A Design and Implementation of Learned Index for Processing Multi-Dimensional Queries Over Relational Data

Parmiss Shahinfard

A Thesis
in
The Department
of
Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements
for the Degree of
Master of Science (Computer Science) at
Concordia University
Montréal, Québec, Canada

June 2025

CONCORDIA UNIVERSITY
School of Graduate Studies

This is to certify that the thesis prepared

By: **Parmiss Shahinfard**

Entitled: **A Design and Implementation of Learned Index for Processing Multi-Dimensional Queries Over Relational Data**

and submitted in partial fulfillment of the requirements for the degree of

Master of Science(Computer Science)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the Final Examining Committee:

_____ Chair
Dr. Sabine Bergler

_____ Examiner
Dr. Sabine Bergler

_____ Examiner
Dr. Gosta Grahne

_____ Supervisor
Dr. Nematollaah Shiri

Approved by _____
Dr. Joey Paquet , Chair
Department of Computer Science and Software Engineering

_____ 2025

_____
Dr. Mourad Debbabi, Dean
Gina Cody School of Engineering and Computer Science

Abstract

We study the performance evaluation and analysis of Flood, a learned index designed to process multi-dimensional queries over relational data. Unlike traditional indexing methods such as KD-Trees and R-Trees, which rely on static partitioning strategies, Flood leverages machine learning techniques to dynamically adapt its grid structure and data layout based on data distribution and query workloads.

We identify and evaluate the core components of the Flood framework—including grid-based partitioning, learned layout optimization, and refinement steps—and assess the impact of each component on overall system performance. We compare Flood's efficiency and scalability against baselines such as KD-Trees, Z-order indexing, and brute-force scan across multiple datasets and query workloads. Our experiments reveal that scan emerges as the dominant bottleneck, with layout tuning and component configuration introducing significant overhead.

To gain a deeper understanding of Flood and explore opportunities for improvement, we developed a modular prototype implementation from scratch. This modular design enabled a systematic, in-depth performance study by isolating components and allowing for alternative configurations. We also refined the cost model calibration and proposed a new optimization strategy for guiding layout selection.

Our work contributes to more effective configuration and tuning of learned indexes by offering insights into the trade-offs, limitations, and opportunities of using learned indexes as an alternative or complementary solution for supporting multi-dimensional queries in relational database systems.

Acknowledgments

This research was partially supported by Faculty Research Support (FRS) grants from Concordia University.

I would like to express my deepest gratitude to my supervisor, Dr. Shiri, for his continuous support, insightful guidance, and encouragement throughout the course of this research. I am also thankful for the opportunity to serve as a teaching assistant for the Advanced Database Systems course, taught by Dr. Shiri. This role allowed me to draft the course project inspired by the themes of this thesis, particularly in multi-dimensional indexing, and in return, several student projects across two semesters sparked new ideas and perspectives that influenced my work. Engaging with students and seeing their interpretations of learned indexing problems provided valuable insights that helped refine my own thinking. This interplay between teaching and research significantly enriched my understanding of the subject and contributed meaningfully to the development of this thesis.

Last but not least, I want to thank my mother and father for their never-ending support for my education and personal growth, and for the life and opportunities they have given me. I am forever grateful.

Contents

List of Figures

List of Abbreviations

BST Binary Search Tree. 44, 53

CDF Cumulative Distribution Function. viii, 6, 17, 26–28, 43, 44, 52, 54

DBMS Database Management System. 20

I/O Input/Output. 20

KD-Tree K-Dimensional Tree. 4, 7, 11

MD Multi-Dimensional. vi, 16, 19, 21, 22, 25

ML Machine Learning. 3, 6, 16, 18, 22, 24

$\mathbf{N_c}$ Number of Intersecting Cells. 45, 55, 57

$\mathbf{N_s}$ Number of Scanned records. 45, 55, 57

PGM Piecewise Geometric Model. 22

PLM Piecewise Linear Model. 34, 43, 53

QuadTree Quadrant Tree. 22

R-Tree Rectangle Tree. 4, 7, 12, 22

RF Random Forest. 24

RMI Recursive Model Index. v, 4, 6, 7, 18, 28, 44, 52

SSTable Sorted String Table. 20

XGBoost eXtreme Gradient Boosting. 24, 45

Z-order Z-order Indexing. 7, 13

Chapter 1

Introduction

In an era where information is growing exponentially, the ability to efficiently query and retrieve relevant information from massive datasets has become crucial. Many data-intensive applications today increasingly rely on rapid access to high-dimensional data. From online retail analytics and supply chain optimization to geospatial systems and scientific simulations, these modern applications have workloads that require filtering across multiple attributes to drive timely and accurate decisions. The core technology to support such applications is the choice of indexing strategies. In this thesis, our focus is on indexing structured relational data, where records are organized in tables with well-defined attributes and types, and queries typically involve multi-attribute filtering over large row-oriented datasets.

Although traditional multidimensional index structures have long been foundational, they struggle to keep up with the size, complexity, and evolving nature of contemporary datasets (as discussed in Chapter 2). These limitations have sparked research into alternative strategies, with particular attention on "learned indexes" that adapt to both data and query workloads using Machine Learning (ML) techniques.

Recent advancements in Machine Learning have created a new, exciting research direction in database systems: *ML for Databases*. This direction aims to improve core database modules using machine learning techniques. Notable examples of active research areas in this growing field include self-driving databases [2], learned query optimizers [3], and learned indexes [4]. These early systems have already shown promising performance gains over traditional methods, paving the way for new database research and development directions.

Among these innovations, learned indexes rethink the indexing problem as a prediction task. Instead of traversing fixed structures like B-Trees, they train ML models to predict a

key's position based on the data's patterns. This is done using error-correction techniques that verify predictions during lookup. The nature of an index problem traditionally necessitates full (i.e., 100%) accuracy, so relying solely on ML models to completely replace conventional structures raises some concerns and challenges. This guided research in hybrid methods tries to balance efficiency and reliability by combining learned models with conventional index structures.

The Recursive Model Index (RMI) [4] is a major solution and also a pioneer of this hybrid design. It preserves the hierarchical structure of a tree while replacing each node with a trained ML model. This layered design allows RMI to benefit from the efficiency of ML predictions while retaining the robustness of tree-based search methods. Although introduced relatively recently, the concept of learned indexes has expanded the research scope at a quick pace and attracted increased research attention.

With the success of one-dimensional learned indexes, researchers have recently begun to explore extending them to multi-dimensional data. Traditional methods like K-Dimensional Tree (KD-Tree) [5], Rectangle Tree (R-Tree) [6], and space-filling curves such as Z-ordering [7] have been the backbone of multi-dimensional query processing for years. However, these methods face persistent challenges such as poor scalability in high-dimensional spaces, sensitivity to data skew, difficulty in maintaining balanced partitions, and inefficiencies in handling dynamic workloads, which are discussed in depth in Chapter 2. For this reason, there has been growing interest in applying learned indexing techniques to multi-dimensional data. This poses new challenges, however. One-dimensional learned indexes typically assume that the data is sorted along a single attribute, which allows for efficient model training and lookup. However, in multi-dimensional settings, there is no inherently defined linear ordering across multiple attributes. This lack of natural ordering complicates the process of learning data distributions and makes it more challenging to design effective error correction mechanisms for MD data, compared to the one-dimensional case, as multi-dimensional learned indexes often require more sophisticated data layout strategies.

While RMIs [4] have proven to be effective for one-dimensional sorted data, extending these methods to multi-dimensional spaces remains a challenge. Flood [1], which will be studied in depth in Chapter 4, emerged as the first learned multi-dimensional index to dynamically adapt its grid layout based on both data and workload characteristics. However, several aspects of its design, particularly around efficiency and refinement, deserve further study, improvement, and tuning.

1.1 Motivation and Impact

While traditional indexing methods have long provided reliable solutions for ”static” data, in general, their effectiveness reduces in modern applications when faced with evolving query patterns and/or when faced with diverse, shifting data distributions. More importantly, they miss the opportunity to take advantage of the valuable insights hidden in the data itself and in the queries that are repeatedly executed over time. This naturally raises an important question: if both the data and the queries contain so much useful information, why not use it to build more adaptive and effective indexes, and how to utilize/exploit it?

This is the central idea behind learned multi-dimensional indexing. Instead of just storing data, these indexes learn from it. Flood is among the early frameworks to embrace this idea in a multi-dimensional setting over structured relational data. It is an adaptive grid-based approach designed to tackle the complexity of high-dimensional filtering and the demands of practical query workloads.

In this thesis, we provide a comprehensive study and analysis of learned index for multi-dimensional data, focusing specifically on the Flood framework. We provide practical insights by revisiting its design and implementing it from scratch, and evaluating its performance. Our research objectives focus on rigorously benchmarking Flood and evaluating its practical merits and identifying its limitations. In addition to developing the Flood indexing framework and evaluating its performance, we go deeper to investigate its internal structure and mechanics, identify bottlenecks, and identify opportunities to further improve its capabilities and scalability for large MD data.

1.2 Thesis Contributions

This thesis makes a number of contributions aimed at understanding, evaluating, and improving the learned index, focusing on the Flood framework, one of the earliest developments of learned multi-dimensional indexes. By building Flood from the ground up using a modular and tunable architecture, we identify key performance insights and propose targeted enhancements. Throughout this work, we focus on *structured relational datasets*, such as the TPC-H benchmark dataset, that simulate real-world relational database workloads involving multi-attribute filtering over tabular records.

- **Modular Design and Implementation:** We design and implement Flood following a modular architecture that cleanly separates layout construction, query execution, and optimization. This allows us to isolate and evaluate each module: projection,

refinement, and scan, under controlled workloads. Through detailed empirical analysis, we identify scan as the dominant query-time bottleneck, often accounting for over 80% of execution time. This finding encourages further efforts to reduce scan volume and improve layout effectiveness.

- **Component-Level Trade-off Studies:** We explore the interaction between refinement techniques and exact range filtering—an optional step that precisely checks whether records fall within query bounds. We also evaluate the impact of layout parameters (e.g., bin count, sort dimension), B-tree degree for refinement, and alternative flattening methods (Empirical Cumulative Distribution Function (CDF) vs. RMI) to identify robust configurations across diverse datasets and workloads.

- **Revised Cost Model**: We enhance Flood's cost model by introducing a two-stage predictive pipeline: neural networks to estimate N_c and N_s, and tree-based models (e.g., XGBoost) to estimate latency components. This improves prediction accuracy and layout optimization reliability.

