Optimizing Block Skipping for High-Dimensional Data with Learned Adaptive Curve

XU CHEN[*], University of Electronic Science and Technology of China, China
SHUNCHENG LIU, Huawei Technologies Co., Ltd., China
TONG YUAN, Huawei Technologies Co., Ltd., China
TAO YE, Huawei Technologies Co., Ltd., China
KAI ZENG[†], Huawei Technologies Co., Ltd., China
HAN SU[†], University of Electronic Science and Technology of China, China
KAI ZHENG[†‡], University of Electronic Science and Technology of China, China

In the realm of big data and cloud analytics, efficiently managing and retrieving high-dimensional data presents a critical challenge. Traditional indexes often struggle with the storage overhead inherent in large datasets. There is a growing interest in the adoption of Small Materialize Aggregation (SMA) among cloud database vendors due to its ability to maintain lightweight block-level metadata, facilitating efficient block skipping. However, SMA performance relies heavily on data layout. This is especially critical in scenarios with wide tables containing hundreds of dimensions, where the curse of dimensionality exacerbates the issue. In this paper, we propose *AdaCurve*, a novel approach aimed at enhancing block skipping in high-dimensional datasets through adaptive optimization of data layout. Unlike conventional static and non-adaptive space-filling curves (SFCs), *AdaCurve* leverages machine learning to develop an adaptive curve—a dynamically adjusting optimal projection function tailored to high-dimensional workloads and data characteristics. We introduce an attention-based network to handle high-dimensional data and a learnable objective for training adaptive curves in an end-to-end manner. Extensive experiments conducted on the Spark with real-world datasets demonstrate the effectiveness of *AdaCurve*. We have shown that *AdaCurve* effectively scales to datasets with dimensions of up to 1,000 columns, achieving a 2.8× improvement in block skipping compared to SFCs.

CCS Concepts: • **Information systems → Record and block layout**.

Additional Key Words and Phrases: data layout, high dimension, machine learning

[*]The work was done during internship at Huawei Cloud GaussDB(DWS) Team.
[†]Corresponding authors
[‡]Corresponding author Kai Zheng is with School of Computer Science and Engineering, UESTC. He is also with Shenzhen Institute for Advanced Study, UESTC.

Authors' Contact Information: Xu Chen, University of Electronic Science and Technology of China, China, xuchen@std.uestc.edu.cn; Shuncheng Liu, Huawei Technologies Co., Ltd., China, liushuncheng1@huawei.com; Tong Yuan, Huawei Technologies Co., Ltd., China, yuantong@huawei.com; Tao Ye, Huawei Technologies Co., Ltd., China, yetao1@huawei.com; Kai Zeng, Huawei Technologies Co., Ltd., China, kai.zeng@huawei.com; Han Su, University of Electronic Science and Technology of China, China, hansu@uestc.edu.cn; Kai Zheng, University of Electronic Science and Technology of China, China, zhengkai@uestc.edu.cn.

ACM 2836-6573/2025/2-ART60
https://doi.org/10.1145/3709710

ACM Reference Format:
Xu Chen, Shuncheng Liu, Tong Yuan, Tao Ye, Kai Zeng, Han Su, and Kai Zheng. 2025. Optimizing Block Skipping for High-Dimensional Data with Learned Adaptive Curve. *Proc. ACM Manag. Data* 3, 1 (SIGMOD), Article 60 (February 2025), 26 pages. https://doi.org/10.1145/3709710

1 Introduction

In the big data era, data movement within cloud environments is essential for analytics. Leading cloud database providers like Databricks and Amazon Redshift have adopted columnar storage, which organizes data vertically into *blocks* that enhance compression and efficiency by allowing the system to skip irrelevant columns. These blocks form the smallest units for I/O operations. However, with the separation of computing and storage in modern cloud setups, where data is stored remotely, the cost of I/O operations has significantly increased. To reduce the I/O cost, traditional indexing methods like B-trees, R-trees, and KD-trees, once popular in early analytics systems, now struggle to meet the demands of large-scale cloud environments due to their substantial storage overhead and maintenance costs [62].

Small Materialized Aggregations (SMA) [39] are increasingly popular as effective, lightweight indexing methods in modern data management, particularly in analytics systems using columnar storage formats like Parquet and ORC. SMA stores minimum and maximum statistics to efficiently skip irrelevant data blocks, thereby reducing unnecessary I/O operations and enhancing data retrieval efficiency. Unlike indexes such as B-Trees, SMA remains effective for block skipping even when data or query patterns change, eliminating the need for immediate data layout updates or reorganization. However, the success of SMA depends on data being properly clustered with no overlapping min/max values between blocks. Simple strategies such as sorting or range-partitioning can achieve this for one-dimensional data. Nonetheless, managing enterprise-level wide tables, which can encompass hundreds to thousands of dimensions, is a formidable task [11, 30]. For instance, sorting one column can inadvertently affect others' clustering properties. When handling workloads that cover tens or hundreds of columns, the block-skipping performance decreases significantly. In summary, this complexity brings three fundamental challenges that require the data layout to adapt to (1) high-dimensionality, (2) specific data characteristics, and (3) particular workload patterns, as detailed in Section 3.2.

To solve multi-dimensional data skipping problems, Space-Filling Curves (SFCs) such as Z-order [43] and Hilbert curves [42] have been used to convert multi-dimensional data into one-dimensional sequences, which enhance the compactness of ranges in SMA and preserving data proximity [36, 57]. However, a major drawback of SFCs is that they cannot adapt to specific workloads or data. They require a complex configuration tuning process based on factors like query selectivity and data distribution [18, 19], which is a nontrivial task and beyond human capabilities. Recent learning-based approaches [22, 37, 48] aim to tailor SFCs to specific datasets and workloads by integrating various bit-merge patterns to generate SFC combinations. Despite these innovations, SFC-based methods still face challenges with scalability in terms of performance and computational efficiency. As the number of dimensions increases, performance can degrade to near-linear search levels, and high dimensionality introduces significant computational overhead. Additionally, we argue that the optimal data layout goes far beyond the combination of SFCs. Therefore, it is essential to explore more comprehensive data layout strategies.

In this paper, we address these limitations by introducing the concept of the *adaptive curve* — an *end-to-end* projection function that maps multi-dimensional tuples to a sort key, aiming to optimize data layout for enhanced data-skipping performance and efficiency. To the best of our knowledge, the adaptive curve is the first approach to simultaneously unify and optimize the aforementioned fundamental challenges.

We propose *AdaCurve*, which approximates the optimal projection function by training a Machine Learning (ML) model. Learning the projection function offers several benefits: (1) The adaptive curve enables a more comprehensive exploration of the data layout's potential. Unlike SFCs-based methods [22, 37], the adaptive curve is a projection function without relying on the SFC family or human-designed rules such as bit merging patterns. (2) The adaptive curve is more adaptive. End-to-end learning enables automated feature selection, which makes it easier to handle high-dimensional data or complex workloads. ML algorithms, such as autoencoders [27], can analyze datasets to discover patterns and retain key information. (3) The adaptive curve generates sort keys efficiently. The idea of the learned index has demonstrated extensively that the learned component has a much faster inference time than tree traversal [34, 35] or bit interleaving [49, 56]. These advantages enable our method to generate sort orders for scalable data layouts.

Although the concept is straightforward, learning an optimal projection function is non-trivial: (1) The first challenge is how to effectively integrate the influence of both data characteristics and workload patterns. (2) The second challenge is how to accurately quantify scan cost (i.e., the number of data blocks to retrieve) to enable effective end-to-end training. *For the first challenge*, we design a compression-decompression architecture to effectively merge and filter features from both data and workloads through the process of feature compression. We utilize an attention-based feature selector [10] as the fundamental building block in the adaptive curve to learn and reduce high-dimensional data to a sparse vector, as many columns contribute minimally to the computation of the sort key. The attention-based feature selector is particularly effective in high-dimensional spaces and shares a similar motivation with traditional SFCs (as shown in Section 5.2). *To address the second challenge*, we approximate the scan cost by converting sort keys into differentiable block indices and aggregating the blocks that need to be scanned. The proposed scan cost is proven to reduce false positive blocks effectively. Lastly, with our great adaptive nature of *AdaCurve*, our learned curve can be adaptive to dynamic data and workloads by incremental model updating and partial cluster reorganization.

In summary, we conclude our main contributions as follows:

- We are the first to introduce the concept of learning an end-to-end adaptive curve, which offers a more comprehensive layout search space and greater adaptability compared to SFCs.
- We propose learning an attention-based adaptive curve to simultaneously unify and optimize data and workload features, enabling scalability to high-dimensional and dynamic scenarios. Additionally, we make the measurement of scan cost differentiable to facilitate an effective learning process.
- We conduct extensive experiments on the open-source data processing engine Spark, demonstrating the superiority of our method in block skipping. *AdaCurve* effectively optimizes datasets with up to 1,000 dimensions, showcasing robust scalability in complex data settings.

2 Related Works

Data Skipping. In analytical databases, data skipping [50] has emerged as a powerful tool for enhancing query performance and efficiency. The core idea behind data skipping is to avoid unnecessary data access, particularly in scenarios involving large datasets or scan-oriented queries. We categorize data-skipping methods into two types: *index-based* and *transformation-based*, based on whether they maintain and store an explicit index structure or rely on lightweight aggregation techniques for skipping data.

Index-based Methods. The index-based methods rely on maintaining explicit index structures, such as the B-Tree family, grid file [46], R-tree [26], or Kd-tree [12], to facilitate data skipping. These methods can effectively partition and organize data, providing precise data access, especially in multi-dimensional scenarios. However, these indexes cannot learn from specific workloads and thus lack the opportunity to achieve superior performance. Recently, Flood [44] and Tsunami [20] have introduced data space partitioning methods based on workload distribution and the Cumulative Distribution Function (CDF) of each dimension. In addition, instead of partitioning data based on data distribution, Sun et al. [52, 53] extract predicates from workload and partition data based on workload pattern. They also maintain more block-level metadata (i.e., predicates) for aggressive block skipping. Following a similar idea, Qd-tree [59] and its variants [19] propose to use fine-grained predicates from workload to partition multi-dimensional space. While fine-grained block skipping has shown potential, it carries a risk of overfitting to specific workloads and might lack the necessary robustness for adaptive learning from data.

