ = 90 (Figure 8b). Next, we select position $a_2$ = 1 and swap the 1st and the 2nd bits to get $\sigma_3$ = YXYXXY with cost $C_3$ = 48 (Figure 8c). We store all the intermediate results into memory MQ for learning the DQN model in Figure 8d, where we show the BMCs without encoding. Figure 8e shows the cost ratios, i.e., $C_t/C_1$, which decrease as t increases (Figures 8a to 8c are three of the steps). The learned BMC approaches the optimum in this process.*

**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 aim to evaluate the (1) efficiency and (2) effectiveness of the proposed cost estimation algorithms, as well as (3) LBMC vs. other SFCs, including the learning-based ones.

### 6.1 Experimental Settings

Our cost estimation algorithms (i.e., GC and LC) and BMC learning algorithm (i.e., LBMC) are implemented in Python (available at https://anonymous.4open.science/r/LearnSFC-B6D8). The learning of BMC is supported by TensorFlow. We run experiments on a desktop computer running 64-bit Ubuntu 20.04 with a 3.60 GHz Intel i9 CPU, 64 GB RAM, and a 500 GB SSD.

**Datasets.** We use two real datasets: **OSM** [20] and **NYC** [28]. OSM contains 100 million 2-dimensional location points (2.2 GB). NYC contains some 150 million yellow taxi transactions (8.4 GB). After cleansing incomplete records, we retain the pick-up locations (2-dimensional points) of 100 million records. Additionally, we follow the study of the state-of-the-art competitor, the BMTree [13], and use two synthetic datasets, each with 100 million points: **UNI** and **SKEW**, which follow uniform and skewed distributions.

**Queries.** We again follow the BMTree study and generate synthetic query workloads. Specifically, 1,000 synthetic queries are used for SFC learning, while 2,000 queries are generated separately for testing. The queries are of uniform size and follow the distributions of their respective datasets. To assess our cost estimation algorithms (Sections 6.2 and 6.3), we employ square queries, since the query shape does not impact the cost estimation time.

**Evaluation metrics.** The core evaluation metrics used are (1) the **cost estimation time**, (2) the **average number of block accesses per query** when using different SFC ordering for query processing (in PostgreSQL), and (3) the **SFC learning time**.

**Parameter settings.** Table 5 summarises the parameter values used, with default values in **bold**. In the table, $n$ denotes the number of queries; $\delta$ denotes the edge length of a query; $d$ denotes the data dimensionality; and $N$ denotes the dataset cardinality. We randomly sample from the datasets described above to obtain datasets of different cardinalities.

For SFCs, a key parameter is the number of bits $\ell$, which impacts the curve value mapping efficiency substantially. To evaluate the cost estimation efficiency, we restrict $\ell$ to 18, beyond which a naive local cost baseline becomes computationally infeasible. In later experiments, we set $\ell$ = 20 following the BMTree to balance the computational costs of curve value mapping and cost estimation.

The BMTree has two additional parameters: the dataset sampling rate $\rho$ to form a subset for query cost estimation, and the depth $h$ of space partitioning.

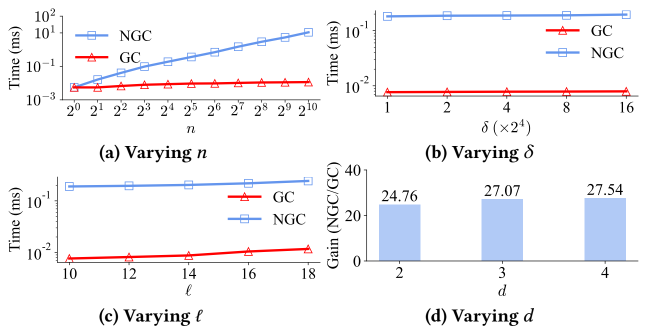
**Table 5: Parameter settings.**

| Experiments | Parameter | Values |
|---|---|---|
| Cost estimation efficiency | $n$ | $2^0, 2^1, 2^2, 2^3, \mathbf{2^4}, 2^5, 2^6, 2^7, 2^8, 2^9, 2^{10}$ |
| | $\delta(\times 2^4)$ | **1**, 2, 4, 8, 16 |
| | $\ell$ | **10**, 12, 14, 16, 18 |
| | $d$ | **2**, 3, 4 |
| Cost estimation effectiveness | $N$ | $10^4, 10^5, 10^6, \mathbf{10^7}, 10^8$ |
| | $\rho(\times 10^{-3})$ | 0.1, 0.25, 0.5, 0.75, **1**, 2.5, 5, 7.5, 10 |
| | $h$ | 5, 6, 7, 8, 9, **10** |
| | $n$ | 100, 500, 1000, 1500, **2000** |
| | datasets | **OSM**, SKEW |
| Query efficiency | $N$ | $10^4, 10^5, 10^6, \mathbf{10^7}, 10^8$ |
| | aspect ratio | 16 : 1, 4 : 1, 1 : 1, 1 : 4, **1 : 16** |
| | $\delta(\times 2^6)$ | 1, 2, **4**, 8, 16 |
| | datasets | **OSM**, NYC, UNI, SKEW, |

## 6.2 Cost Estimation Efficiency

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 6 and 9. 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 5 and 10, denoted by **NGC** and **NLC**.