- **Robust Layout Optimization:** We replace Flood's original gradient descent approach with a custom greedy coordinate descent algorithm enhanced with random restarts. This method is more stable in discrete search spaces and consistently discovers lower-cost layouts, leading to improved query performance and optimization reliability.

- **Benchmark Data and Experimental Evaluation:** We adopt the TPC-H benchmark suite and generate data and queries to evaluate Flood and its variants against traditional baselines, including KD-Trees, Z-Order Indexing, and brute-force search. Our experiments span diverse dataset scales and query workloads, providing detailed insights into how each component impacts end-to-end performance.

These contributions collectively advance the understanding and practical use of learned multi-dimensional indexing by providing a modular, tunable implementation, identifying key bottlenecks, improving cost prediction, and an improved optimization process under real-world constraints.

1.3 Thesis Organization

Our study aims to advance the understanding of learned multi-dimensional indexing and provide practical insights for future systems integrating ML into core database components.

The remainder of this thesis is organized as follows:

Chapter 2 introduces the core concepts in multi-dimensional indexing. We will review traditional techniques such as KD-Tree and R-Tree, and present the challenges posed by high-dimensional, large-scale data. Chapter 3 surveys the growing body of work on learned indexes, with a focus on RMI, Flood, and other solutions including LiSA, SageDB, and Tsunami. Chapter 4 describes the architectural design of the Flood framework, explains each component in detail, and presents our modular approach that supports configurable flattening, refinement, and scan strategies. It also introduces our proposed changes to improve the cost model and layout optimization algorithm. Chapter 5 presents our experiments and results using a different range of dataset sizes and query workloads. It includes evaluation and comparison of component-level performance breakdowns, layout optimization analysis, cost model accuracy, optimization algorithm, and benchmarking against KD-Tree, Z-order Indexing (Z-order), and brute-force scan baselines.

Chapter 2

Background

In this chapter, we review concepts and techniques related to this thesis. We begin with a brief review of basic traditional indexing structures in databases and then consider multi-dimensional counterparts.

2.1 Basic Concepts

2.1.1 Traditional Indexing Techniques

Indexing is a main technique in database systems that helps significantly improve the performance of query processing by reducing the amount of data scanned from disks when retrieving records. An index is a specialized data structure (such as a B-Tree, Hash Table, or Bitmap Index) that stores a subset of table data in an organized manner, allowing for efficient lookups and retrieval.

Indexes can be categorized based on different perspectives, as follows:

Primary vs. Secondary Indexes

- **Primary Index**: This index is built on the primary key or another unique ordering attribute of some relational data, also called a table. It determines the physical order of records and allows efficient retrieval based on key values. Only one primary index can exist per table.
- **Secondary Index**: This index is built on non-primary attribute(s). It does not affect the physical order in which the records are stored in the table and is used to speed

up lookups for queries on frequently accessed attributes. A table can have multiple secondary indexes.

Storage-Based Classification Indexing techniques can be broadly categorized based on where the index resides during query processing. Two primary types are commonly distinguished: in-memory and on-disk indexes.

- **In-Memory Indexes**: These indexes are fully stored in the computer's main memory (RAM), allowing for extremely fast lookups and low-latency access. They are particularly well-suited for real-time applications or scenarios where high-throughput query performance is critical. However, their scalability is limited by the available RAM, which may in turn restrict their applicability for very large datasets.
- **On-Disk Indexes**: In contrast to in-memory indexes, on-disk indexes are stored on persistent (Secondary) storage such as SSDs or HDDs. While they introduce higher access latency compared to in-memory indexes, they can handle significantly larger datasets and remain usable even when the data is too large to fit into the main memory. These indexes are commonly used in data warehousing, OLAP systems, and big data applications where disk-based storage is necessary.

2.2 Multi-Dimensional Indexing

Efficient data retrieval is a major challenge in database systems, particularly in analytical workloads where queries involve filtering and aggregating over multiple attributes.

Traditional database indexes are primarily optimized for one-dimensional data, meaning they efficiently handle searches based on a single attribute at a time. However, some indexing methods, such as composite indexes, can improve multi-attribute queries to some extent. Many real-world applications, however, involve queries that span multiple attributes, e.g., geospatial queries and multi-column filtering in data analytics applications. In such contexts, traditional one-dimensional indexes fail to provide efficient query performance due to their inability to optimize searches across multiple attributes. This necessitates multi-dimensional indexing.

Multi-dimensional (MD) indexes are specialized data structures that optimize queries spanning multiple attributes by employing partitioning strategies suited for two or more dimensions, such as space partitioning or bounding volume hierarchies. Unlike one-dimensional

indexes such as B-Trees [8, 9], which are optimized for single-attribute lookups or range queries, multi-dimensional indexes provide efficient support for:

- **Partial match queries**
- **Range queries**, where data points satisfying the conditions over multi-dimensional intervals are retrieved.
- **Nearest Neighbor (NN)** which identifies the closest data point(s) to a given point based on similarity across multiple attributes. A common extension is **K-Nearest Neighbor (KNN)**, which retrieves the top K most similar data points in the multi-dimensional space. **Spatial joins** are used to compare multi-dimensional objects by calculating distances or evaluating spatial conditions (e.g., whether they intersect, are within a given range, or one contains the other).

2.2.1 Applications of Multi-Dimensional Indexes

Many real-world applications require multi-dimensional indexing due to their reliance on datasets with multiple dimensions. Multi-dimensional indexes are essential in various application domains. Examples of such applications include, but are not limited to:

- **Geographic Information System (GIS)**: Efficient retrieval of spatial objects [6].
- **Big Data Analytics**: Filtering large datasets by multiple attributes (e.g., time-series analysis, sales reports).
- **Online Analytical Processing (OLAP)** : Query optimization for high-dimensional datasets.

For example, Retail Analytics applications involve queries filtering customer transactions by product type, price range, location, and time of purchase. Similarly, in Geographic Information Systems, spatial indexing efficiently retrieves data points based on latitude, longitude, altitude, and timestamps. Multimedia databases store images, videos, and audio files as complex feature vectors (e.g., color histograms, object shapes, timestamps). Images, for example, are transformed into high-dimensional feature vectors representing their most distinctive characteristics. Multi-dimensional indexes can then be used to index these feature vectors and to support similarity queries, such as kNN searches, which retrieve the top-k matching images [4].

2.3 Traditional Multi-Dimensional Indexing

Historically, multi-dimensional indexing structures have been categorized into tree-based, space-filling curve-based, and grid-based indexing techniques. These structures have been extensively studied and used in modern database management systems (DBMS), spatial databases, and data warehouses.

2.3.1 Tree-Based Indexing

Tree-based index structures utilize hierarchical partitioning to divide the data space into regions to support efficient search and query execution.

KD-Tree The structure of a KD Tree is a form of binary space partitioning, where data space is recursively divided along alternating dimensions (see Figure 2.1). At each level of the tree, a split is made along one coordinate axis, cycling through the dimensions. The left child contains points less than the split value, and the right child contains the rest.

Figure 2.1 illustrates this process: the left side shows how the space is partitioned by vertical and horizontal splits, while the right side shows the resulting binary tree structure. Each node corresponds to a region split and is labeled with the splitting dimension.

While KD Trees have been effectively used for exact match, range queries, and nearest neighbor searches in multi-dimensional (MD) data, they suffer from the following limitations:

- Scalability limitations in high-dimensional spaces [5].
- Degradation in performance due to the curse of dimensionality, making them inefficient beyond 10–15 dimensions.

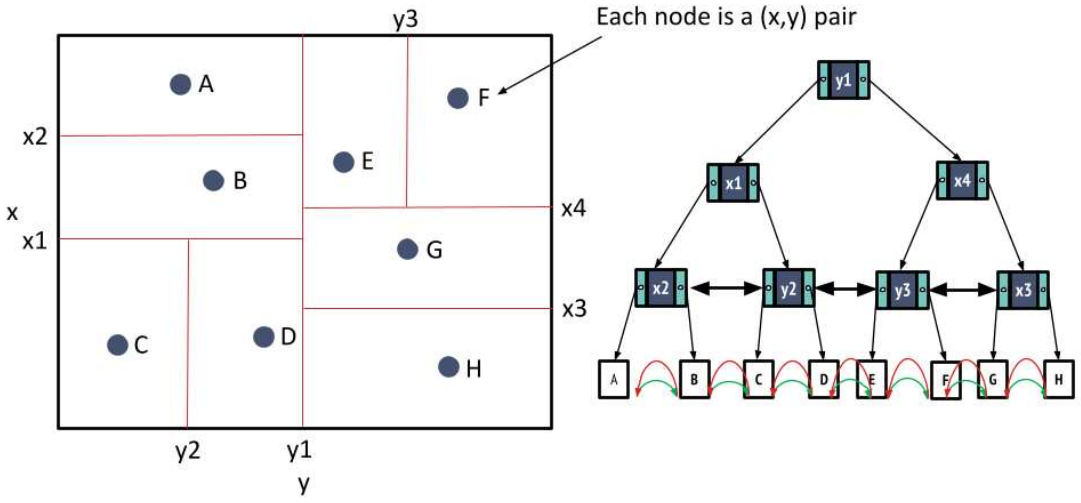

Figure 2.1: KD-Tree construction process: (Left) recursive spatial partitioning using alternating dimensions; (Right) corresponding tree structure with alternating split axes.

R-Trees and Variants R-Tree [6] are tree-based indexing structures designed for multi-dimensional data such as spatial objects. They organize data by enclosing nearby points or shapes (like rectangles or polygons) into larger rectangles called *Minimum Bounding Rectangles* (MBRs). These MBRs are then grouped hierarchically in a tree structure, allowing efficient querying for range, intersection, and nearest neighbor operations.

Figure 2.2 illustrates how R-Trees function. In part (a), data objects (numbered 1 through 9) are grouped into bounding rectangles (labeled a through d). These are further organized into higher-level rectangles (I and II), and finally combined into the root node. Part (b) shows the corresponding tree structure, where each internal node points to its bounding rectangles, and leaf nodes contain the actual data objects.

This hierarchical grouping allows spatial queries, such as "find all objects that intersect with a region", to prune large portions of the search space by ignoring branches of the tree whose MBRs do not overlap with the query.