Transformation-based Methods. Transformation-based Methods use lightweight aggregation techniques such as small materialized aggregates (SMA) [39] and zone map [25] to skip irrelevant data at a coarse block level. These methods maintain precomputed aggregate values or data distribution statistics to facilitate data skipping. Transformation-based approaches are generally more lightweight because they do not require maintaining complex index structures, making them better suited for large-scale, dynamic datasets. Their effectiveness, however, is highly dependent on data layout. Since there is no natural sort order for multi-dimensional data, transformation-based methods manipulate the blocking scheme by converting multi-dimensional data into one-dimensional sort keys. The space-filling curve (SFC) is a common transformation technique. SFCs such as, C-curve [28], Z order [43] and Hilbert curves [42] are widely used to sort multi-dimensional data in current database systems [1–4, 47]. Various works [40, 41] have demonstrated that SFCs have great locality-preservation properties, which helps cluster neighbor data points in a multidimensional space. However, SFCs require careful tuning (e.g., specifying column combinations) to achieve optimal performance and cannot adapt to workload or data characteristics. Recently, learning-based SFCs [22, 37, 48] have attempted to harness ML techniques to acquire a projection function derived from a combination of SFCs. Additionally, distance-based projection functions [17, 29, 58] have been employed to cluster multidimensional data points by sorting data based on their distance from the cluster center. However, none of these approaches have demonstrated effectiveness in high-dimensional datasets or the ability to generalize beyond the SFC family.

Index-based vs. Transformation-based Data Layouts. Both of them have distinct advantages and limitations. Despite index-based methods achieving promising performance, in this work, we focus on optimizing transformation-based methods for the following reasons: 1) Wide application. Recent trends in data warehouses leverage data compression and clustering for effective data skipping instead of index structures [62]. SMA is widely adopted in modern data analytics systems such as Amazon DynamoDB [3], Databricks [4], and HBase [47]. 2) Ease of deployment. Index-based methods can provide high precision and efficient query performance by maintaining detailed index structures. However, these methods come with storage and maintenance costs in large, high-dimensional datasets [24, 39]. Transformation-based methods with SMA are lightweight, involving only sort key generation and storing summarized information, making them more space-efficient. Especially for dynamic situation, SMA remains effective for block skipping even when data or query patterns change, allowing partial data layout updates to be performed lazily without blocking queries.

3 Problem Definition & Challenges

3.1 Problem Definition

Given a set of multi-dimensional tuples $\mathcal{T}$, each tuple t has n dimensions, denoted as $t = (d_1, d_2, \cdots, d_n)$. We aim to partition $\mathcal{T}$ into a set of fix-sized blocks, denoted as $\mathcal{B} = \{b_1, b_2, \cdots, b_k\}$, where $|b_i| = |b_j|$ for $\forall i, j$. Each block is a collection of non-overlapping subsets of $\mathcal{T}$. The union of every block b

forms $\mathcal{T}$. A block is the smallest unit for I/O operations. During query planning, each block will be determined whether it can be skipped or not for the given query q_j by an SMA index:

Definition 1 (Small Materialized Aggregates, SMA). *In database systems, SMA store precomputed summary statistics including minimums and maximums of each dimension in a block, aiming to reduce unnecessary I/O during query planning.*

Each block or query field is represented by $F(\cdot) \subseteq \mathbb{R}^n$, defined as $F(\cdot) = [min(d_1), max(d_1)] \times [min(d_2), max(d_2)] \times \ldots \times [min(d_n), max(d_n)]$. This represents the range of maximum and minimum values across all dimensions for the tuples in a block or as specified by a query. For a given block b_i and query q_j, we use a function $S(b_i, q_j)$ to denote the filtering condition of the SMA index:

$$S(b_i, q_j) = \begin{cases} 1, & \text{if } F(b_i) \cap F(q_j) \neq \emptyset \\ 0, & \text{otherwise} \end{cases} \tag{1}$$

where $S(b_i, q_j)$ is a binary function that determines whether block b_i will be scanned by query q_j or not. The filtering condition is determined by the simultaneous intersection of n dimensions. For instance, $S(b_i, q_j) = 1$ when the field defined by the block b_i intersects with the query's range.

In this paper, we mainly focus on optimizing the scan cost of a workload $\mathcal{W} = \{q_1, q_2, \cdots, q_m\}$ on the tuple set $\mathcal{T}$. We calculate the scan cost based on the number of scanned blocks:

$$C(\mathcal{W}, \mathcal{B}) = \sum_{b_i \in \mathcal{B}} \sum_{q_j \in \mathcal{W}} S(b_i, q_j) \tag{2}$$

Previous works [21, 59] have shown that the scan cost is proportional to the execution time for the scan-oriented workload.

The problem of optimal data layout and partitioning centers on identifying a partition function that maps each tuple in $\mathcal{T}$ to a specific block ID, indicating the disk storage location for each tuple. This function is required to minimize the scanning cost $C(\mathcal{W}, \mathcal{B})$ for a given workload $\mathcal{W}$. Equivalently, we aim to learn a projection function P:

$$P : \mathbb{R}^n \longrightarrow \mathbb{R},$$

which maps a n-dimensional tuple to a sort key determining the relative position of the input tuple in the tuple set. The block ID can be simply calculated as $\lfloor \text{rank(tuple)}/|b| \rfloor$. Thus, we can define the optimal sort order problem as follows:

Problem 1 (optimal sort order). *Given a set of tuple $\mathcal{T}$ and a workload $\mathcal{W}$, a skipping function S, fixed block size $|b|$, find the optimal projection function P that minimizes the scan cost $C(\mathcal{W}, \mathcal{B})$.*

SFCs are specialized solutions for the Problem 1 offering optimization in simpler scenarios. However, their performance cannot be guaranteed in complex real-world scenarios. Both traditional and recent learning-based approaches still face challenges that will be discussed in the next subsection, indicating a significant opportunity for exploration beyond the existing SFC family to discover new solutions.

3.2 Challenges

In this section, we identify three fundamental challenges. Addressing these challenges can significantly enhance the effectiveness of data skipping, which serves as the primary motivation for the development of our method.

Challenges 1: Adaptation to High Dimension. High dimensions can impede the effectiveness of data skipping. As the number of dimensions increases, the effectiveness of SFCs in preserving the locality of data, which is crucial for efficient retrieval, decreases dramatically. In our empirical study

Table 1. An empirical study on the curse of dimensionality.

	1D	2D	3D	4D	5D	6D	7D	8D
Block #	388	182	126	90	108	136	143	152
Rows	230.9k	915.3k	405.6k	213.1k	174.7k	145.9k	145.7k	143.1k

using the TPC-H (100X) dataset, we examined the effect of dimensionality on block skipping with the Z-order curve. Table 1 shows the scan cost (number of blocks) for different column combinations. We select the most selective column in each combination. Initially, the scan cost drops from 388 blocks in 1D to 90 in 4D as dimensions increase. However, expanding from 5D to 8D reverses this trend, with scans peaking at 152 blocks for 8D, demonstrating the curse of dimensionality's impact on efficiency despite selecting "essential" dimensions.

Challenges 2: Adaptation to Data Characteristics. Data characteristics including data distribution and correlation can significantly influence the efficacy of data skipping. For the data distribution, limited value ranges in fields reduce traditional SFC effectiveness, while high cardinality fields show better SFC performance. For the data correlation, previous studies [23, 32, 45] have proven that utilizing column correlations through soft functional dependencies can significantly improve data skipping, optimizing data organization, and aiding in dimensionality reduction. Thus, a nuanced understanding of the dataset's specific characteristics to effectively tailor the data skipping is a key challenge.

Table 2. An empirical study on the effectiveness of workload pattern.

	Col1	Col2	Col3	Col4	Col5	Col6	Col7	Col8
Block #	388	398	445	552	824	811	824	842
Rows	230.9k	238.2k	265.2k	329.1k	492.1k	483.9k	492.2k	501.5k

Challenges 3: Adaptation to Workload Pattern. Customizing data access to match actual workload patterns can greatly improve data skipping by focusing on the most relevant data segments, thereby enhancing system efficiency. However, identifying an appropriate workload pattern becomes challenging when the workload is complex, involving numerous queries that span a wide range of columns [52, 59]. In our preliminary experiment, as reported in Table 2, we kept the dimensions (1D) of the sorting keys fixed and varied the sorting column to mitigate the effects of dimensionality. Rows in Table 2 represents the number of selected rows for the workload on a given column. The selectivity (as a way to measure workload patterns) largely reflects the number of blocks scanned. However, this trend may not hold under more complex conditions, necessitating further in-depth investigation.

Our Insight: Adaptability is the key. In fields like IoT and healthcare, where datasets are often high-dimensional (more than 100 dimensions) and complex [14, 55, 61], SFCs demand expert tuning. ML approaches offer a promising alternative, enabling adaptive and self-improving block-skipping mechanisms that minimize manual tuning. This advancement brings us closer to autonomous, intelligent databases that are well-suited for the dynamic demands of modern data analysis. Nevertheless, current research on ML-enhanced data skipping [22, 37] largely focuses on datasets with fewer than 10 dimensions, leaving the high-dimensional domain less explored. Our approach is a learning-based mapping function, without the constraints of traditional SFC patterns, capable of adapting to specific datasets and workloads, and surpasses what manual methods can achieve.

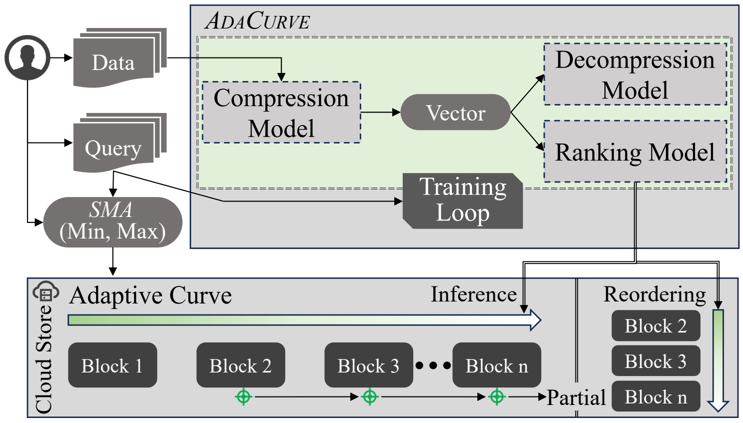


Fig. 1. Framework Overview.

4 Overview

We present an overview of a block-skipping framework called *AdaCurve*, designed to learn a block partitioning layout for multi-dimensional datasets and workloads. *AdaCurve* aims to create an optimal layout that minimizes the scan cost (as defined in Section 3.1) for specific datasets and workloads. The framework consists of two main components: (1) Online Inference, which applies the learned partition, and (2) Offline Optimization, which optimizes the block partition. The framework is depicted in Figure 1.

Online Inference. *AdaCurve* generates the data layout under the guidance of a learned adaptive curve. The adaptive curve takes a tuple as input and outputs a sort key, which assigns tuples to horizontal partitions called blocks. SMA is then generated and stored for each block. For dynamic data or workloads, we support partial and lazy updates to the adaptive curve and data layout to balance performance with the overhead of reorganization (detailed in Section 7).