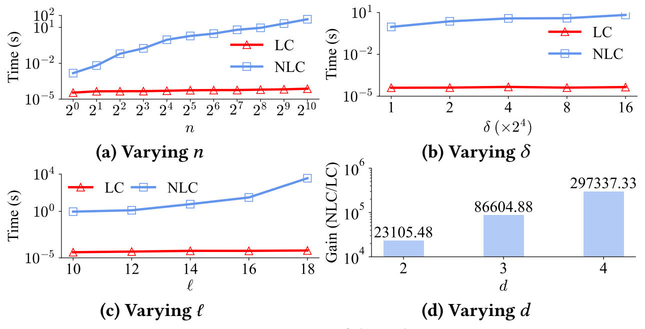
We vary the number of queries $n$, the query size (via $\delta$), and the number of bits $\ell$. We run experiments for 2- to 4-dimensional spaces. Due to page limits, we focus on the 2-dimensional space (the algorithms' comparative results are similar for $d \in \{3, 4\}$). As the cost estimation is data independent, a dataset is not needed to study their efficiency. The queries are generated at random locations.



(a) Varying $n$ (b) Varying $\delta$ (c) Varying $\ell$ (d) Varying $d$

**Figure 9: Running times of global cost estimation.**

*6.2.1 Efficiency of GC.* Figures 9a and 9b show the impact of $n$ and $\delta$, respectively. Since GC takes $O(d \cdot \ell)$ time to compute (after the initialization step), its running time is unaffected by $n$ and $\delta$. NGC takes $O(n \cdot d \cdot \ell)$ time. Its running time grows linearly with $n$ and is unaffected by $\delta$ as shown in the figures. Figure 9c shows that the running times of GC and NGC both increase with $\ell$, which is consistent with their time complexities. Since the relative performance of our algorithm and the baseline is stable when $\ell$ 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 $d$. Here, we show the performance gain (i.e., the running time of NGC over that of GC) instead of the absolute running times, which are of different scales when $d$ is varied such that it is difficult to observe the relative performance. We see that GC is faster than NGC by 24x. Overall, GC is consistently faster than NGC, with up to more than an order of magnitude performance gain, which confirms the high efficiency of GC.



(a) Varying $n$ (b) Varying $\delta$ (c) Varying $\ell$ (d) Varying $d$

**Figure 10: Running times of local cost estimation.**

*6.2.2 Efficiency of LC.* Figures 10a to 10d show 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 five orders of magnitude when $d = 4$.

*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 $\delta$, $\ell$, 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: Initialization costs 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 |

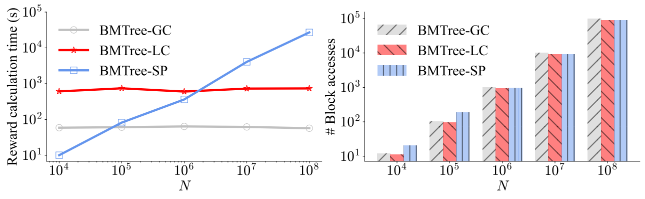
## 6.3 Effectiveness of Cost Estimation

We next explore the applicability and effectiveness of our GC and LC cost estimations by using them to replace the built-in cost estimations of the state-of-the-art SFC learning algorithm, the BMTree. We denote the resulting variants by **BMTree-GC** and **BMTree-LC**. The original BMTree uses a data sampling-based empirical cost estimation method. We denote it as **BMTree-SP**.

We report the time cost of reward calculation for the three variants, as the other steps of the variants are the same. After the SFCs are learned by the three variants, we build a B$^+$-tree with each SFC in PostgreSQL to index the input dataset. We measure the average number of block accesses as reported by PostgreSQL to process each of the queries as described earlier.

*6.3.1 Varying the Dataset Cardinality.* We start by varying the dataset cardinality $N$ from $10^4$ to $10^8$. Figure 11 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, the indices built using all three algorithms require 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 index building costs may use BMTree-GC.