- **R*-Trees** [10] improve upon standard R-Trees by optimizing how the tree splits nodes to minimize overlap between MBRs. This reduces unnecessary comparisons during queries.
- **Hilbert R-Trees** [11] use a space-filling curve (like Z-order or Hilbert curves) to sort and group spatial data in a way that preserves spatial locality more effectively.

Despite these enhancements, R-Trees and their variants can still struggle with skewed or irregular data distributions. As objects become unevenly spread or overlapping, MBRs may significantly intersect, increasing the number of tree nodes that must be checked during queries. This overlap reduces efficiency and often requires frequent rebalancing or reorganization of the tree.

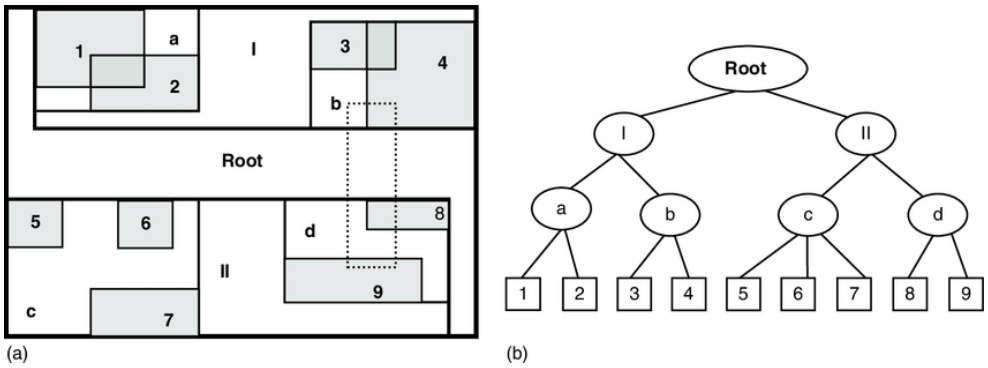

Figure 2.2: (a) Spatial partitioning using R-Tree: objects are grouped into bounding rectangles (a, b, c, d), which are then grouped into higher-level regions (I, II); (b) Corresponding tree structure with internal and leaf nodes.

2.3.2 Space-Filling Curve-Based Indexing

Space-filling curves transform multi-dimensional data into a single dimension while preserving spatial locality, enabling compatibility with traditional one-dimensional indexes, an example of which is Z-Order, explained as follows.

Z-Order (Morton Order) The Z-order index [7] is a space-filling curve that maps multi-dimensional coordinates into a single one-dimensional sequence by interleaving the binary representations of each dimension. This transformation enables sorted access to multi-attribute data, which can improve performance for range queries in some cases.

Figure 2.3 illustrates this idea: an 8×8 grid where each cell's position is encoded using a 6-bit Morton code (formed by interleaving the 3-bit x and y coordinates). The highlighted path shows how the Z-order curve visits each cell in sequence while attempting to preserve spatial locality.

Despite its advantages, Z-ordering has notable limitations. It tends to perform poorly on dynamic or highly skewed datasets, as the mapping may fail to preserve spatial locality in high-dimensional spaces. As a result, queries may scan a large number of unrelated records, leading to suboptimal performance and inefficient data access. Nevertheless, due

to its simplicity and integration into modern analytic platforms, Z-ordering continues to be used and is thus included in our experimental benchmarks.

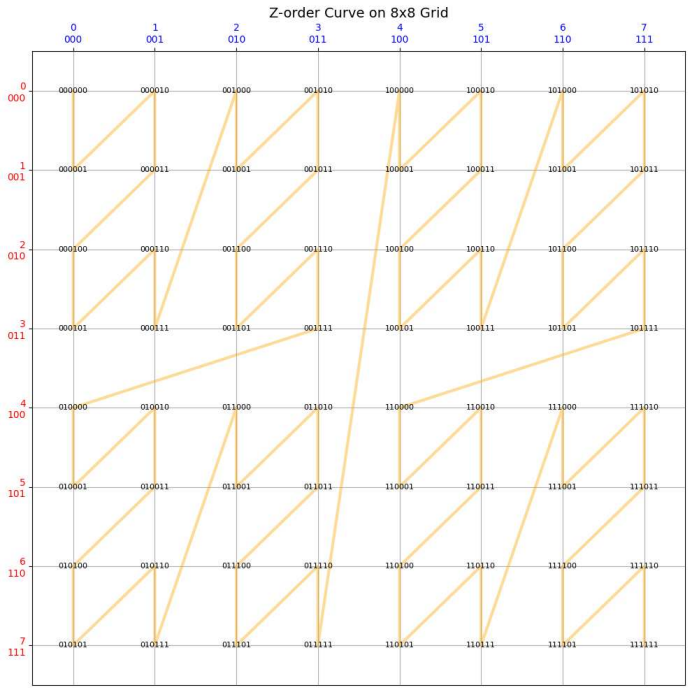

Figure 2.3: Z-order traversal on an 8×8 grid. Each cell is labeled with its Morton code, derived by interleaving binary representations of x and y coordinates. The orange path illustrates the Z-shaped access pattern.

2.4 Traditional Indexes in Practice

Despite their limitations, traditional multi-dimensional indexes such as KD-Trees, R-Trees, and Z-ordering remain widely used in practice and serve as important baselines for comparison.

For example, PostgreSQL supports KD-Tree for two-dimensional point data and uses R-Trees in the PostGIS extension to speed up spatial range and intersection queries on geometries like polygons and lines [12, 13]. MySQL offers native R-Tree indexing on geometry columns in both the InnoDB and MyISAM engines to support spatial queries [14]. Amazon Redshift leverages Z-ordering to physically cluster data blocks in analytical workloads, allowing the system to skip over large portions of irrelevant data during query execution [15].

Although rarely deployed in distributed systems due to scalability challenges in high-dimensional spaces, KD-Trees remain effective for low-dimensional use cases and are still supported in systems like PostgreSQL. In analytical and scientific computing environments, KD-Trees are frequently used for in-memory multi-dimensional queries, with implementations available in libraries such as SciPy [16].

These examples underscore the continued relevance of traditional multi-dimensional indexes and justify their inclusion as comparative baselines in this work. In this thesis, we evaluate these structures not because they are optimal, but because they provide well-understood, reproducible performance characteristics that highlight the challenges addressed by learned indexing techniques.

This chapter reviewed foundational concepts in database indexing, with a focus on traditional multi-dimensional techniques such as KD-Trees, R-Trees, and Z-ordering. We examined their design principles, practical applications, and continued relevance in modern systems like PostgreSQL, MySQL, and Redshift. While these structures remain widely used, we also highlighted their inherent limitations in handling high-dimensional, skewed, or dynamic workloads. These limitations motivate the need for more adaptive indexing strategies. In the next chapter, we explore learned index structures—an emerging class of techniques that leverage machine learning to predict data locations and adapt to distributional patterns. We begin by reviewing the core ideas behind learned indexes, their advantages over traditional methods, and the challenges they introduce, particularly in multi-dimensional settings.

Chapter 3

Related Work

In the previous chapter, we reviewed basic concepts and techniques related to indexing in databases, highlighting the significance of efficient search operations in modern database management systems. Traditional indexing structures, such as B-Trees, R-Trees, and Z-ordering, have been extensively studied and successfully integrated into practical data systems, providing robust performance guarantees [8, 9, 6, 10, 7]. However, as data volumes grow and queries become increasingly complex, these traditional structures face limitations, particularly in Multi-Dimensional (MD) scenarios.

Recent advances in Machine Learning (ML) have introduced a new paradigm in indexing called learned index, which shifts from rigid rule-based structures to adaptive models that can learn from data distribution patterns. Instead of relying solely on pre-defined hierarchical or space-partitioning rules, *learned indexes* employ ML techniques to predict the position of a key value in a dataset, offering a fundamentally different approach to indexing. This chapter provides a comprehensive review of the literature related to learned indexes, motivations, potential benefits, and the challenges they pose. We will also review results on hybrid indexing approaches that incorporate learned models within traditional structures to achieve a performance balance between accuracy and efficiency in time and space.

3.1 The Learned Index Structures

3.1.1 Reframing Indexing as a Machine Learning Problem

The concept of learned index was first introduced by Kraska et al. [4], where they proposed that an index can be formulated as a predictive model. The key idea was that a one-dimensional index, such as a B+ tree, could be *reframed as a ML problem*, where a

model learns the cumulative distribution function (CDF) of the input data. Following this idea, the traditional process of traversing a tree structure to locate a key value is replaced with a *prediction step*, where the model directly estimates the key's position in a sorted dataset.

This idea stems from the observation that, in many datasets, the relationship between key values and their positions in the dataset exhibits a recognizable pattern. If a model can effectively learn this pattern, it can predict the location of a key *without explicit node traversal*, hence reducing search complexity from $\mathcal{O}(\log N)$ steps in B-Trees for N records to an *almost constant time lookup* in normal data distributions. Unlike traditional indexes, which require maintaining explicit hierarchical structures, a learned index *stores a compact model* trained to infer the positions of records with the input search key based on the learned patterns in the data.

3.1.2 Challenges of Fully Replacing Traditional Indexes

Despite the potential efficiency gains, a complete replacement of traditional indexes with ML models has posed significant challenges. While classical index structures provide *theoretical guarantees on worst-case performance*, learned indexes, on the other hand, rely on *approximations* that may not always yield accurate search results. ML models are particularly effective at capturing the *global distribution* of data, but they often fail to maintain high accuracy when predicting individual record positions. This is because, while the *CDF function appears smooth and well-behaved on a macro scale*, it exhibits *irregularities at finer granularity*, a well-known statistical effect that hinders precise key-to-position mappings.

This limitation, referred to as the *last-mile search accuracy problem*, is particularly evident in large datasets. Even if a model can approximate the general distribution of, say, 100 million records, refining the prediction down to a few hundred positions remains challenging. Traditional indexes, such as B-Trees, mitigate this issue through *deterministic traversal mechanisms*, whereas learned indexes rely on statistical estimation, which is approximate as it can introduce unpredictable errors. Furthermore, many real-world datasets exhibit *skewed or evolving distributions*, which complicates the direct application of learned models. Unlike B-Trees, which automatically rebalance as data changes, learned indexes may require *frequent retraining* to maintain accuracy, leading to increased maintenance overhead.

3.2 RMI: A Hybrid Approach

To address the challenges associated with learned indexes in high-performance systems (that is, systems requiring low-latency, high-throughput access to large-scale data), Kraska et al. [4] proposed a hybrid architecture that integrates ML into the traditional indexing paradigm. This led to the development of the Recursive Model Index (RMI), which has since become a foundational architecture for learned indexing, particularly for one-dimensional data.