Learned Adaptive Curve. *AdaCurve* approximates the optimal projection function by training a neural network. *AdaCurve* utilizes a compression-decompression architecture consisting of a compression model, a decompression model, and a ranking model as shown in Figure 1. This architecture is designed to merge data and workload features effectively. For online inference, high-dimensional data is transformed into lower-dimensional vectors by the compression model and ranked by the ranking model. During offline optimization, these models are trained concurrently to learn knowledge from both the data and the workload.

Offline Optimization. *AdaCurve* learns the adaptive curve through end-to-end backpropagation, reducing reliance on manual feature engineering and heuristic tuning. This phase includes two key aspects: *learning from workload* and *learning from data*. Learning from workload uses a differentiable sorting operator to measure approximate scan costs. To reduce false positives in block scanning, we enhance the training procedure by contrastive learning. Learning from data employs an autoencoder to compress and then reconstruct input vectors, capturing essential data patterns and reducing dimensionality.

Note that both learning processes are integrated and scaled through a training loop designed for large datasets and workloads, ensuring the adaptive curve is robust and well-tuned before deployment. This approach facilitates a seamless transition to real-world data handling scenarios.

5 Adaptive Curve

AdaCurve aims to learn an adaptive curve that is tailored to specific data and workloads for block skipping. In this section, we first describe the featurization strategy. Then we illustrate how to build an adaptive curve model.

5.1 Featurization

The featurization module will encode basic features from multi-dimensional tuples into embedding vectors for neural network consumption. Previous work [33, 60] use one-hot encoding or embedding to represent high-dimensional tuples. However, these methods are limited by the domain size and number of dimensions, which can lead to a large number of parameters and redundant features.

Since *AdaCurve* aims to learn the optimal sort key for a multi-dimensional dataset, the sort keys from every single attribute are key factors. Therefore, we propose to project every column into *rank space*. We denote the feature vector as T.

Rank Space Encoding. Specifically, for every column A, we first get the domain of the column denoted as $\text{dom}(A) = \{v_1, v_2, ..., v_n\}$, where v is the unique value from column A. Then we sort the $dom(A) = \{v_{i_1}, v_{i_2}, ..., v_{i_n}\}$ based on its value, where $v_{i_1} < v_{i_2} < ... < v_{i_n}$. Next we build k number of buckets $b_j = \{v_{i_j}, v_{i_{j+1}}, ..., v_{i_{j+m}}\}$ for $\text{dom}(A)$, where $1 <= j <= k$ and there are m elements in the bucket. Finally, column A_i can be mapped into the bucket index $T(v_i) = j$ satisfying $v_i \in b_j$. For example, column $A = \{\text{cherry}, \text{apple}, \text{banana}\}$ can be mapped to rank index $\{3, 1, 2\}$ based on alphabetical order. Such mapping can be achieved by scanning the dataset or acquiring from the histogram. The size of bucket k can be tuned based on the complexity of data distribution or data shifting. Large k makes the feature more accurate because it can better capture the fine-grained differences in the data. Large k loses the ability to generalize to new data because it makes the feature more dependent on the training data. Therefore, it is important to choose the size of k carefully based on the specific application.

Design Space. For high-dimensional data, richer encodings can be used, such as: 1) Adding the probability distribution value of each unique value can make the model more robust in cases where the data and workload are changing. 2) Using bit encoding for each value (similar to SFCs). In this study, only the rank value is used as a feature, and it is able to perform well.

5.2 Adaptive with Attention

Next, we present *AdaCurve* learning-based adaptive curve, which consists of our attention blocks. It aims to give the sort key for a given tuple. Before we give a formal description of *AdaCurve* neural network architecture, it is worth briefly introducing the human-engineering projection function.

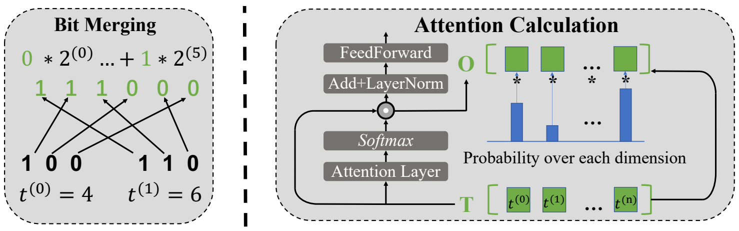


Fig. 2. Attention Mechanism Compared with Bit Merging.

Background. SFCs are essentially functions that compress high-dimensional space information into one dimension. The bit merging operation of SFCs can be expressed as the following equation.

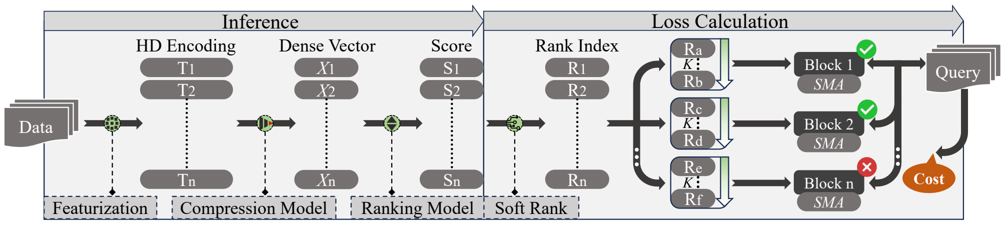


Fig. 3. Scan Cost Measurement Pipline.

$$f(x;\theta) = \sum_{i=1}^{d} \sum_{j=1}^{K} \theta_j^{(i)} \cdot t_j^{(i)}. \tag{3}$$

The equation provided represents a generic form of a function that could be used to map a tuple t in a d-dimensional space to a point in one-dimensional space, using a set of parameters θ. θ could be thought of as weights applied to each element t_j ($t_j \in t$) of the high-dimensional space. $\sum_{i=1}^{d} \sum_{j=1}^{K}$ indicates that the function sums over all dimensions d and all components K within each dimension. The concept of bit merging in SFCs can be seen as a form of *weighted sum* where each bit or component from the multi-dimensional data is combined in a way that reflects its position within the overall structure of the curve. The weights (in this case, represented by the parameters $\theta \in \{2^0, 2^1, \ldots, 2^{d \cdot K}\}$ determine the significance of each bit's position when calculating the final one-dimensional value. For tasks like data indexing and retrieval, it allows for efficient storage and search operations.

However, human-engineered patterns in SFCs are difficult to be effective and adaptive in complex situations. Therefore, we argue that it is better to learn a mapping function without rule constraints, which can be adaptively served for specific data and workloads.

Attention Block. The attention mechanism in deep learning models, particularly in neural networks, is used to weigh the importance of different parts of the input data [10]. It's a dynamic process where the weights (attention scores) are learned from the data itself during the training process. In our setting, the SFC family can be regarded as a special solution to achieve dimensional transformation. In comparison, as shown in Figure 2, the attention mechanism is more comprehensive.

Our attention mechanism translates input data into attention scores using a network of four linear transformations and nonlinear activations. The attention scores, $A = f_{\text{attention}}(T)$, are derived directly from the input T through this network. These scores are then normalized into a probability distribution, known as attention weights $W = \text{Softmax}(A)$, using the softmax function.

The final output O is the element-wise product of the original input T and the attention weights W, which scale each element of the input by its respective importance as determined by the model:

$$O = T \odot W \tag{4}$$

Compared with SFC bit merging, the attention mechanism is data-driven and learns to assign weights during the training of the neural network, while SFC bit merging uses a fixed, predetermined set of weights. Mathematically, the attention mechanism's weights are probabilities that sum up to 1 over each dimension, ensuring a distribution of "focus" across the dimensions. SFC bit merging weights are usually not probabilities but are instead based on the positional significance of each bit in the binary representation.

Then, the output of the attention mechanism O is enhanced by a series of transformations including residual connections, layer normalization [9], and a feedforward network akin to the Transformer architecture [54], promoting a comprehensive integration across all dimensions.

The above description forms the basis of our attention block, which is a fundamental component of the adaptive curve. Each layer of the attention block can filter or compress input features. In addition, each attention block is stackable for deep feature learning.

6 Optimizing Adaptive Curve

In this section, we illustrate how to optimize the adaptive curve including two learning objectives: learning from workload and learning from data. Then we illustrate the whole training loop.

6.1 Learning from Workload

The score generation process is denoted as the inference part in Figure 3. *AdaCurve* utilizes a compression model M and a ranking model R that maps each high-dimensional input data T to a one-dimensional scalar S, thereby enabling the ordering of the original high-dimensional data based on such scalar. Specifically, the compression model, consisting of our attention blocks, will first compress the high-dimensional input data into a lower-dimensional dense vector X, denoted as $M : \mathbb{R}^h \rightarrow \mathbb{R}^l$. h is the original dimension and l is the predefined lower dimension. In this work, we specifically target a reduction to 16 dimensions for inputs exceeding 100 dimensions. Then, the ranking model R, consisting of basic multi-layer perceptron, will transform the lower-dimensional vector $\mathbb{R}^l$ into a scalar S, denoted as $R : \mathbb{R}^l \rightarrow \mathbb{R}$. Once all the data has been scored, the overall ordering of the data and the allocation of blocks are determined.

The scan cost measurement pipeline is shown in Figure 3. For learning from workload, we optimize the adaptive curve in two aspects: *global cost* and *local cost*. The global cost measures the clustering property of the data approximately. We measure the global cost for every true positive block, i.e., each retrieved block that contains the data points required by the query. The local cost reflects the number of false positive points so we only measure it on the false positive blocks, i.e., each retrieved block that does not contain the data points required by the query. Both cost measurement helps the model converge to a better solution, thus their summation constitutes the scan cost.

The definition of scan cost in Section 3.1 is the number of blocks that need to be scanned. A key challenge in measuring the scan cost is to quantify this value accurately and make it differentiable for model training. Our key insight is that we can convert the score of each data point S into an equidistant rank index based on its sort order, denoted as $R(S)$, and then calculate the scan cost by adding or subtracting these indices. However, the ranking operator is typically non-differentiable. To solve that, we employ a soft rank operator as an approximation to convert the scoring values into ranking indices [13]. The soft rank operator approximates the ranking indices by transforming the ranking process into a differentiable operation through linear programming within the permutation polytope, thus enabling gradient-based optimization.

Global Cost. To this end, *AdaCurve* designs a differentiable cost calculation method with the supervision from the given workload. Once we obtain the ranking indices $R(s)$, for every true positive block, we accumulate the global cost by calculating the differences between adjacent true positive data points in those blocks denoted as following:

$$L_{\text{GlobalCost}} = \sum_{S_i, S_j \in B_k} (|R(S_i) - R(S_j)|), \tag{5}$$

where $R(S_i)$ and $R(S_j)$ denote the rank indices of tuples within the same scanned blocks, and $R(S_i)$ and $R(S_j)$ are adjacent true positive points. The identification of true positive points is based on the supervision of workloads.