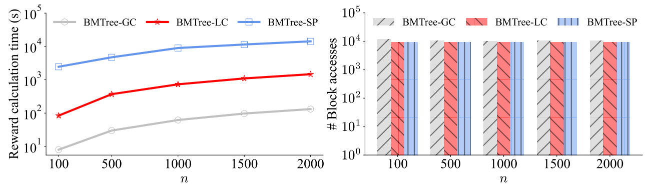
(a) Reward calculation time (b) Query processing cost

**Figure 11: Varying the dataset cardinality (OSM).**

*6.3.2 Varying the Number of Queries.* Next, we vary the number of queries used in curve learning, $n$, from 100 to 2,000. We see that BMTree-LC and BMTree-GC consistently outperform BMTree-SP by one and two orders of magnitude in terms of the reward calculation time, respectively (Figure 12a). We note that, now the computation times of BMTree-LC and BMTree-GC vary with $n$, which differs from what was reported in Figures 9a and 10a. This happens because the BMTree uses different BMCs in different subspaces to accommodate different data and query patterns. As there are more queries, more different patterns may need to be considered, resulting in more different BMCs, each of which requires a different GC and LC cost estimation. Thus, the cost estimation costs grow with the number of queries $n$.

Meanwhile, the query costs of the three algorithms are again close, e.g., 9,199, 9,248, and 10,462, for BMTree-LC, BMTree-SP, and BMTree-GC, respectively, when $n$ is 1,500. The higher query cost of BMTree-GC shows that while GC is extremely simple and efficient, it may not find the most query-efficient curves, which underlines the importance of the LC cost estimation algorithm.

We further observe a slight drop in the number of block accesses as $n$ increases. Intuitively, using more queries for curve learning can lead to curves that better suit the query workload.



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

**Figure 12: Varying the number of queries (OSM).**

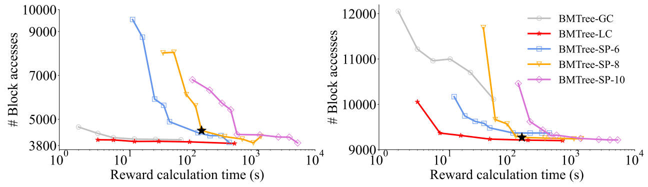
*6.3.3 Varying the Sampling Rate and the Depth of the BMTree.* Two alternative approaches to improve the curve learning efficiency of the BMTree are (1) to reduce its data sampling rate $\rho$ and (2) to reduce the depth of its space partitioning $h$.

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 $\rho$ 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 $\rho$ values (points on the right come from larger $\rho$ values).

BMTree-GL and BMTree-LC are plotted with one polyline each, as they are not impacted by $\rho$. 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.**

## 6.4 Query Efficiency with BMC Learning

We proceed to study the BMC learning efficiency of **LBMC** and the query efficiency of the indices built using the learned BMCs.

**Competitors.** We compare with five different SFC-based ordering techniques. (1) **QUILTS** [19] orders data points by a BMC derived by a curve design method as described in Section 2. We implement it according to its paper as the source code is unavailable. (2) **ZC** [21] orders data points by their Z-curve values. (3) **HC** [10] orders data points by their Hilbert curve values. (4) **LC**, which is also called the C-Curve, orders data points lexicographically by their dimension values [13, 19]. (5) **BMTree** [13] orders data points by multiple BMCs in different sub-spaces. We use its released code (with $h = 8$ and $\rho = 0.001$ to balance the reward calculation time and the query costs, cf. the '★'-points on BMTree-SP-8 in Figure 13). We cannot compare with the recent learned SFC, LMSFC [7], because its source code and some implementation details are unavailable. We do not compare with RSMI [25] as it has been shown to be outperformed by the BMTree [13].

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.

*6.4.1 Overall Results.* Figure 14 shows the average number of block accesses on all four datasets. LBMC outperforms all competitors consistently. On SKEW, the advantage of LBMC over the BMTree is the most pronounced. It reduces the average number of block accesses by 28x (111 vs. 3,084) and by 6x (111 vs. 674) in comparison with the BMTree and QUILTS, respectively. On NYC, the advantage of LBMC over the BMTree is the least, yet it still requires only 2,638 block accesses which is fewer than that of the BMTree at 3,448. These results suggest that LBMC is highly efficient at reducing the query costs across diverse datasets.

LC is the worst, which is expected as LC curves fail to preserve the data locality. The BMTree and QUILTS outperform LC, ZC, and HC on real data such as NYC, where they benefit more from the query based optimizations. However, there are no consistent results across the different datasets. We conjecture that fine-tuning of the parameter values of $h$ and $\rho$ may be needed for the BMTree over each different dataset. Such fine-tuning is not required by LBMC.