The RMI architecture is structured as a hierarchy of models designed to mimic the recursive refinement of B-Trees. At its core, RMI treats the indexing process as a prediction task: the goal is to estimate the position of a key value in a sorted array. Rather than relying on a single monolithic model, RMI adopts a multi-stage design. A coarse-grained model at the first level provides an initial estimate of a key's position based on the overall data distribution. This estimate is then refined through a cascade of smaller, specialized models, each trained on increasingly localized subsets of the data.

Each model is trained to minimize squared error over its region. Upper-stage models are trained on the full dataset, while lower-stage models are trained on data subsets routed by the predictions of their parent. RMI typically uses lightweight models such as linear regressors or shallow fully-connected neural networks, which offer low inference latency and small memory footprints compared to deeper or more expressive architectures.

To enhance lookup performance, RMI uses model-biased search. Rather than starting a binary search from the middle, the search begins at the predicted position and probes neighboring entries (e.g., $pos-\epsilon$, pos, $pos+\epsilon$), benefiting from hardware prefetching [4]. Precomputed min- and max-error bounds for each leaf model define the search window, guaranteeing correctness even when predictions are imprecise.

Training is efficient and parallelizable. The top-level model—often a linear model or a shallow neural net—converges quickly on randomized data. Lower-stage models are trained in parallel on their respective partitions. Since the models only approximate position (rather than perform exact classification), even simple architectures suffice.

This staged design offers a number of advantages:

- **Accuracy:** By breaking the search task into progressively finer regions, each model can focus on learning patterns within a limited scope, improving prediction precision.
- **Efficiency:** The use of lightweight models—such as simple linear regressors or shallow neural networks—at each stage reduces overall computation time and resource usage compared to a single large and/or complex model.

- **Robustness:** Like B-Trees, RMI gracefully handles a wide range of data distributions by allowing hierarchical refinement, which makes it resilient to skew and non-uniformity. In cases where the model error is high, RMI allows fallback to traditional B-Trees [4].

RMI's predictive nature significantly improves query efficiency for read-heavy workloads, particularly range queries over sorted data. As reported in [4], for static and sorted datasets, RMI could outperform traditional B-Trees in terms of both speed and memory usage.

3.2.1 Empirical Evaluation and Practical Trade-offs

Empirical performance evaluations of learned indexes reported in [4] demonstrate that they significantly outperform traditional B-Trees in lookup performance for MD data. Their evaluation results indicate that learned indexes can achieve up to an order of magnitude faster query processing while consuming an order of magnitude less memory than cache-optimized B-Trees. These benefits make learned indexes particularly attractive for read-heavy workloads, where query efficiency is paramount.

However, despite their advantages, learned indexes remain limited in practical deployment, particularly in MD indexing scenarios. While they work well for one-dimensional key lookups, real-world applications often involve multi-attribute queries, where filtering is performed across multiple dimensions. The hierarchical refinement approach of RMI does not extend naturally to these MD cases, leading to degraded performance when applied to complex query workloads. This limitation has resulted in further research to improve hybrid indexing approaches that can effectively combine learned models with traditional MD indexing structures, for increased query processing performance [1, 17, 18].

3.2.2 Learned Indexes at Google Scale

Although learned indexes have demonstrated strong potential in academic settings — offering reduced memory usage and faster lookups than traditional B-Trees — early designs focused primarily on in-memory use, limiting their applicability to production systems.

To address this, Abu-Libdeh et al. [19] proposed a learned index for Google's Bigtable, a distributed, disk-based key-value store. Their approach replaces Bigtable's traditional B-Tree block index with a lightweight, monotonic regression model that maps keys to

approximate disk blocks, rather than exact byte offsets. This design, integrated during Sorted String Table (SSTable) creation, ensures sorted order and compatibility with range queries.

The predicted block-to-offset mappings are stored compactly and incorporated into Bigtable's existing Input/Output (I/O) pipeline. Empirical results show a 38% reduction in tail-latency (e.g., 99th percentile response time) and a 54% increase in throughput. These improvements highlight the practicality of learned indexes when carefully adapted for large-scale, disk-resident systems.

3.3 Learned Indexing for Multi-Dimensional Data: Introducing Flood

As data volumes expand at an exponential rate, conventional multi-dimensional index structures, such as R-Trees [6] and their variants [10, 11, 20, 21, 22], have been widely adopted to enable efficient data retrieval and query processing in large-scale multi-dimensional databases. These techniques, including B+ Trees and R-Trees, have been extensively studied and integrated into mainstream Database Management Systems (DBMSs) such as Oracle [23] and PostgreSQL [24]. However, despite their widespread use, traditional indexing structures rely on static partitioning strategies and hierarchical traversal mechanisms, which may not adapt effectively to changes in data distributions and query workloads.

To overcome these limitations, researchers have investigated extending learned indexing techniques to multi-dimensional data. One of the earliest and a major advancement in this area is Flood, recognized as the first in-memory learned multi-dimensional index capable of dynamically adapting to both data distributions and query workloads [1]. This is because, unlike conventional recursive learned indexes, which extend one-dimensional models to higher-dimensional data, Flood integrates machine learning with grid-based partitioning to optimize multi-dimensional range queries.

Instead of employing a fixed hierarchical structure, Flood modifies its grid layout dynamically, refining frequently accessed regions with finer partitions while maintaining coarser indexing in regions with fewer queries.

3.4 Other MD Learned Indexing Techniques

Beyond Flood, several other learned MD techniques have been proposed, each presenting unique innovative features and trade-offs. Notable among these techniques are SageDB [25], LISA [18], and Tsunami [17], which employ different machine-learning strategies to enhance indexing performance in multi-dimensional database systems, each focusing on and improving some specific aspects involved.

3.4.1 LISA

Learned Index Structure for Spatial Data (LISA) is a disk-based learned spatial index optimized for large-scale spatial datasets [18]. Unlike traditional spatial indexing methods such as R-Trees and KD-Trees, which suffer from high disk I/O costs, LISA employs learned models to predict spatial locality and reorganize data placement accordingly. This reduces unnecessary disk access and significantly improves spatial range query performance. However, its reliance on learned spatial distributions means that sudden shifts in data patterns may require model retraining, which would increase maintenance overhead.

3.4.2 Tsunami

Tsunami is another proposed learned multi-dimensional index designed to improve performance on highly skewed workloads and datasets with strong attribute correlations [17]. It builds upon hierarchical partitioning and workload-driven adaptation. Tsunami is similar in spirit to Flood but differs in two key innovations: the Grid Tree and the Augmented Grid. The Grid Tree in Tsunami adapts partitioning granularity based on workload skew, enabling finer partitions in frequently queried regions. The Augmented Grid captures inter-attribute correlations, allowing for more efficient query processing in datasets where attributes are statistically dependent. In such cases, the value of one attribute provides information about the value of another, which enables more effective partitioning strategies than those assuming attribute independence. While Flood introduced a learned multi-dimensional indexing approach optimized for average-case workloads, Tsunami complements it by targeting scenarios characterized by workload skew and attribute correlation [17].

3.4.3 SageDB

SageDB [25] is a fully learned database system that extends ML techniques beyond indexing to encompass query optimization and storage layout management. It integrates learned models at multiple levels to dynamically optimize query execution plans and data organization. By adapting to workload patterns, SageDB enhances query efficiency and reduces storage overhead. However, its dependency on machine learning introduces computational overhead, thus making it less suitable for rapidly evolving workloads that require frequent retraining.

3.4.4 LiLIS

LiLIS (Lightweight Distributed Learned Index for Spatial Data) [26] extends learned indexing to distributed spatial systems. Unlike centralized structures such as R-Trees and Quadrant Trees (QuadTrees)[27], LiLIS is designed for scalability and robustness in large-scale, skewed workloads. It adopts a two-level architecture: a global K-Means clustering layer paired with a learned routing model for partitioning, and local Piecewise Geometric Models (PGMs) indexes for efficient range queries within each partition. This architecture allows LiLIS to scale across machines while maintaining high performance and memory efficiency. Experiments report up to 3.6 times faster queries and nearly 10 times lower memory usage compared to state-of-the-art spatial indexes, making it a compelling solution for distributed environments.

These learned MD techniques differ from traditional solutions by leveraging machine learning to construct data-driven and adaptive index structures, thereby improving query performance. Each model presents unique trade-offs in terms of adaptability, computational overhead, and efficiency that help demonstrate both the potential and the limitations of learned indexes in handling dynamic and high-dimensional data. Together, these models reflect the growing body of research and innovation in learned indexing, which indicates its importance and motivates increased activities and interests in this emerging and evolving research in database systems.

3.5 Conclusion

Learned indexes provided opportunities to shift from traditional rule-based structures to adaptive, data-driven models. While techniques like RMI have shown desired performance

for one-dimensional data, extending them to multi-dimensional workloads introduced challenges in accuracy, scalability, and adaptability.

Flood, the primary focus of this thesis, addresses these challenges by combining grid-based partitioning with learning-based optimization, and yields promising results for multi-dimensional range queries. These results motivate further investigation into Flood's design to better understand its individual components and identify opportunities for improvement.

To this end, we followed a modular approach to design and implement Flood from scratch, analyze its architecture, and evaluate the effectiveness of its component modules.

Chapter 4 presents both the original Flood architecture and our modular design and implementation, detailing the query execution pipeline, partitioning strategies, and layout optimization techniques. This thesis highlights Flood's core design while introducing configurable components and alternative strategies developed in this work. Together, these elements enable systematic evaluation of our Flood as a whole and its component modules individually. Our study lays the foundation for more adaptable and efficient learned indexing systems.

Chapter 4

The Flood Framework: Revisited

This chapter details the architectural design of the Flood indexing framework, our revised prototype system of Flood, and the methodology followed to evaluate and improve its performance. While we follow the high-level principles introduced by Nathan et al. [1], the prototype system we developed was built from scratch following a modular and extensible design methodology. This provided flexibility that, in turn, enabled us to isolate core components, analyze their performance under varying workloads, and identify opportunities to enhance some aspects compared to the original Flood framework.

Building on the foundations introduced in Chapters 2 and 3, in what follows, we first review the overall architecture of Flood, including grid construction, data reordering, and the cell table used for query processing. We then describe the execution pipeline we adopted to implement range queries, covering the projection, refinement, and scan stages. In this context, we use the terms *dimension*, *column*, and *attribute* interchangeably to refer to the schema of the dataset table. A *multi-dimensional* index simply refers to indexing based on multiple attributes.