As shown in Figure 4, the red area represents the blocks for which the cost needs to be computed, with three blocks covered by the query: B2, B3, and B4 based on updated SMA for each block. We will use rank indices for all the true positive data (denoted in red range) that needs to be scanned in the above three blocks. In Figure 4, assuming the first query range covers indices 6 to 9, then the differences between all points from 6 to 9 need to be calculated sequentially.

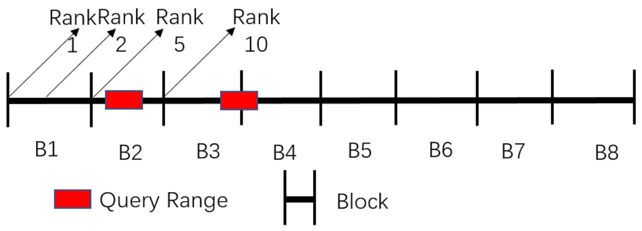


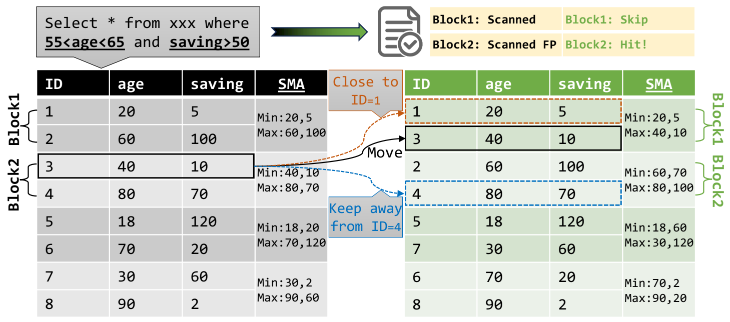
Fig. 4. Measure Global Cost in a Differentiable Way.

Local Cost. Eliminating false positives is a critical issue for block retrieval to achieve better performance. Therefore, we design a local cost metric to approximate the scan cost of false positive data blocks and help reduce false positives simultaneously. We use Figure 5 to illustrate the basic idea.

We assume that each two rows of data constitutes a block, and the minimum and maximum values of each block are given. The query predicate is to select records with ages between 55 and 65 and saving greater than 50. In this example, for the black table on the left, because the min-max range of each block overlaps with the query condition, the query engine cannot skip any block.

In the case of block 2, despite the tuples within failing to satisfy the specified query predicates (e.g., the age for ID 4 is 80, exceeding the targeted range of 55 to 65 years), the range of age within this block (40 to 80 years) intersects with the query predicate. Consequently, block 2 cannot be skipped, under the presumption that it might contain relevant data. In this context, "False Positive" signifies the mistaken presumption that a block may contain data meeting the query's criteria, potentially causing redundant scan cost. In practice, this phenomenon is exceedingly common, particularly in the context of queries with low selectivity.

By leveraging the principles of Supervised Contrastive Learning (SCL) as outlined by Khosla et al. [31], we train our model to exhibit a clustering behavior characterized by two key strategies: 1) *Encouraging divergence within false positive points*: Within a block identified as a false positive, where a pair of tuples intersect with the query's range, we encourage these tuples to be positioned as far apart from each other as possible. This strategy aims to mitigate the impact of false positives by reducing their likelihood of collectively influencing block skipping. 2) *Promoting proximity among tuples located at the block boundaries*: Conversely, we advocate for the close arrangement of tuple pairs especially at the block boundaries that do not intersect with the query's range. In this way, we enhance the model's ability to form coherent clusters of non-relevant data, thereby facilitating block skipping. For example, in blocks identified as false positives where tuples cause the SMA range to intersect with the query condition, we aim to distance such tuples as much as possible (e.g., ID 3 and ID 4). Conversely, we strive to position tuples like ID 3 and ID 1 closer together, as they can facilitate block skipping. The local cost loss is defined as follows:



```
Select * from xxx where
55<age<65 and saving>50
```

ID	age	saving	SMA
1	20	5	Min:20,5 Max:60,100
2	60	100	
3	40	10	Min:40,10 Max:80,70
4	80	70	
5	18	120	Min:18,20 Max:70,120
6	70	20	
7	30	60	Min:30,2 Max:90,60
8	90	2	

ID	age	saving	SMA
1	20	5	Min:20,5 Max:40,10
3	40	10	
2	60	100	Min:60,70 Max:80,100
4	80	70	
5	18	120	Min:18,60 Max:30,120
7	30	60	
6	70	20	Min:70,2 Max:90,20
8	90	2	

Fig. 5. Eliminating False Positive with Supervised Contrastive Learning.

$$L_{\text{LocalCost}} = R(S_{\max}) - R(S_{\min}) - \frac{\sum_{a \in A} \left(R(S^a_{T_{\max}}) - R(S^a_{T_{\min}}) \right)}{|A|}, \tag{6}$$

the first part, $R(S_{\max}) - R(S_{\min})$, represents the distance between a tuple pair located at the block boundary, having the largest and smallest rank indices, respectively. In second part, $R(S^a_{T_{\max}}) - R(S^a_{T_{\min}}$, contains tuple $T_{\max}$ and the tuple $T_{\min}$ since the interval of the two points overlaps with the query condition. Specifically, we pick data points with the maximum and minimum value for each column $a \in A$.

Finally, the total scan cost is approximated by the summation of global cost and local cost:

$$L_{\text{ScanCost}} = L_{\text{GlobalCost}} + L_{\text{LocalCost}}, \tag{7}$$

the adaptive curve targets to minimize the overall scan cost, optimizing for both the clustering of true positive points (global cost) and the minimization of false positive blocks (local cost). The similar ranking-based methods are also applied in other database learned components [15, 16].

6.2 Learning from the Data

Autoencoders [27] are powerful tools for learning representations of high-dimensional data, often used for dimensionality reduction and feature learning. Autoencoders can compress high-dimensional data into a lower-dimensional space (latent space). This is particularly useful to alleviate the curse of dimensionality. By learning to reconstruct the original input, autoencoders inherently learn useful features from the data. This can lead to more efficient representations that can improve the performance of downstream tasks.

Consider an autoencoder with an input $T \in \mathbb{R}^h$, where h is the dimensionality of the input space. The autoencoder consists of two main components: the encoder and the decoder. The encoder M maps the input to a lower-dimensional space Z:

$$M : \mathbb{R}^h \to \mathbb{R}^l, \quad Z = M(T; \theta_M) \tag{8}$$

where $l < h$ and θ_M represents the parameters of the encoder. It is noteworthy that the encoder mentioned here refers to the compression model M we discussed earlier.

The decoder D attempts to reconstruct the input from the compressed representation Z:

$$D : \mathbb{R}^l \to \mathbb{R}^h, \quad \hat{T} = D(Z; \theta_D) \tag{9}$$

where θ_D represents the parameters of the decoder.

The reconstruction loss L is typically defined as the mean squared error between the input T and the reconstructed output $\hat{T}$:

$$L_{AE}(T, \hat{T}) = \frac{1}{n} \sum_{i=1}^{n} (T_i - \hat{T}_i)^2 \tag{10}$$

The goal of training an autoencoder is to minimize the reconstruction loss by adjusting the parameters θ_M and θ_D.

6.3 Training Loop

To scale the training process to a larger dataset and reduce the training overhead, *AdaCurve* training loop is conducted on a sample of the original dataset (in practice, we choose 1000 rows in one iteration), which involves four steps:

1) Prioritized Sampling: This step involves selectively extracting a subset of the data that reflects the most commonly accessed points in user queries. In this way, we focus the training of our adaptive curve on the data segments that are most impactful for query performance.

2) Model-Predicted Sorting: In this phase, our ML model evaluates each tuple and assigns it a ranking score. These scores are used to rank the data and we recollect each block SMA.

3) Identification of Blocks for Scanning: Leveraging the query predicates, this step identifies which blocks of data will be scanned based on the corresponding SMA.

4) Model Training: Here, the sampled and ranking data serves as the training set for our ML model. We measure loss function as a weighted sum of all individual losses:

$$L_{\text{overall}} = \alpha L_{\text{AE}} + L_{\text{scancost}} \tag{11}$$

where α is the weights that balance the importance of each loss component. The objective of the training process is to minimize $L_{overall}$ by adjusting the model parameters.

Through iterative learning, the model progressively refines its predicted ranking score, enhancing its ability to forecast the index positions of tuples with better data-skipping performance.

7 Dynamic Data and Workload

In this section, we articulate how *AdaCurve* adapts to dynamic data and workload shifts. With the evolving nature of data and workloads, fully re-partitioning the entire dataset becomes impractical and costly, particularly with the influx of real-time data disrupting the original clustering and sorting integrity. An alternative approach focuses on sorting only the incoming data, ensuring improved block partitioning for new entries. However, this method may lead to increasing disorder in the overall dataset over time. The primary challenge here is to strike a balance between minimizing the impact of data reorganization and enhancing data-skipping efficiency.

To achieve this equilibrium, our strategy emphasizes maintaining *local order* to uphold the efficiency of processing new data and workloads. As the dataset and workload evolve, previously organized data can become disorganized. We have two assumptions:

- The disorganized part is on a relatively small scale but disproportionately contributes to the disorganization.
- The law of diminishing return applies to the re-partitioning efforts.

For example, data that is incrementally updated has its new property, such as a collection of tuples from a new date, which primarily affects localized areas rather than the entire data structure. It requires only minimal reorganization. Considering its relatively small scale compared to the historical data, comprehensive re-partitioning becomes increasingly less effective, consuming resources that outweigh the marginal gains achieved. Similarly, as new workloads increasingly focus on specific data regions, targeted adjustments rather than extensive re-partitioning can restore order efficiently. Consequently, our focus is on identifying the disorganized partitions and ensuring local order for operational efficiency.

When data or query patterns shift, SMA remains effective for block skipping without requiring immediate data layout updates or reorganization. Partial data layout and model updates can be performed lazily without interrupting queries. Therefore, our method does not impede the influx of new data, while updating the model to adapt to disorganized data. Specifically, our method comprises a three-step approach: 1) Model Adaptation, ensuring *AdaCurve* aligns with the current data and workload; 2) Locating Unorganized Partitions by using the updated model to identify poorly organized blocks; 3) Partial Cluster Reorganization, selectively restructuring data to boost efficiency and coherence without overhauling the entire dataset.

7.1 Model Adaptation

When changes in data, denoted as $\Delta\mathcal{T}$, or in workload, denoted as $\Delta\mathcal{W}$, are observed, the user or database administrator (DBA) can trigger optimization of the current data layout. As an initial step, we refine the adaptive curve to ensure it remains aligned with these dynamics. We employ two techniques to ensure the model adaptation is lightweight: partial rebucketing and incremental updates.