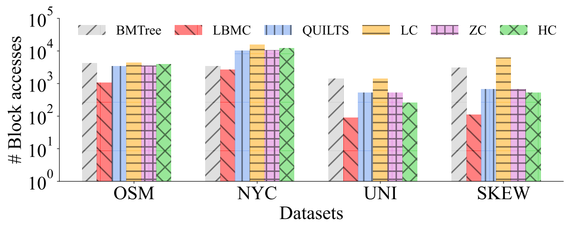


**Figure 14: Block access over all datasets.**

*6.4.2 Varying the Dataset Cardinality.* We further study the impact of dataset cardinality $N$. Figure 15 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 $N = 10^7$).

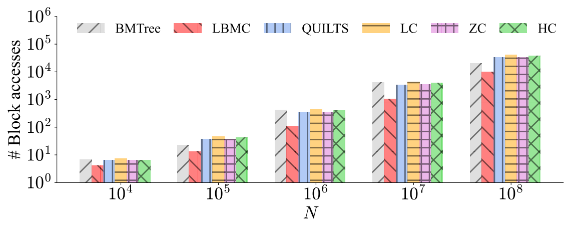


**Figure 15: Varying cardinality (OSM).**

We report the SFC learning times of the BMTree and LBMC when varying $N$ in Table 7. 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 $N$.

LC, 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 |

*6.4.3 Varying the Aspect Ratio of Queries.* Figure 16 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 LC 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 $\ell$ bits ($\ell = 20$ by default).

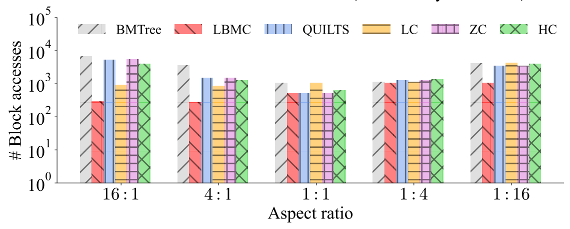


**Figure 16: Varying the query aspect ratio (OSM).**

*6.4.4 Varying the Edge Length of Queries.* Figure 17 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.

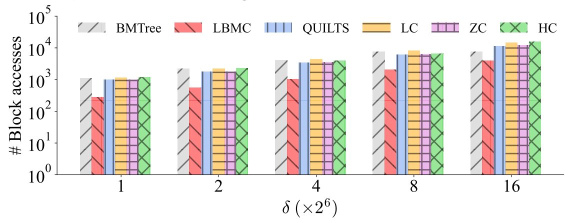


**Figure 17: Varying the query edge length (OSM).**

## 7 CONCLUSIONS AND FUTURE WORK

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 and after an $O(n)$-time initialization. We extended these algorithms to the state-of-the-art curve learning algorithm, the BMTree, which originally measured the effectiveness of SFCs by querying the data points to be indexed. Experimental results show that the proposed algorithms are capable of reducing 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 result learned BMCs are shown to achieve lower query costs than those of the BMTree and other baselines under nearly all settings tested.

In future work, it is of interest to design cost estimation algorithms for non-BMCs, e.g., HC, and use learning-based techniques to build more efficient multi-dimensional indices.

## 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: 2023-10-10.

[2] Apache Hudi. 2021. *https://hudi.apache.org/blog/2021/12/29/hudi-zorder-and-hilbert-space-filling-curves.* Accessed: 2023-10-10.

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

[4] Databricks Engineering Blog. 2018. *https://databricks.com/blog/2018/07/31/processing-petabytes-of-data-in-seconds-with-databricks-delta.html.* Accessed: 2023-10-10.

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

[6] 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.

[7] 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.

[8] 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).

[9] 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.

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

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

[12] 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.

[13] 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.

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

[15] Microsoft. 2023. *https://learn.microsoft.com/en-us/sql/relational-databases/indexes/indexes?view=sql-server-ver16.* Accessed: 2023-10-10.

[16] 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).

[17] 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.

[18] Vikram Nathan, Jialin Ding, Mohammad Alizadeh, and Tim Kraska. 2020. Learning Multi-Dimensional Indexes. In *SIGMOD*. 985–1000.

[19] 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.

[20] OpenStreetMap. 2018. *OpenStreetMap North America data dump. https://download.geofabrik.de.* Accessed: 2023-10-10.

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

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

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

[24] PostgreSQL. 2023. *https://www.postgresql.org/docs/current/indexes-multicolumn.html.* Accessed: 2023-10-10.

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

[26] 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.

[27] S2 Geometry. 2023. *http://s2geometry.io.* Accessed: 2023-10-10.

[28] TLC Trip Record Data. 2022. *https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page.* Accessed: 2023-10-10.

[29] 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.

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

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

[32] 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.