Beyond faithful reconstruction of the Flood framework, this chapter introduces several original contributions aimed at improving performance and usability. These include:

- A redesigned cost estimation pipeline using synthetic data and ML models (Random Forest (RF), eXtreme Gradient Boosting (XGBoost), Neural Networks) to better predict query latency.
- A new layout optimization algorithm — *greedy coordinate descent with random restarts* — designed to overcome limitations of the gradient descent method used in the original design of the Flood framework.

- Adopting a modular approach in our implementation that enables configurable flattening methods, refinement strategies, and layout tuning options for detailed performance analysis.

The remainder of this chapter is organized as follows. Section 4.1 describes the layout creation process, including binning, cell assignment, and data ordering. We then describe the query processing pipeline. Section 4.2 presents the layout optimization formulation, our improvements to the cost model, and the optimization strategies we considered and evaluated. Finally, we conclude the chapter with a summary of key design choices that enable modular experimentation, each of which is further analyzed in Chapter 5.

Unless stated otherwise, the architectural structure described in this chapter is consistent with the design proposed by Nathan et al. [1], but is extended with implementation-level refinements and empirical improvements introduced in this thesis.

4.1 Flood Layout Architecture

As introduced in Chapter 2, Flood is a multi-dimensional, read-optimized, in-memory index designed to accelerate MD range queries on tabular data (relations). Unlike traditional index structures that rely on fixed partitioning of data, Flood dynamically adapts its layout based on both the data distribution and the observed query workload. This adaptability is a key feature of Flood. The architecture consists of two components that perform the following two main processes:

(1) An **offline layout construction component**, where the index is built based on selected layout parameters.

(2) An **online query execution component**, which uses an existing layout to process queries efficiently.

Figure 4.1 illustrates the Flood architecture and these two components. This section presents the overall architectural design of the Flood framework as introduced in the original paper. We will then present our implementation of Flood and evaluation method. In particular, we describe the steps for index creation, including grid construction, data reordering, and cell table generation. We then present the query execution pipeline and its implementation details.

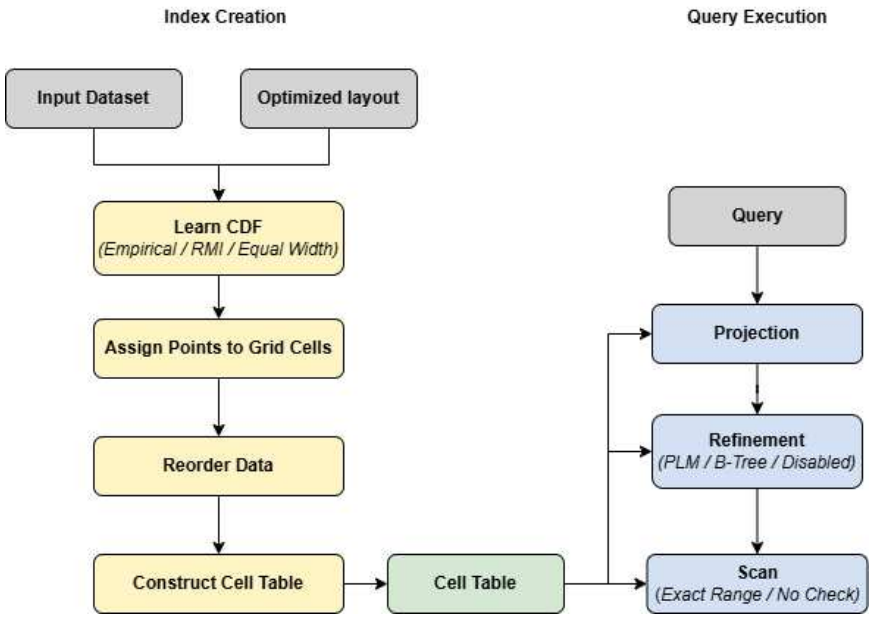

Figure 4.1: An overview of the Flood indexing architecture. The left side illustrates index creation: learning the CDF, assigning grid cells, reordering data, and constructing the cell table. The right side shows query execution, consisting of projection, optional refinement, and scanning.

4.1.1 Grid Creation

Flood's data layout relies on a multi-dimensional grid structure that organizes records into spatially defined cells, enabling efficient evaluation of range queries. Unlike traditional hierarchical indexing methods, Flood partitions the space explicitly using a fixed grid defined by layout parameters that control both dimensional ordering and bin granularity.

A layout in Flood is a configuration pair (O, C) defined using the following two vectors:

- A **layout vector** $O = [d_1, d_2, \ldots, d_d]$, which is a permutation of the d attributes of the input. The first $d-1$ attributes are used as the grid dimensions (i.e., indexing attributes), while the last one d_d serves as the *sort dimension*.
- A **bin count vector** $C = [c_1, c_2, \ldots, c_{d-1}]$, where each c_i specifies the number of bins (or columns) used to partition grid dimension d_i.

Given a layout configuration (O, C), Flood constructs a regular $(d-1)$-dimensional grid by dividing each dimension d_i into c_i bins, resulting in a total of $\prod_{i=1}^{d-1} c_i$ grid cells. Each record is assigned to a unique cell based on its values in the first $d-1$ dimensions. Within each cell, records are then sorted by the final attribute d_d, referred to as the *sort*

dimension. This intra-cell sorting is critical for efficient refinement and scan operations performed during query execution.

To impose a deterministic ordering across the entire dataset, Flood performs a depth-first traversal of the grid cells, based on the order of the dimension of the index attributes in O. Cells are visited in lexicographic order according to their grid coordinates, and all records within each cell are placed in the final data layout in the order determined by the sort attribute.

Two major benefits come from this layout design: it facilitates data locality by bringing together records that are likely to be queried together, as well as fine-grained pruning of irrelevant regions during query execution.

The actual boundaries used to define the bin partitions in each grid dimension are determined through a flattening process aimed at balancing the number of points per bin. The techniques used to compute these bin boundaries, including both empirical and learned methods, are discussed next in Section 4.1.2.

4.1.2 Deciding Bin Boundaries

One of the main challenges in constructing an efficient multi-dimensional index is determining how to divide the domain of each attribute into bins. The most straightforward strategy is to use *equal-width binning*, where each dimension is divided into some intervals of equal width. However, this strategy may not be desired for non-uniform or skewed data distributions. As a result, some grid cells may contain a disproportionately large number of points, while others may remain sparsely populated or even empty. Such imbalances degrade query performance by increasing scan overhead and causing load imbalance across cells.

To address this problem, Flood adopts a *flattening* strategy — also referred to as CDF-based binning, using which the number of points in the grid cells is about the same. This strategy is also known as *equi-depth binning* or *equal-frequency binning*, and has been widely used in prior work to improve query efficiency by ensuring a more balanced distribution of data in the index structure.

The key idea behind flattening is to construct bin boundaries that partition each dimension such that each bin contains approximately the same number of records. This balances the density of data across the grid cells and hence making the average search performance the same for any data item. In the context of Flood, this is especially important because

the index relies on tight alignment between data distribution and query selectivity to minimize the number of scanned points while maintaining a similar average number of scanned points across all possible query workloads.

Flattening in Flood is implemented using models that approximate the CDF for each attribute. The CDF of a dataset represents the probability that a value drawn from the data is less than or equal to a given threshold. By inverting the CDF, we can determine the data value corresponding to a given percentile and use this value to define bin boundaries that result in approximately equal-sized partitions.

Motivation for CDF-Based Flattening

Flood's query execution pipeline relies heavily on efficient *projection*, the process of mapping a query's predicate bounds onto the grid structure to identify candidate cells. If bin boundaries are unevenly spaced (as may be the case with equi-width binning), this projection step becomes more expensive. Determining the correct bin index for a query bound would require a binary search or similar lookup over the bin boundaries, which may lead to increased query processing overhead.

By modeling the CDF as a continuous function $F(v)$, Flood can directly compute the bin index that corresponds to a query bound v using the following formula:

$$\text{bin index} = \lfloor F(v) \times n \rfloor$$

where $F(v)$ is the estimated CDF value at point v, and n is the number of bins for the given dimension.

This way of flattening of CDF enables efficient, constant-time computation of bin indices during query projection, and eliminates the need for expensive lookups which in turn results in improved performance.

Flattening Methods

The original Flood framework proposed using RMI to approximate the CDF of each attribute and determine bin boundaries accordingly. This allowed fast, projection-friendly transformations while adapting to skewed or non-linear data distributions.

In our implementation, we retain the overall CDF-based flattening strategy due to its computational efficiency during query projection. However, instead of hard-coding RMI, we modularized the flattening step and made it "configurable", allowing us to experiment with and evaluate alternative strategies and assess their impact on index performance.

We implemented and studied the impact of the following flattening strategies:

- **Equal-width binning**: The simplest method, where each attribute's domain is divided into intervals of equal size. While easy to compute, this method performs poorly on skewed datasets, often resulting in unbalanced grids with dense and sparse regions.
- **Empirical CDF**: A non-parametric method that sorts attribute values and defines bin boundaries based on percentiles. While straightforward and easy to implement, this method requires sorting and cumulative counting, which increases index creation time. It slightly reduces refinement time by eliminating learned model traversal during query processing.
- **RMI**: This was the original approach proposed in Flood. It uses a hierarchical learned model to approximate the inverse CDF. Our implementation uses a three-stage structure: one model at the root, $\sqrt{n}$ models at the second level, and n linear models at the leaf level. The parameter n is tuned based on the input dataset size and skew to balance accuracy and overhead.

The results of our empirical evaluation showed that using flattening methods significantly improved query performance compared to using no flattening (Equal-width binning). However, in the refinement step, empirical CDF offers minor benefits because it constitutes only a fraction of the overall query time. In contrast, the presence of any flattening technique—empirical or learned—helps achieve balanced grid partitions and speeds up query execution. Detailed experimental comparisons of these methods, including trade-offs in index creation time and query execution time, are presented in Chapter 5.

4.1.3 Data Sorting and Cell Table Construction

Once the grid structure has been defined and each data point has been assigned to a corresponding grid cell, the next step is to finalize the physical data layout in memory. Flood achieves this through a two-level ordering process described below that improves data locality and facilitates efficient query processing.