7.1.1 Partial Rebucketing . When data updates occur, $\Delta\mathcal{T}$ can surpass the original rank space encoding as outlined in Section 5.1, potentially leading to unpredictable outcomes from the adaptive curve. To effectively manage the input feature space, we monitor variability within each bucket, using manually set thresholds to initiate partial rebucketing. For instance, if the variance within a bucket exceeds 20% of the bucket's mean value, this triggers a review of the bucket boundaries. Then, buckets that violate this threshold are recalibrated by interpolating between the values at the percentile edges of the data within that bucket rather than undergoing a complete dataset recalibration. This method ensures efficiency when changes in data distribution are localized to specific segments of the dataset.

7.1.2 Incremental Updates. The incremental update is a lightweight and robust model training strategy facing out-of-distribution situations. We adopt Experience Replay (ER) [51], where models are continuously updated with new data. The key aspect of ER is balancing learning from new data while retaining knowledge from the old data by interleaving training on both new and old data. Our approach utilizes ER by prioritizing new data and new workloads during sampling. For the new data $\Delta\mathcal{T}$, we incorporate a strategy where recent updates to the dataset are given a higher probability of being revisited in subsequent training iterations. For the dynamic workload $\Delta\mathcal{W}$, each batch of newly sampled data will be retained if it is within the new workload query range, and these will be sampled at a higher probability for later training iteration. Thus, the model can rapidly adapt to changing conditions while maintaining a comprehensive understanding based on its accumulated knowledge.

7.2 Locating Unorganized Partition.

The incrementally trained adaptive curve provides an improved storage layout for dynamic data and workloads. Then we leverage the updated adaptive curve to locate the most unorganized segments of data efficiently. This process involves splitting the entire dataset into continuous segments Seg $= \{b_i, b_{i+1}, \ldots, b_j\}$ of a given size (involving several blocks). Then, for every data segment, the Coefficient of Variation (CV) is adopted to evaluate the disorganization. CV is a statistical measure of the relative dispersion or variability of data points around a dataset's mean. Specifically, the disorganization is defined as $O_{\text{Seg}} = \sum_{b_i \in \text{Seg}} \text{CV}_i = \sum_{b_i \in \text{Seg}} \frac{\sigma_{b_i}}{\mu_{b_i}}$, where σ and μ are calculated by the updated sort key from the adaptive curve. σ is the standard deviation of the updated sort key and μ is the mean of the updated sort key for each $b_j \in \text{Seg}$. Once identify the unorganized segment, we prioritize to re-partition it.

Note that, the updated adaptive curve is used as an approximation to locate the disorganization of each segment, as model inference is fast. Generally, a larger CV indicates a greater potential performance improvement from re-partitioning. Periodically, we can increase the size of segments to encompass more blocks, thereby facilitating the merging and re-partitioning of larger segments. This approach helps optimize data organization and enhance system performance efficiently.

7.3 Partial Cluster Reorganization

The Partial Cluster Reorganization (PCR) technique in columnar storage emphasizes maintaining local order rather than global order across the entire dataset. It achieves this by sorting data into predefined segments based on specific columns. This local sorting, typically executed every 70 Compression Units (CU) (approximately every 4.2 million rows), results in clusters of data that are locally ordered within their respective segments. The segment size can be specified by users. It significantly enhances query efficiency within those local segments while keeping the overall data organization manageable and efficient.

When integrated with *AdaCurve*, this system is further optimized. Once PCR finishes a segment, it swaps the existing blocks. During periods of low activity, *AdaCurve*, in conjunction with the VACUUM process, can automatically merge and reorganize segments that have become disorganized, thus maintaining the system's efficiency and data order, especially in dynamic business environments where data is continually changing.

8 Experiments

8.1 Setup

Table 3. Summary of Datasets

Datasets	Type	Dimension	#Rows	Size on Disk
Lineitem	Synthetic	3–16	60M	7.2G
Syn-Uni	Synthetic	3–1000	10M	46G
Syn-Skew	Synthetic	3–1000	10M	46G
IoT	Real	3–115	7M	7.6G

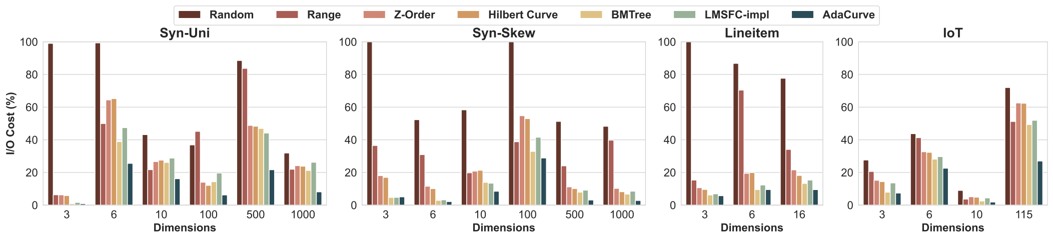


Fig. 6. I/O Cost of the Skewed Workload Results on Spark.

Datasets. We use the following datasets in our evaluation (summarized in Table 3):

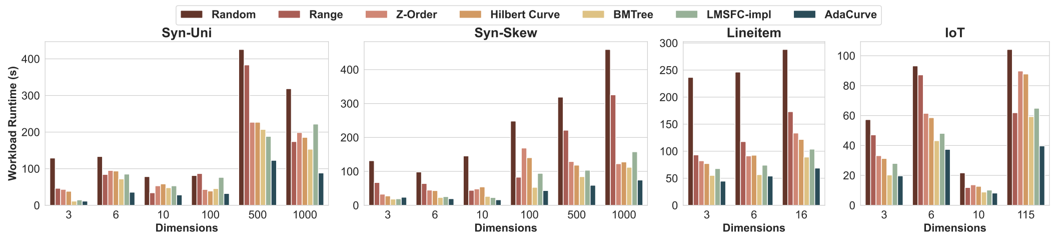
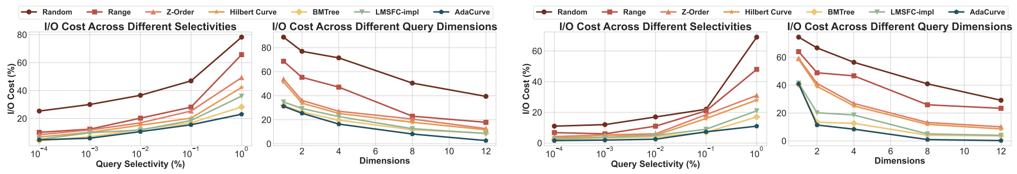


Fig. 7. Latency the Skewed Workload Results on Spark.



(a) Vary Uniform Workload's Selectivity and Dimensions.

(b) Vary Skewed Workload's Selectivity and Dimensions.

Fig. 8. Comparison of Workload's Selectivity and Dimensions: Uniform vs. Skewed Workloads.

• Lineitem: Lineitem is the fact table from TPC-H [6], an industry-standard benchmark for evaluating analytics workloads. We use a scale factor of 10. We utilize read-heavy queries derived from template Q6 to test data skipping performance. We also extend Q6 to cover more attributes.
• Syn-Uni: We generate the synthetic data from uniform distributions with 3 to 1000 dimensions. Every dimension of data is sampled from a distinct uniform distribution.
• Syn-Skew: We generate synthetic data with skewed distributions with 3 to 1000 dimensions. The first half of dimensions is generated from skewed Gaussian distribution. For the remaining dimensions, we use a random walk model, which is widely used in high-dimensional datasets [30].
• IoT: IoT [38] is a real-world botnet dataset, especially for the IoT, derived from 9 commercial IoT devices authentically infected by Mirai and BASHLITE.
Workloads. We use scan-intensive queries aligning with the format of template Q6 from TPC-H. Following previous works [22, 37], the configuration of a workload is determined by its central point and the span it covers along each dimension. We create the central points for these queries using one of two methods: (i) In the **Skewed (Skew)** mode, the central points are selected randomly from the set of existing data points. We fixed the number of central points to 15. (ii) In the **Uniform (Uni)** mode, the central points are chosen at random from across the entire scope of the data space. Every workload will cover all dimensions for the given table.
Metrics. We evaluate performance based on two metrics:
• I/O Cost: The logic evaluation on the number of blocks accessed across the entire workload for different fixed-sized partitioning strategies. To provide a clear comparison, we normalize the I/O cost relative to the maximum number of blocks scaling it to a range from 0% to 100%.
• Runtime: Workload runtime is defined as the sum of per-query latencies.
Implementation and Analytics System. We use Python to implement all models and algorithms. For all experiments, we use a fixed block size of 60000, unless specified otherwise. We evaluate logic I/O cost in Python as different analytics systems may have different underlying implementations. We implement a shallow integration of different block partition methods into database systems by adding an extra block ID column to each table, then ask the database system to sort and write

blocks based on this column. For *AdaCurve* model, we use 2 layers of attention block with ReLu activation and 16 dimension for compression and decompression model. The ranking model is a MLP with 4 linear layers. The number of parameters ranging 203K (0.8MB) from 1.5M (5.7MB). The weight α is set to 1 by default.

We use Spark (version 3.3) [8] as the execution engine since it supports pushing the predicate down to the data source. We compare the effectiveness of block partition based on SMA without any traditional index, which is common in OLAP systems. To be specific, we leverage Spark integrated with Delta Lake features [7], which allow for block partitioning and the storage of block-level SMA, to manage data storage and retrieval. Spark operates on Parquet file format on disk. We sort the data and then evenly split it into file blocks. This method is consistent across all baseline layouts. Spark is deployed on a Ubuntu server equipped with an Intel(R) Xeon(R) Silver 4214 2.20GHz CPU with 48 cores, 256GB DDR4 main memory, and a 1TB HDD.

Baselines. We compare *AdaCurve* against the following baselines:

• **Random**. Random is the insertion of data in no specific order.

• **Range**. Range is sorting data based on the most frequently accessed single column from the workload.

• **Z-Order**. Z-Order is a space-filling curve that maps multidimensional data to one dimension by interleaving the bits of the coordinates of the points.

• **BMTree** [37] is an innovative SFC that employs piecewise SFCs optimized for multi-dimensional data. We adopt their source code [5] to conduct experiment on our datasets. We try our best to adapt BMTree to wide tables (e.g., tables with hundreds of columns). We limit the bit length to 18 and set the sampling rate as 0.05 and the max depth as 20. For datasets with more than 10 dimensions, we manually select the 10 most frequently accessed columns.

• **LMSFC-impl** [22]. LMSFC is a SFC-based method optimized for multi-dimensional data. It optimizes an adaptive SFC with SMBO. LMSFC is not open-sourced, we reproduce the LMSFC's SMBO learning algorithm denoted as LMSFC-impl. We set the sampling rate to 0.01.