First, a global ordering of grid cells is imposed using a *depth-first traversal* based on the attribute sequence specified in the layout vector O. Cells are visited in lexicographic order of their grid coordinates: starting with the first grid dimension d_1, ties are broken using

the next dimension d_2, and so on. This traversal strategy ensures a predictable, compact arrangement of neighboring cells in memory, which is critical for efficient cache management during query evaluation.

Second, within each grid cell, all data points are sorted by their value in the *sort dimension* d_d. This intra-cell ordering plays a crucial role during the refinement and scan phases of query execution. In particular, when a query includes a range predicate on the sort dimension, Flood can leverage this order to prune irrelevant points using binary search or early termination, depending on the refinement strategy. Thus, this step acts as an optimization layer, which, in particular, is critical for online query processing by reducing the filtering overhead.

The resulting physical layout can be thought of as a serialized array of records, where points in the same grid cell are stored contiguously and are already ordered on the sort attribute. An illustration of this serialization strategy for a two-dimensional example is shown in Figure 4.2.

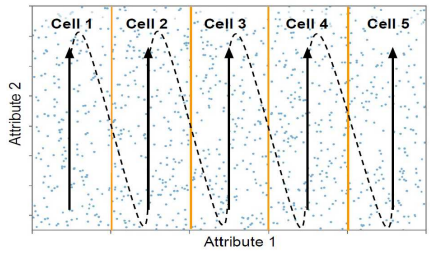

Figure 4.2: Adapted from [1], this figure illustrates a basic layout in Flood with two dimensions. First, the points are assigned to grid cells along Attribute 1, and then sorted by Attribute 2 within each cell. The arrows indicate the final linear serialization order.

Following the layout construction, a metadata structure known as the *cell table* is generated. This table maintains, for each grid cell, a pointer to its starting position and ending position in the sorted data array. In practice, this table enables Flood to quickly retrieve the relevant segments of the data for a given range query without having to scan the entire dataset. To demonstrate the flattening and cell assignment process in Flood, we consider a toy dataset with three attributes: `Price`, `Discount`, and `Rating`, where, `Price` and `Discount` are used as grid dimensions, and `Rating` serves as the sort dimension.

Table 4.1: Toy dataset with 3 attributes

ID	Price	Discount	Rating
1	100	10	4.2
2	150	5	3.9
3	120	15	4.7
4	200	20	2.5
5	180	25	3.0
6	160	10	4.8
7	170	5	3.3
8	140	20	4.1

Considering the layout vector $O = [\texttt{Price}, \texttt{Discount}, \texttt{Rating}]$, where the first two attributes define the grid dimensions and the third attribute (`Rating`) is used as the sort dimension, each grid dimension is independently flattened into 2 bins using the empirical CDF.

Empirical CDF Calculation: In Flood, we use an *empirical CDF* to determine bin boundaries during flattening. The empirical CDF for a value v is defined as:

$$F(v) = \frac{1}{N} \sum_{i=1}^{N} \mathbf{1}_{x_i \leq v}$$

where

- $N = 8$ is the number of records in our example,
- x_i are the attribute values,
- $\mathbf{1}_{x_i \leq v}$ is the indicator function.

For example, to flatten `Price`, we sort the values:

$$[100,\ 120,\ 140,\ 150,\ 160,\ 170,\ 180,\ 200]$$

To split into 2 bins, we find v such that $F(v) = 0.5$. Since the values for items 4 of 8 values are at most 150, the bin boundary would be:

$$v = 150 \quad \text{(i.e., Price Bin 0: } \leq 150,\ \text{Price Bin 1: } > 150)$$

Similarly, for `Discount`, the bin boundaries are determined as follows:

$$[5,\ 5,\ 10,\ 10,\ 15,\ 20,\ 20,\ 25] \Rightarrow F(10) = \frac{4}{8} = 0.5 \Rightarrow \text{Bin boundary at } 10$$

This results in:

- **Price Bins:** Bin $0 \leq 150$, Bin $1 > 150$
- **Discount Bins:** Bin $0 \leq 10$, Bin $1 > 10$

Grid Cell Assignment: In the next step, each record is mapped to a grid cell (i, j) based on its bin indices:

Table 4.2: Grid cell assignment based on binning

ID	`Price Bin`	`Discount Bin`	**Cell (i,j)**	`Rating`
1	0	0	(0,0)	4.2
2	0	0	(0,0)	3.9
3	0	1	(0,1)	4.7
4	1	1	(1,1)	2.5
5	1	1	(1,1)	3.0
6	1	0	(1,0)	4.8
7	1	0	(1,0)	3.3
8	0	1	(0,1)	4.1

Intra-Cell Sorting and Final Record Order: Each cell is sorted by the `Rating` attribute. Then, cells are arranged in lexicographic order: (0,0), (0,1), (1,0), (1,1). The final global order becomes:

Table 4.3: Final Record Order After Grid Assignment and Intra-Cell Sorting

Final Position	Record ID
0	2
1	1
2	8
3	3
4	7
5	6
6	4
7	5

Cell Table (as stored in Flood): Instead of storing full index ranges, Flood stores only the **starting offset** of each non-empty cell in the final layout. This allows fast lookup and avoids storing duplicate boundaries.

Table 4.4: Flood-style cell table (starting offset per non-empty cell)

Grid Cell (i,j)	Start Offset
(0,0)	0
(0,1)	2
(1,0)	4
(1,1)	6

This layout allows efficient query execution. The cell table is used to fetch the start offset for each matching cell, enabling direct access to relevant records during projection. This cell table is a compact, array-like data structure indexed by cell IDs, which—due to the deterministic reordering of data—correspond directly to physical addresses in memory. This design supports constant-time lookup of cell boundaries and is essential for efficient implementation of the projection and refinement steps described in Section 4.1.4.

4.1.4 Query Execution

Once the index is built and the dataset is laid out and organized according to the grid structure, Flood is ready to handle range queries. A typical query provides filters over one or more attributes, and Flood's task is to retrieve the subset of data points that fall within the specified multi-dimensional range, also known as a *query hyper-cube*.

Flood processes queries through three main stages:

- **Projection:** This step identifies the grid cells that intersect the filter conditions in a query. For filters on indexed dimensions, it computes a multi-dimensional range (a hyper-rectangle in grid space) and uses the *cell table* to retrieve the corresponding actual physical ranges in the dataset.
- **Refinement:** If the query includes a filter on the *sort dimension* (the last argument in the layout), refinement can further narrow down the candidate data points within each cell. In our implementation, we use a Piecewise Linear Model (PLM) to predict the value range corresponding to the sort predicate, leveraging B-Tree lookups for fast access. If there is no sort filter present or the refinement module is disabled, this step is skipped.
- **Scan:** This is the final step, which linearly scans the candidate records from the previous steps and checks whether each record satisfies all query predicates. In general, scan dominates the total query time—especially for large and/or dense cells—making it the major bottleneck in Flood's performance.

Each of these steps helps narrow the search space, but their relative impact varies depending on the query workload and layout configuration. As shown by our experiment results in Chapter 5, the scan step typically accounts for the majority of total query execution time.

Motivating Example: How Layout Affects Query Performance

To better understand how grid layout impacts query execution efficiency, we present an illustrative example comparing two different layout configurations for the same dataset and query workload.

Dataset Overview

The dataset consists of 1000 sales records with three numeric attributes:

- **Price** (index attribute): Product price, approximately in the range 30 to 260.
- **Discount** (index attribute): Discount applied, ranging from 0 to 30.
- **Rating** (sort dimension): Product rating in the range [1, 5].

In both layouts, `Price` and `Discount` serve as the grid dimensions, while `Rating` is used as the sort attribute to enable refinement. The data distribution is moderately skewed and noisy to reflect realistic sales patterns.

Query Specification

We define a sample query with the following predicates:

$$60 < \text{Price} < 140, \quad 6 < \text{Discount} < 10, \quad 3 < \text{Rating} < 5$$

This query selects a specific price and discount range along with a high rating preference. It is used to demonstrate how layout choice affects the number of scanned and refined records.

Layout Comparison

We compare two layouts:

- **Layout 1:** $L = [2, 4]$: 2 bins for `Price`, 4 for `Discount`
- **Layout 2:** $L = [4, 2]$: 4 bins for `Price`, 2 for `Discount`

Figures 4.3 and 4.4 visualize these layouts and show the number of data points within each intersecting cell along with the red dashed query rectangle.

Results Summary:

- **Layout 1 (L = [2,4]):**
  - 4 intersecting cells
  - Total candidate points: 446
  - Points after refinement: 349
  - Points scanned: 349
- **Layout 2 (L = [4,2]):**
  - 6 intersecting cells
  - Total candidate points: 758
  - Points after refinement: 580

- Points scanned: 580

Discussion:

Although both layouts cover the same query range, Layout 1 results in fewer intersecting cells and fewer scanned points. This is because its grid structure aligns better with the shape of the query region: the narrower binning on `Discount` and coarser binning on `Price` lead to tighter coverage. In contrast, Layout 2's finer binning on `Price` and coarse bins on `Discount` cause the query to span more cells vertically, increasing projection cost and scan overhead.

This example highlights a core motivation behind Flood's layout learning: the right grid configuration can significantly reduce the number of candidate records. Poor alignment between query shape and grid boundaries leads to redundant cell accesses and excessive scan operations.

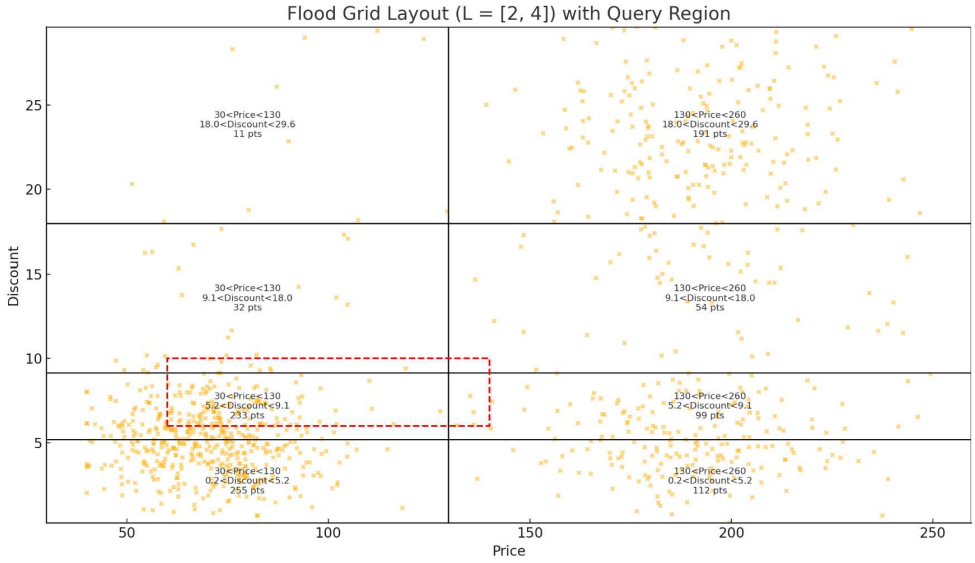

Figure 4.3: Flood Grid Layout with $L = [2, 4]$ and query region overlay.