Note that every SFC-based method needs to be tuned carefully to achieve high performance. Thus, we select the three most frequently accessed columns from the workload to construct the Z-Order and Hilbert curve as existing work [20]. For the learning-based methods, we do not compare with Qd-tree [59] since it does not fit in the realm of the transformation-based method.

8.2 Overall Results

8.2.1 Adaptation to various datasets. In this experiment, the performance of different data layout strategies is evaluated under a skewed workload. The baselines compared include Random, Range, Z-Order, Hilbert, BMTree, and LMSFC-impl using Apache Spark with a block size set to 60,000. The I/O cost is reported in Figure 6 and the latency performance is shown in Figure 7.

In terms of I/O cost, we observe that *AdaCurve* achieves the best performance across all tested datasets, followed by BMTree, LMSFC-impl, and Hilbert, with Z-Order, Range, and Random showing relatively poorer results. Specifically, *AdaCurve* demonstrates significant improvements over traditional methods, such as Hilbert and Z-Order, achieving an average improvement of 2.8× (from 1.4× to 7.2×). Compared to Range and Random, which are the default options in many database systems, *AdaCurve* achieves even larger reductions in I/O cost, improving performance by 4.9× to 14.2× over Range and by 14.2× to 113.8× over Random. In comparison to learning-based methods like BMTree and LMSFC-impl, *AdaCurve* also shows an advantage, achieving an average enhancement of 1.6× over BMTree, with improvements ranging from 0.9× to 2.6×, and an average improvement of 1.9× over LMSFC-impl, ranging from 0.9× to 3.2×. On 3-dimensional datasets, our method demonstrates similar performance to both BMTree and LMSFC-impl.

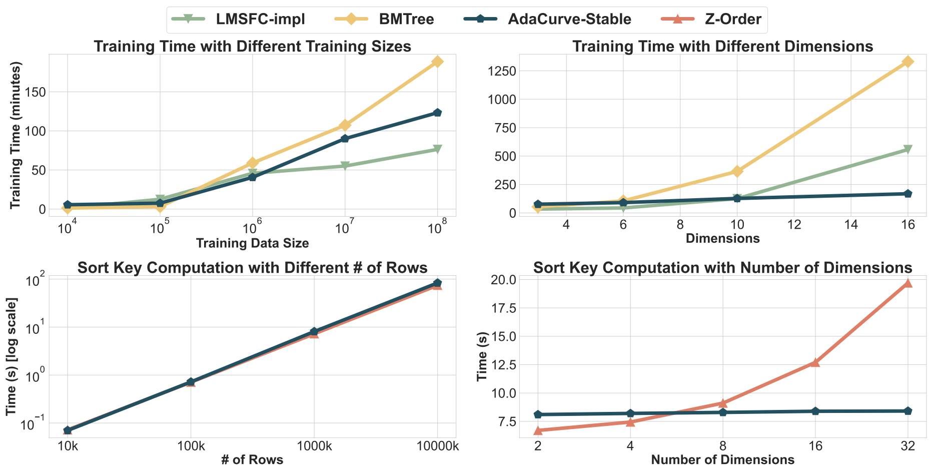


Fig. 9. Training and Sort Key Computation Overhead.

In terms of latency, *AdaCurve* also achieves the best performance across all tested datasets, significantly outperforming traditional methods like Hilbert Curve and Z-Order by an average of 2.1× (from 1.2× to 3.9×). Compared to Range and Random, *AdaCurve* achieves even larger reductions, averaging 2.6× to 4.7× improvement. Against learning-based methods such as BMTree and LMSFC-impl, *AdaCurve* shows an average enhancement of 1.3× to 1.7×. These results reinforce *AdaCurve*'s superiority in managing both I/O cost and latency, particularly for high-dimensional data and complex workloads.

Compared to learning-based methods like BMTree and LMSFC-impl, *AdaCurve*'s performance advantage is more pronounced in high-dimensional datasets. Specifically, for datasets with more than 10 dimensions, *AdaCurve* shows consistent improvements over BMTree and LMSFC-impl, with an average enhancement of 2.1× in these scenarios. In contrast, for low-dimensional datasets (less than or equal to 10 dimensions), *AdaCurve*'s performance is on par with these methods, showing similar I/O cost efficiency.

The performance gain of *AdaCurve* comes from (1) its ability to integrate both data and workload characteristics effectively, optimizing the access patterns accordingly, (2) *AdaCurve* leverages an adaptive attention mechanism, allowing it to adapt to wide tables and effectively manage complex relationships among different features, further enhancing its efficiency.

Interestingly, Z-Order and Hilbert, as multi-dimensional indexes, fail to outperform the Range method on high-dimensional datasets such as Syn-Uni-1000, Syn-Skew-100, and IoT-115. Despite their theoretical advantages in preserving spatial locality, the performance of space-filling curves (SFCs) can degrade significantly on wide tables. This suggests that (1) the choice of data-skipping method can have a significant impact on query performance, and (2) the curse of dimensionality poses challenges for SFCs when dealing with high-dimensional data. This highlights the necessity of adaptive approaches like *AdaCurve*.

8.2.2 Adaptation to various queries. In this experiment, we evaluate the performance of *AdaCurve* against all baselines under both uniform and skewed workloads, with various query selectivity ranging from 0.0001% to 1% and various query dimensions ranging from 1 to 12. Figure 8a and Figure 8b show the I/O cost across different query selectivities and dimensions for uniform and skewed workloads respectively.

In the uniform workload, the results indicate that *AdaCurve* consistently outperforms all other methods, followed by BMTree, Hilbert Curve, Z-Order, LMSFC-impl, Range, and Random. Specifically, *AdaCurve* achieves an average improvement of approximately 1.4× compared to Hilbert Curve and an improvement of about 1.6× compared to Z-Order. *AdaCurve* is 1.2× better and 1.5× better than BMTree and LMSFC-impl at query selectivity 1% since *AdaCurve* can more effectively skip irrelevant data and focus only on the required blocks. For different query dimensions, *AdaCurve* 's advantage becomes more evident in high-dimensional queries. At 12 dimensions, *AdaCurve* 's I/O cost was 2.67%, significantly outperforming all other methods.

In the skewed workload, *AdaCurve* achieves even better performance on skewed datasets compared to the baselines. *AdaCurve* continues to outperform all baselines, demonstrating its robustness across different types of query distributions. For example, At a query selectivity of 1%, it is 22.5× better than Random, 12.4× better than Range, 10.4× better than Z-Order, and 9.3× better than Hilbert Curve. When compared to learning-based methods like BMTree and LMSFC-impl, *AdaCurve* shows an improvement of 7.5× and 7.3×, respectively.

This superior performance can be attributed to *AdaCurve*'s ability to effectively integrate data characteristics with workload features, along with its use of an adaptive attention network that is well-suited for handling high-dimensional datasets.

8.3 Overhead

Figure 9 provides an empirical analysis of training overhead and sort key computation overhead under various training data sizes and dimensions.

The top row of plots illustrates the training time required for different methods, including LMSFC-impl, BMTree, *AdaCurve*-Stable, across increasing data sizes and dimensions. *AdaCurve*-Stable means training the adaptive curve to a stable performance. The results show that BMTree experiences a rapid increase in training time as the data size and dimensionality grow, particularly beyond 10^7 dataset size and beyond 6 dimensions, respectively. This is because SFC-based methods' action space of bit merging patterns grows exponentially with the increase in dimensions, leading to a significant increase in the complexity of finding optimal bit combinations. *AdaCurve*-Stable maintains a relatively stable training time as both training data size and dimensionality increase, demonstrating scalability advantages over the baselines.

The bottom row of Figure 9 presents the time required to generate sort keys with respect to different numbers of rows and dimensions. Since BMTree and LMSFC-impl exhibit similar efficiency to Z-Order, their plots are omitted for clarity. For Z-Order, in order to scale to higher dimensions, we limit the bit length to 20 and use vectorized execution with NumPy to optimize the computation. The results show that Z-Order's key generation time increases significantly with higher dimensions. In contrast, *AdaCurve*-Stable demonstrates a nearly constant key generation time as the dimensionality increases, indicating *AdaCurve*'s scalability and efficiency in handling high-dimensional datasets. This is because high dimensions are filtered and computed in parallel using matrix operations in *AdaCurve*. For different row counts under the same dimension, both methods exhibit linear scaling.

Overall, these graphs demonstrate that *AdaCurve* offers a more scalable solution, particularly when dealing with large and high-dimensional datasets, by selectively focusing on the most relevant dimensions and leveraging efficient matrix-based computations.

8.4 Dynamic Data and Workload

Dynamic Data. Figure 10 and Table 4 present the evaluation of *AdaCurve* on dynamic data. The left line plot in Figure 10 shows the trend of the updated I/O cost with model adaption technique

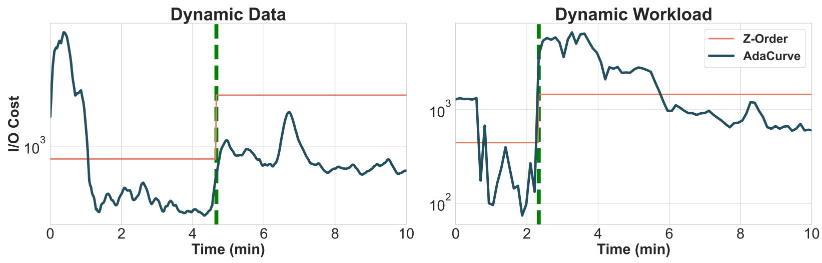


Fig. 10. Training Time of Model Adaption for Dynamic Data and Workload. Dash Line Indicates Data or Workload Shifts.

Table 4. Number of Blocks to Be Skipped with Data Insertion and Partial Cluster Reorganization.

Insert	Metrics	PCR	Full PCR	Full Sort	Total Blocks
2 batch	# Blocks Skipped	32	41	48	68
	Train+Reorg time (min)	3.5	6.9	10.2	-
4 batch	# Blocks Skipped	64	99	108	136
	Train+Reorg time (min)	4.5	14.5	18.6	-
6 batch	# Blocks Skipped	96	143	171	204
	Train+Reorg time (min)	4.5	20.8	26.3	-

Table 5. Number of Blocks to Be Skipped with Workload Shift and Partial Cluster Reorganization.

% of Shift	Metrics	PCR	Full PCR	Full Sort	Total Blocks
10%	# Blocks Skipped	580	621	649	1000
	Train+Reorg time (min)	10.2	18.2	34.2	-
30%	# Blocks Skipped	513	588	623	1000
	Train+Reorg time (min)	10.5	21.9	34.2	-
90%	# Blocks Skipped	471	611	683	1000
	Train+Reorg time (min)	11.2	23.9	34.2	-

over time, with the green dashed line indicating the point of data insertion. The inserted data maintains the same distribution as the original data. As shown in the plot, *AdaCurve* adapts quickly to the inserted data, stabilizing with minimal fluctuations within a short period (approximately 2-3 minutes). This efficiency is attributed to *AdaCurve*'s bucket encoding mechanism and the implementation of a rapid, lightweight model adaptation approach.

Table 4 details the number of blocks skipped with data insertion and partial cluster reorganization (PCR). The reorganization methods are defined as follows: PCR represents our partial training and update approach, Full PCR involves local reorganization of the entire dataset, and Full Sort represents complete sorting of the dataset. One batch of insert is 200k rows. The results indicate that, with 6 batches inserted, PCR requires only about 17.1% of the time needed for Full Sort $\left(\frac{4.5}{26.3} \times 100\%\right)$, while achieving approximately 56.1% of the effectiveness in terms of blocks skipped $\left(\frac{96}{171} \times 100\%\right)$. This demonstrates the efficiency of the PCR method, balancing reduced re-organization overhead with effective data skipping, making it well-suited for dynamic data environments.

Dynamic Workload. Figure 10 and Table 5 present the evaluation of *AdaCurve* on dynamic workloads. For dynamic workloads, the right line plot in Figure 11 shows the trend of the updated I/O cost over time. In this experiment, we varied 50% of the workload. Compared to data changes,

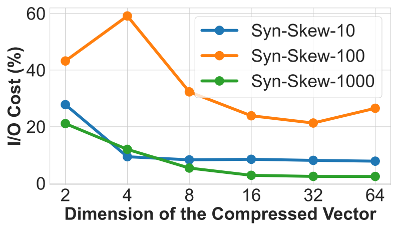


Fig. 11. Varying Compressed Vector Dimension.

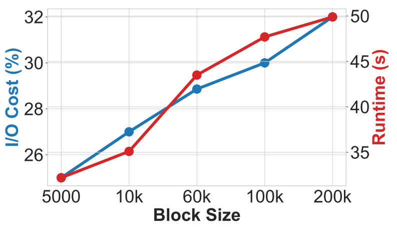


Fig. 12. Varying Block Size for Performance.

Table 6. Varying the Number of Samples.

Sample Number	100	300	500	1,000	1,500
I/O Cost (%)	34.2	32.8	29.3	28.8	28.1
Training Time (min)	23.7	64.6	78.4	90.1	173.5

adapting to workload changes is more challenging. Initially, there was significant fluctuation, and *AdaCurve* took approximately 7-8 minutes to stabilize. After the workload shift, *AdaCurve* adapts quickly, reducing I/O cost significantly compared to the baseline Z-Order. This quick adaptation is due to *AdaCurve*'s flexible model adaptation mechanism, which allows for efficient workload-specific optimization.

Table 5 details the number of blocks skipped with workload shift and PCR. The results indicate that, for a 90% workload shift, PCR requires only about 32.7% of the time needed for Full Sort $\left(\frac{11.2}{34.2} \times 100\%\right)$, while achieving approximately 69.0% of the effectiveness in terms of blocks skipped $\left(\frac{471}{683} \times 100\%\right)$. This demonstrates that PCR is effective in balancing reduced computational overhead while maintaining a high level of block-skipping efficiency, making it suitable for dynamic workload environments.

8.5 Design Choices

8.5.1 Varying Settings. We evaluate *AdaCurve*'s performance under various settings: the dimension of compressed vector, block size, and prioritized sampling size.

Dimension of Compressed Vector. Based on Figure 11, we analyze the effect of the dimension of the compressed vector on the performance of *AdaCurve* under three datasets: Syn-Skewed-10, Syn-Skewed-100, and Syn-Skewed-1000. For low-dimensional datasets (Syn-Skewed-10), 4 or 8 dimensions suffice, as less information needs capturing, whereas training with 2 dimensions failed due to insufficient representation. For higher-dimensional datasets (Syn-Skewed-100 and 1000), higher-dimensional compressed vectors yield better performance. This suggests that for complex data, retaining more dimensions helps capture richer information. However, we observe some performance fluctuation for Syn-Skewed-1000, particularly at higher dimensions. This fluctuation might be attributed to limited training loops, making it challenging for model convergence. Thus, we select 16 dimensions compressed vectors for datasets with more than 100 dimensions.

Block Size. Figure 12 shows the effect of varying block sizes on I/O cost and runtime for Syn-Skewed-100. The results indicate that smaller block sizes perform better in terms of both I/O cost and runtime. This can be attributed to the fact that, given the skewed nature of the dataset, smaller blocks provide better locality, which enhances data-skipping effectiveness. However, in practice,

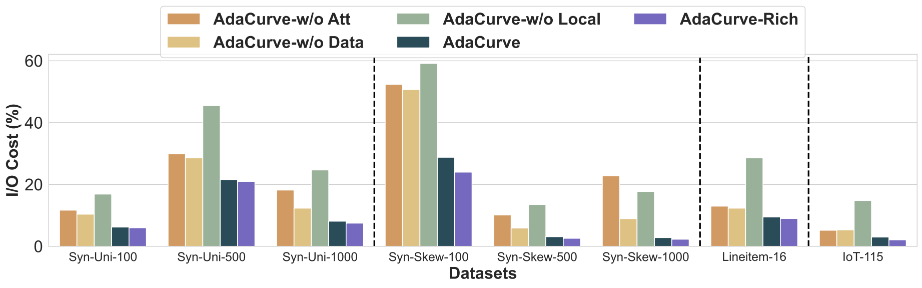


Fig. 13. I/O Cost of *AdaCurve* Variants.

smaller block sizes can lead to increased metadata overhead. In practice, we typically choose a fixed block size of 60,000 rows, providing a reasonable trade-off between performance and overhead.
Prioritized Sampling Size. Table 6 presents the impact of varying the sample number per training loop on I/O cost and training time for the Syn-Skew-100 dataset. The results show that larger sample sizes improve model convergence by accumulating more diverse examples, thereby reducing the influence of noise. However, accurately estimating scan cost is a key challenge in data layout generation, and excessively large sample sizes can introduce errors in rank index calculations (implemented via soft rank as an approximation). On the other hand, smaller sample sizes are processed faster, allowing for more frequent model iterations, which can enhance overall training efficiency. Therefore, a balance must be struck between convergence quality and computational efficiency. In practice, we use 1,000 samples as the sampling size.

8.5.2 AdaCurve Variants. Figure 13 shows the I/O cost of *AdaCurve* variants: *AdaCurve*-w/o Att replaces the attention module with a linear network; *AdaCurve*-w/o Local removes the local cost calculation; *AdaCurve*-w/o Data removes the data reconstruction loss; and *AdaCurve*-Rich adds workload features as extra encoding.

The results reveal several key insights. (1) For *AdaCurve*-w/o Attn, removing the attention module significantly degrades performance, especially on workload-intensive and high-dimensional datasets (e.g., Syn-Skew-100 and Syn-Uni-500), indicating the importance of attention in capturing complex patterns. (2) *AdaCurve*-w/o Data shows a noticeable impact on skewed and IoT datasets due to its inability to adequately capture complex, high-dimensional data distributions, leading to higher I/O costs. (3) *AdaCurve*-w/o Local results in the greatest impact, as the local cost component effectively reduces false positives for block skipping, which is particularly beneficial for low-selectivity workloads, as observed in the IoT dataset. (4) *AdaCurve*-Rich, with richer encoding from workload features, consistently improves performance across all datasets. Our encoding method includes the frequency of each dimension in the workload as well as selectivity, enhancing the representation quality. We believe there is more room for exploration of this encoding approach in future work.

Overall, these observations highlight the crucial roles of each component in *AdaCurve*, and show how different modifications impact its ability to achieve efficient data skipping and optimal layout generation.

8.6 Behaviors Learned by *AdaCurve*

To gain insights, we now analyze why *AdaCurve* can self-improve to specific data and workload. Figure 14 illustrates the dynamism of the attention mechanism in *AdaCurve*, which fine-tunes its focus on relevant data features across learning epochs. The study involves a six-dimensional

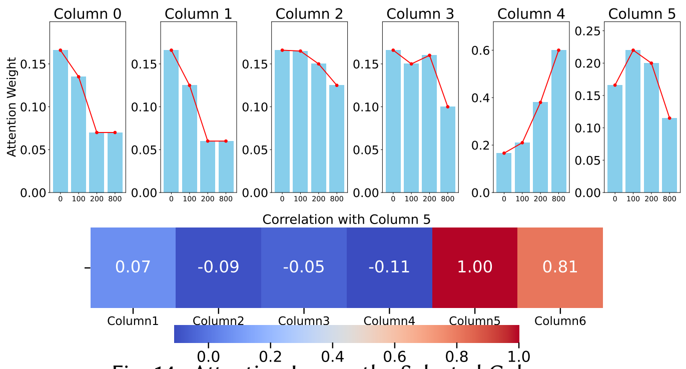


Fig. 14. Attention Learns the Selected Columns.

dataset, with queries specifically targeting the fifth and sixth dimensions. Each bar graph illustrates the weights assigned by the attention module to each dimension across different epochs, labeled 0, 100, 200, and 800.

How to learn from workload. Initially distributing weights evenly, the attention model evolves, discerning and assigning greater weights to the fifth and sixth columns, corresponding to their query significance. By epoch 800, the fifth column, in particular, is weighted heavily, indicative of its critical role in query processing.

How to learn from data. The heatmap below the bar graphs represents the correlation coefficients between each column and the pivotal fifth column. Initially, the model identifies potential significance in Column 6, increasing its attention weight. However, it intelligently reduces the weight since Columns 5 and 6 are highly correlated. This nuanced adjustment is a testament to the model's effective dimensionality reduction strategy, ensuring computational resources are allocated to the most informative features, thereby enhancing overall performance and efficiency in query processing.

This progression demonstrates the model's learning process as it iteratively refines its attention scores, increasingly focusing on the dimensions of interest. The attention mechanism's adaptability is evident, as it aligns its weights to both workload and data perspective.

9 Conclusion

In this study, we introduced the *AdaCurve*, an innovative strategy for optimizing multi-dimensional data layouts that harnesses the power of machine learning to adapt dynamically to changing data and query patterns. *AdaCurve* represents a paradigm shift in database storage layout, employing machine learning to adaptively and dynamically optimize data access patterns. The method's continuous optimization and adaptability to data and workload changes culminate in a robust framework that significantly improves query efficiency. This underscores the transformative potential of machine learning based approaches in optimizing database systems for enhanced performance.

Acknowledgments

This work is partially supported by NSFC (No. 62472068), Shenzhen Municipal Science and Technology R&D Funding Basic Research Program (JCYJ20210324133607021), and Municipal Government of Quzhou under Grant (No. 2023D044).