Although both layouts cover the same query range, Layout 1 results in fewer intersecting cells and fewer scanned points. This is because its grid structure aligns better with the shape of the query region: the narrower binning on `Discount` and coarser binning on `Price` lead to tighter coverage. In contrast, Layout 2's finer binning on `Price` and coarse bins on `Discount` cause the query to span more cells vertically, increasing projection cost and scan overhead.

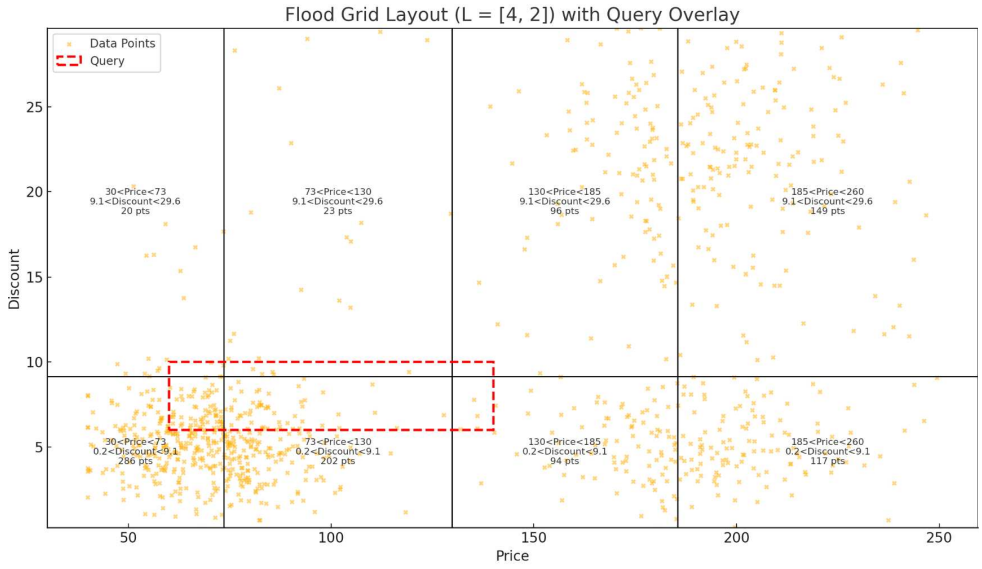

Figure 4.4: Flood Grid Layout with $L = [4, 2]$ and query region overlay.

This example highlights a core motivation behind Flood's layout learning: the right grid configuration can significantly reduce the number of candidate records, making query execution more efficient. Poor alignment between the query shape and grid boundaries leads to redundant cell accesses and excessive scan operations. This is precisely the challenge addressed by the layout optimization process, which we explore in the next section.

4.2 Layout Optimization

A key feature that differentiates Flood from traditional multi-dimensional indexes is that it is a *learned index*. Rather than relying on fixed heuristics or rule-based partitioning strategies, Flood adapts to the observed patterns in both the data distribution and the query workload. It "learns" an effective layout—how the data should be partitioned across dimensions—to optimize query performance for the given scenario.

So far, we explained how the framework executes queries given a layout. The learning component lies in configuring that layout. In other words, given a dataset and a query workload, Flood searches for a layout that is tailored to those characteristics in order to minimize query cost.

4.2.1 Problem Formulation

We define a layout L for a d-dimensional dataset D as:

$$L = (O, \{c_i\}_{0 \leq i < d-1})$$

Where O is an ordering of the d attributes (with the last attribute used as the *sort dimension*), and $\{c_i\}$ is the number of bins assigned to each of the remaining $d - 1$ grid dimensions.

The objective is to minimize the average query execution time over a representative workload Q:

$$\min_{L} \; E_{q \in Q}[\text{Time}(D, q, L)]$$

Here, D is the dataset, Q is a set of representative queries, and L is the layout configuration to be learned.

Evaluating every candidate layout by building its full index and executing all queries is prohibitively expensive. Therefore, Flood employs a learned cost model to estimate the execution time for each layout configuration efficiently.

4.2.2 Cost Model

The Flood framework relies on a machine learning-based cost model to guide layout optimization. Rather than evaluating every candidate layout by physically building the index and executing queries (a prohibitively expensive process), Flood uses this model to predict query latency based on statistics extracted from the data, the layout, and the workload. This prediction enables fast and scalable search across layout configurations.

The core idea here is to model query execution time as:

$$\text{Time}(D, q, L) = w_p N_c + w_r N_c + w_s N_s$$

Here, N_c denotes the number of grid cells whose ranges intersect with the query predicates. Each cell contains data points that may match the query. After projection, the refinement step(if applicable) filters candidates within each cell based on the sort dimension. Finally, N_s is the number of data points examined during the scan step against the full query conditions. The weights w_p, w_r, and w_s represent the cost of projection per cell, refinement per cell, and scanning per point, respectively. These weights are learned from training data.

In the original Flood design, these weights were learned using regression models (typically Random Forests) trained on synthetic workloads. The values of N_c and N_s were estimated using a sampling-and-scaling strategy, where queries were simulated on a small subset of the data and the observed statistics were scaled to approximate full-data behavior.

The training pipeline consists of the following components:

- **Synthetic Dataset Generation:** Random datasets are generated with various distributions (uniform, normal, skewed) and dimensionality to simulate a wide variety of scenarios.
- **Random Query Generator:** A large set of queries (typically 100 per layout) is generated using quantile-based sampling across different index dimensions.
- **Layout Generator:** Layouts are sampled by randomizing dimension orderings and selecting bin counts from reasonable ranges.
- **Feature Extraction:** For each query-layout pair, they extract features that describe the layout (e.g., grid size, cell density), the query (e.g., selectivity, filtered dimensions), and the data distribution (e.g., skew, sparsity).
- **Model Training:** In our proposed pipeline, instead of using Random Forest regressors for the weights w_p, w_r, and w_s, we used XGBoost models for better accuracy and generalization. Additionally, we trained neural networks to directly predict N_c and N_s from the layout and query features, hence eliminating the need for sampling-based estimation.

These changes improved the modeling quality by providing more accurate and robust cost prediction during layout optimization, as shown by our evaluation results presented in Chapter 5.

4.2.3 Optimization Procedure

The goal of the layout optimization process in Flood is to identify a configuration that minimizes the expected query execution cost for a given dataset and workload. Each layout is defined by a permutation of attributes O and a corresponding bin count vector $C = [c_0, c_1, \ldots, c_{d-1}]$, where d is the total number of attributes of the data points and the last attribute c_d is treated as the sort dimension. The layout determines how the data is partitioned into grid cells and also how query predicates are projected and refined.

Challenges in Layout Optimization. Layout tuning in Flood presents several technical challenges due to the structure of the search space. The problem is inherently discrete and combinatorial, requiring integer bin counts and factorial permutations of dimensions. Additionally, the cost function is defined by non-differentiable models such as decision trees or ensembles, which introduce irregularities and discontinuities into the optimization landscape. Key challenges include:

- **Integer Constraints:** Bin counts must be integers, preventing the use of standard gradient-based optimizers without discretization. Any relaxation to continuous space introduces rounding effects that risk convergence [28].
- **Non-Differentiability:** The cost surface is shaped by decision-tree regressors (e.g., Random Forests, XGBoost), which are inherently non-smooth and produce step-wise changes in output, which would further limit the utility of gradient-based methods.
- **Volatile Cost Landscape:** Small modifications in layout can result in large, abrupt shifts in estimated cost due to threshold behavior in tree models, making the search prone to oscillation and convergence to poor local minima.

Explored Strategy: Gradient Descent with Finite Differences. To replicate the optimization procedure described in the original Flood paper, we implemented a finite-difference-based version of gradient descent. Since true gradients are unavailable, we estimated them by perturbing each bin count c_i by ±1 and measuring the change in predicted cost. The update direction was chosen to be opposite to the sign of the gradient, as used in standard descent methods.

We then considered two variants: (1) simultaneous updates to all dimensions in each iteration, and (2) evaluating all dimensions but updating only the one that yields the best improvement. The first variant (1) suffered from oscillations, as interactions between dimensions produced conflicting changes in the layout. The second variant (2), greedy per-dimension updates, produced smoother convergence and improved cost early in the search. However, it was computationally expensive, requiring multiple evaluations per iteration, and often settled in local optima. Its deterministic nature and limited step size further restricted exploration of the layout space.

Adopted Solution: Greedy Coordinate Descent with Random Restarts. Given the limitations of gradient descent in discrete, non-differentiable spaces, we adopted a greedy coordinate descent strategy with random restarts. This method is conceptually well-suited to the

nature of the problem, as it balances local search efficiency with global exploration [29]. At each iteration, a single coordinate c_i is randomly selected, and the algorithm evaluates cost changes for incrementing or decrementing its value by 1 (bounded within allowed limits). If the change leads to a cost reduction, it is accepted; otherwise, the configuration remains unchanged.

To prevent the optimizer from getting stuck in local minima, the process is periodically restarted from a new, randomly initialized layout. These restarts ensure diverse coverage of the search space and improve the likelihood of finding a better overall configuration. Compared to global methods, this approach scales better, does not require smoothness or differentiability, and is naturally compatible with integer-valued parameters [30, 31].

Conceptually, greedy coordinate descent with restarts avoids the pitfalls of premature convergence by (1) limiting updates to one axis at a time, thereby simplifying the optimization step and reducing volatility, and (2) incorporating randomness to reinitialize the search and explore alternative solutions. Furthermore, the method is straightforward to implement, interpretable, and more flexible to incorporate custom bounds or heuristics.