References

[1] 2016. Compound and Interleaved Sort Keys. https://aws.amazon.com/cn/blogs/big-data/amazon-redshift-engineerings-advanced-table-design-playbook-compound-and-interleaved-sort-keys/.

[2] 2017. Attribute Clustering. https://docs.oracle.com/database/121/DWHSG/attcluster.htm.

[3] 2017. Z-Order Indexing for Multifaceted Queries in Amazon DynamoDB: Part 1. https://aws.amazon.com/cn/blogs/database/z-order-indexing-for-multifaceted-queries-in-amazon-dynamodb-part-1/.

[4] 2018. Processing Petabytes of Data in Seconds with Databricks Delta. https://www.databricks.com/blog/2018/07/31/processing-petabytes-of-data-in-seconds-with-databricks-delta.html.

[5] 2023. BMTree source code. https://github.com/gravesprite/Learned-BMTree.

[6] 2024. TPC-H. https://www.tpc.org/tpch/.

[7] Michael Armbrust, Tathagata Das, Liwen Sun, Burak Yavuz, Shixiong Zhu, Mukul Murthy, Joseph Torres, Herman van Hovell, Adrian Ionescu, Alicja Łuszczak, et al. 2020. Delta lake: high-performance ACID table storage over cloud object stores. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3411–3424.

[8] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. 2015. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*. 1383–1394.

[9] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. 2016. Layer normalization. *arXiv preprint arXiv:1607.06450* (2016).

[10] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473* (2014).

[11] Ladjel Bellatreche. 2012. *Dimension table selection strategies to referential partition a fact table of relational data warehouses*. Springer.

[12] Jon Louis Bentley. 1975. Multidimensional binary search trees used for associative searching. *Commun. ACM* 18, 9 (1975), 509–517.

[13] Mathieu Blondel, Olivier Teboul, Quentin Berthet, and Josip Djolonga. 2020. Fast differentiable sorting and ranking. In *International Conference on Machine Learning*. PMLR, 950–959.

[14] Ralf Bousseljot, Dieter Kreiseler, and Allard Schnabel. 1995. Nutzung der EKG-Signaldatenbank CARDIODAT der PTB über das Internet. (1995).

[15] Xu Chen, Haitian Chen, Zibo Liang, Shuncheng Liu, Jinghong Wang, Kai Zeng, Han Su, and Kai Zheng. 2023. Leon: A new framework for ml-aided query optimization. *Proceedings of the VLDB Endowment* 16, 9 (2023), 2261–2273.

[16] Xu Chen, Zhen Wang, Shuncheng Liu, Yaliang Li, Kai Zeng, Bolin Ding, Jingren Zhou, Han Su, and Kai Zheng. 2023. Base: Bridging the gap between cost and latency for query optimization. *Proceedings of the VLDB Endowment* 16, 8 (2023), 1958–1966.

[17] Angjela Davitkova, Evica Milchevski, and Sebastian Michel. 2020. The ML-Index: A Multidimensional, Learned Index for Point, Range, and Nearest-Neighbor Queries.. In *EDBT*. 407–410.

[18] Jialin Ding, Matt Abrams, Sanghita Bandyopadhyay, Luciano Di Palma, Yanzhu Ji, Davide Pagano, Gopal Paliwal, Panos Parchas, Pascal Pfeil, Orestis Polychroniou, et al. 2024. Automated multidimensional data layouts in Amazon Redshift. (2024).

[19] Jialin Ding, Umar Farooq Minhas, Badrish Chandramouli, Chi Wang, Yinan Li, Ying Li, Donald Kossmann, Johannes Gehrke, and Tim Kraska. 2021. Instance-optimized data layouts for cloud analytics workloads. In *Proceedings of the 2021 International Conference on Management of Data*. 418–431.

[20] Jialin Ding, Vikram Nathan, Mohammad Alizadeh, and Tim Kraska. 2020. Tsunami: A learned multi-dimensional index for correlated data and skewed workloads. *arXiv preprint arXiv:2006.13282* (2020).

[21] Haowen Dong, Chengliang Chai, Yuyu Luo, Jiabin Liu, Jianhua Feng, and Chaoqun Zhan. 2022. Rw-tree: A learned workload-aware framework for r-tree construction. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 2073–2085.

[22] Jian Gao, Xin Cao, Xin Yao, Gong Zhang, and Wei Wang. 2023. LMSFC: A Novel Multidimensional Index based on Learned Monotonic Space Filling Curves. *arXiv preprint arXiv:2304.12635* (2023).

[23] Behzad Ghaffari, Ali Hadian, and Thomas Heinis. 2020. Leveraging soft functional dependencies for indexing multi-dimensional data. *arXiv preprint arXiv:2006.16393* (2020).

[24] Goetz Graefe. 2009. Fast loads and fast queries. In *International Conference on Data Warehousing and Knowledge Discovery*. Springer, 111–124.

[25] Goetz Graefe and Harumi Kuno. 2010. Fast loads and queries. In *Transactions on Large-Scale Data-and Knowledge-Centered Systems II*. Springer, 31–72.

[26] Antonin Guttman. 1984. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*. 47–57.

[27] Geoffrey E Hinton and Ruslan R Salakhutdinov. 2006. Reducing the dimensionality of data with neural networks. *science* 313, 5786 (2006), 504–507.

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

[29] Hosagrahar V Jagadish, Beng Chin Ooi, Kian-Lee Tan, Cui Yu, and Rui Zhang. 2005. iDistance: An adaptive B+-tree based indexing method for nearest neighbor search. *ACM Transactions on Database Systems (TODS)* 30, 2 (2005), 364–397.

[30] Rong Kang, Wentao Wu, Chen Wang, Ce Zhang, and Jianmin Wang. 2021. The case for ml-enhanced high-dimensional indexes. In *Proceedings of the 3rd International Workshop on Applied AI for Database Systems and Applications*.

[31] Prannay Khosla, Piotr Teterwak, Chen Wang, Aaron Sarna, Yonglong Tian, Phillip Isola, Aaron Maschinot, Ce Liu, and Dilip Krishnan. 2020. Supervised contrastive learning. *Advances in neural information processing systems* 33 (2020).

[32] Hideaki Kimura, George Huo, Alexander Rasin, Samuel Madden, and Stanley B Zdonik. 2009. Correlation maps: A compressed access method for exploiting soft functional dependencies. *Proceedings of the VLDB Endowment* 2, 1 (2009), 1222–1233.

[33] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, and Alfons Kemper. 2018. Learned cardinalities: Estimating correlated joins with deep learning. *arXiv preprint arXiv:1809.00677* (2018).

[34] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. 2019. SOSD: A benchmark for learned indexes. *arXiv preprint arXiv:1911.13014* (2019).

[35] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The case for learned index structures. In *Proceedings of the 2018 international conference on management of data*. 489–504.

[36] Jonathan K Lawder and Peter JH King. 2000. Using space-filling curves for multi-dimensional indexing. In *British National Conference on Databases*. Springer, 20–35.

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

[38] Yair Meidan, Michael Bohadana, Yael Mathov, Yisroel Mirsky, Asaf Shabtai, Dominik Breitenbacher, and Yuval Elovici. 2018. N-baiot—network-based detection of iot botnet attacks using deep autoencoders. *IEEE Pervasive Computing* 17, 3 (2018), 12–22.

[39] Guido Moerkotte. 1998. Small materialized aggregates: A light weight index structure for data warehousing. *None* (1998).

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

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

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

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

[44] Vikram Nathan, Jialin Ding, Mohammad Alizadeh, and Tim Kraska. 2020. Learning multi-dimensional indexes. In *Proceedings of the 2020 ACM SIGMOD international conference on management of data*. 985–1000.

[45] Vikram Nathan, Jialin Ding, Tim Kraska, and Mohammad Alizadeh. 2020. Cortex: Harnessing correlations to boost query performance. *arXiv preprint arXiv:2012.06683* (2020).

[46] Jürg Nievergelt, Hans Hinterberger, and Kenneth C Sevcik. 1984. The grid file: An adaptable, symmetric multikey file structure. *ACM Transactions on Database Systems (TODS)* 9, 1 (1984), 38–71.

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

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

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

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

[51] David Rolnick, Arun Ahuja, Jonathan Schwarz, Timothy Lillicrap, and Gregory Wayne. 2019. Experience replay for continual learning. *Advances in neural information processing systems* 32 (2019).

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

[53] Liwen Sun, Michael J Franklin, Jiannan Wang, and Eugene Wu. 2016. Skipping-oriented partitioning for columnar layouts. *Proceedings of the VLDB Endowment* 10, 4 (2016), 421–432.

[54] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*. 5998–6008.

[55] Patrick Wagner, Nils Strodthoff, Ralf-Dieter Bousseljot, Dieter Kreiseler, Fatima I Lunze, Wojciech Samek, and Tobias Schaeffter. 2020. PTB-XL, a large publicly available electrocardiography dataset. *Scientific data* 7, 1 (2020), 154.

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

[57] Xinyang Wang, Yu Sun, Qiao Sun, Weiwei Lin, James Z Wang, and Wei Li. 2023. HCIndex: a Hilbert-Curve-based clustering index for efficient multi-dimensional queries for cloud storage systems. *Cluster Computing* 26, 3 (2023), 2011–2025.

[58] Pan Xu, Cuong Nguyen, and Srikanta Tirthapura. 2018. Onion curve: A space filling curve with near-optimal clustering. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 1236–1239.

[59] Zongheng Yang, Badrish Chandramouli, Chi Wang, Johannes Gehrke, Yinan Li, Umar Farooq Minhas, Per-Åke Larson, Donald Kossmann, and Rajeev Acharya. 2020. Qd-tree: Learning data layouts for big data analytics. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 193–208.

[60] Zongheng Yang, Eric Liang, Amog Kamsetty, Chenggang Wu, Yan Duan, Xi Chen, Pieter Abbeel, Joseph M Hellerstein, Sanjay Krishnan, and Ion Stoica. 2019. Deep unsupervised cardinality estimation. *arXiv preprint arXiv:1905.04278* (2019).

[61] Azeema Yaseen, Mohsin Nazir, Aneeqa Sabah, Shahzadi Tayyaba, Zuhaib Ashfaq Khan, Muhammad Waseem Ashraf, and Muhammad Ovais Ahmad. 2020. Dimensionality reduction for internet of things using the cuckoo search algorithm: reduced implications of mesh sensor technologies. *Wireless Communications and Mobile Computing* 2020 (2020), 1–21.

[62] Mohamed Ziauddin, Andrew Witkowski, You Jung Kim, Dmitry Potapov, Janaki Lahorani, and Murali Krishna. 2017. Dimensions based data clustering and zone maps. *Proceedings of the VLDB Endowment* 10, 12 (2017), 1622–1633.

Received July 2024; revised September 2024; accepted November 2024