Implementation Details. Our implementation of greedy coordinate descent uses a machine-learned cost model to estimate the parameters: projection (w_p), refinement (w_r), and scan (w_s) times, with a neural network predicting N_c and N_s per query. At each iteration, a random dimension is selected and its bin count is adjusted by a discrete step size if it leads to a lower total estimated cost. To improve robustness, we consider "informed" initializations, where each restart begins with a layout sampled from attribute-specific bounds. That is, for each c_i, values are drawn from $[c_i^{\min}, c_i^{\max}]$ based on the attribute's cardinality and selectivity profile. The step size starts at 16 and is halved every M iterations without improvement. We chose 16 as the initial step size based on empirical tuning, as it offered a good trade-off between early-stage exploration and convergence speed. This enables the algorithm to begin with broad exploration and gradually refine around promising configurations. This adaptive behavior helps the optimizer explore more of the layout space in less time. Early termination is triggered after repeated stagnation. Although global optimality is not guaranteed, we are convinced by our experiments that this approach consistently yields lower-cost, better-balanced layouts in practice, making it a practical and effective strategy for tuning Flood index layouts. The steps of our layout optimization are presented in the following algorithm:

Algorithm Overview. The optimization routine is presented below.
Inputs: D (data points), Q (query workload), T (cost model), R (number of restarts).

```
1:  procedure FindOptimalLayout(D, Q, T, R)
2:      for r ← 1 to R do
3:          C ← RandomInitialCounts()
4:          O ← SampleDimensionOrder()
5:          L ← (O, C)
6:          cost ← T(L, D, Q)
7:          repeat
8:              improved ← False
9:              for i ← 0 to d − 2 do
10:                 for δ ∈ {−1, +1} do
11:                     C′[i] ← max(2, C[i] + δ)
12:                     L′ ← (O, C′)
13:                     new_cost ← T(L′, D, Q)
14:                     if new_cost < cost then
15:                         C ← C′, cost ← new_cost
16:                         improved ← True
17:                     end if
18:                 end for
19:             end for
20:         until not improved
21:         Store L if best so far
22:     end for
23:     return Best layout found
24: end procedure
```

Figure 4.5 illustrates this process.

The above algorithm and the flowchart shown in Figure 4.1 together illustrate the layout optimization process used in Flood. The algorithm first describes how random restarts and greedy coordinate descent are combined to iteratively search for a cost-efficient layout. At each iteration, a new layout $L = (O, C)$ is evaluated using a machine-learned cost model, and adjustments to the bin counts are accepted only if they reduce the predicted cost. After convergence, the best layout found is retained. The flowchart in Figure 4.1 visualizes this procedure, showing how the dataset and query workload are used to evaluate candidate layouts and guide the search toward more efficient configurations.

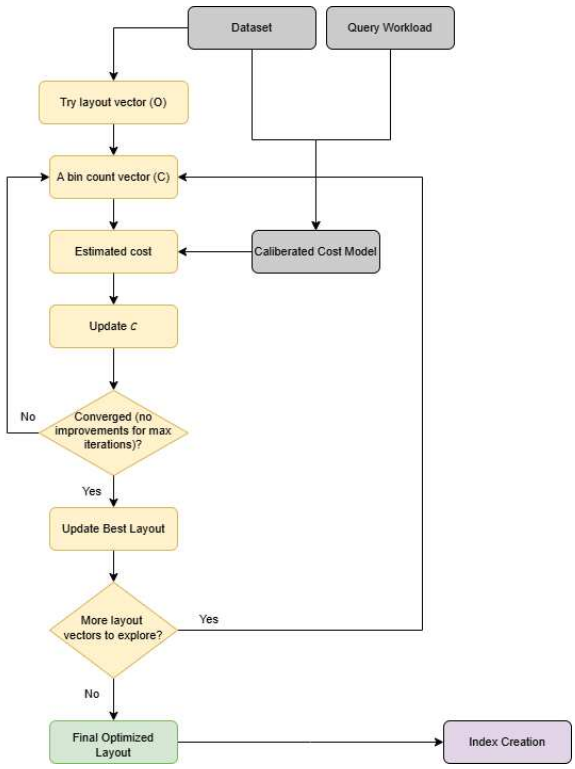

Figure 4.5: Flowchart of layout optimization using greedy coordinate descent with random restarts. Each candidate layout is evaluated using a trained cost model to guide the search.

4.2.4 Systematic Experimental Evaluation

A key aspect of our implementation of learned index is its modular design, which enables systematic, flexible experimentation with different indexing strategies. Each core component—flattening, refinement, and scan steps—can be independently configured, allowing us to isolate and evaluate the performance impact of each design decision under controlled conditions.

The following parameters are fully configurable:

- **Refinement Strategy:** The implementation framework can use either:
  - *Peacewise Linear Model (PLM)* refinement, which approximates the sort dimension's CDF using linear segments and leverages a cache-optimized B-Tree for fast segment lookup. The PLM ensures bounded approximation error and enables fast refinement, particularly effective when queries include a filter on the sort dimension.

- *Binary Search Tree (BST)* refinement, for which we use a traditional search tree over sort values to locate relevant ranges. While being accurate, we found it to be slower than PLM for large cells.
- *No Refinement*, where the framework skips this step entirely and scans the full cell range as-is, it is useful for benchmarking worst-case scenarios.

- **PLM B-Tree Order:** When using PLM, the B-Tree order (i.e., node fanout) used for segment indexing can be tuned to study memory utilization trade-off with lookup speed.
- **Flattening Technique:** The framework allows users to choose between:
  - *Empirical CDF:* Based on directly computing percentiles from sorted values.
  - *RMI:* A learned approximation of the CDF that supports faster projection and finer control in skewed distributions.
- **Exact Range Checking:** Through this option, users can determine whether the runtime attempts to skip refinement and scan altogether for cells that are fully covered by a query. While sometimes useful for selective queries, this option was found to degrade performance in our experiments and is disabled by default.

The modular approach in our design enables comprehensive performance evaluation. In the next chapter, we report the results of our experiments conducted across multiple datasets and query workloads. These experiments were designed to identify bottlenecks and scalability, and explore the memory/execution trade-offs involved in each component's configuration. The flexibility provided by our modular framework was crucial for a systematic study of different strategies and their individual and combined impacts on the overall efficiency of the learned index implemented by Flood.

4.3 Summary

This chapter described our methodology used to implement, evaluate, and improve the Flood indexing framework. We began by outlining the original Flood's architectural components, including grid construction, flattening, and query execution. While grounded in the original design, our implementation introduced several enhancements to support modular experimentation and detailed performance analysis.

We proposed an improved cost modeling pipeline that incorporates neural estimators for Number of Intersecting Cells (N_c) and Number of Scanned records (N_s), and XGBoost regressors for latency component weights, resulting in better predictive accuracy and generalization. To address the shortcomings of Flood's original layout tuning strategy, we replaced gradient descent with greedy coordinate descent and random restarts, which provided a more stable and effective optimization process.

Finally, we developed a prototype system that is fully modular, enabling configurable flattening methods, refinement strategies, and scan options. This development approach allowed us to isolate the performance impact of individual components and laid the groundwork for the empirical evaluation presented in the next chapter.

In Chapter 5, we validate the correctness of our implementation, compare Flood's performance under diverse data and workloads, and analyze how each component contributes to overall efficiency.

Chapter 5

Experiments and Results

5.1 Overview

This chapter presents a comprehensive evaluation of the Flood indexing framework. The goals are threefold: (1) to validate the correctness and evaluate the performance of the implementation of the Flood framework we developed, (2) to analyze the impact of key design choices such as flattening and refinement components, and (3) to assess the effectiveness of our proposed improvements in cost modeling and layout optimization algorithms.

We begin by outlining the experimental setup, followed by validation tests to ensure correctness. The remainder of the chapter presents targeted performance analyses, comparative studies, and ablation experiments across various datasets and query workloads.

5.1.1 Experimental Setup

To evaluate the performance of Flood and compare it with other multi-dimensional indexing techniques, we set up a series of controlled experiments. This included the computing environment, the datasets used, the types of queries tested, and the performance metrics measured.

Environment. All experiments were run on a personal computer with Windows 10 (64-bit), a 12th Gen Intel® Core™ i7-1260P processor (2.10GHz), and 16GB of RAM. The implementation was written in Python 3 and executed in Jupyter Notebook. To ensure a fair comparison across all optimization algorithms and indexing methods, all tests were performed using a single CPU thread.

Datasets. We used two classes of datasets:

- **TPC-H Lineitem:** A standard benchmark dataset [32], with about 6 million records and a total size of roughly 2.7 GB.
- **Synthetic Dataset:** A custom dataset with 10 million records and 7 numeric columns. The data includes uniform, normal, and exponential distributions, as well as skewed values, outliers, and some correlated features to test different indexing scenarios.

Query Workloads. For each dataset, we created 100 range queries. The number of filtered dimensions in each query was randomly chosen between 1 and d, where d is the number of indexed columns. For the synthetic and retail datasets, each filter used a random selectivity between 1% and 10%.

For TPC-H datasets, we also created queries based on common filters in the official benchmark, such as ranges on dates, quantities, and prices. These help simulate realistic decision-support queries.

5.2 Implementation Validation

To ensure the correctness of our Flood index implementation, we compare its query results against two trusted baselines: a brute-force scan and equivalent SQL queries executed over the same dataset using SQLite. We load the dataset into an in-memory SQLite database and generate equivalent SQL query expressions based on the filter ranges. For each test case, we execute both the Flood query and its SQL counterpart, sort and cast the results to ensure data type alignment, and verify exact equality using Pandas' `assert_frame_equal` test function. This function is used to verify the equality and correctness of codes that generate or manipulate data. The test results confirm that the Flood index returns the same results as the ground truth for a wide range of multi-dimensional queries.

5.3 Query Execution Time and Component Impacts

This section presents an in-depth, comprehensive empirical analysis of Flood's internal query processing performance using benchmark datasets ranging from 100K to 6M records. We decompose total query time into its key components: projection, refinement, scan, and exact range filtering, and examine how each component is influenced by grid resolution (i.e., the total number of grid cells), data distribution, query selectivity, the number of cells examined (`Nc`), and the number of records scanned(`Ns`). Our findings show that query

performance is shaped by complex interactions between layout configuration and workload characteristics, justifying Flood's learning-based design.

5.3.1 Query Time Breakdown

To better understand how these components interact, Figure 5.1 presents a stacked bar chart of total query time broken down into projection, refinement, and scan phases, across different grid sizes. Regardless of layout granularity, scan remains the dominant cost, often exceeding 90% of the total time. While refinement and projection overheads increase with grid size, they remain relatively small.

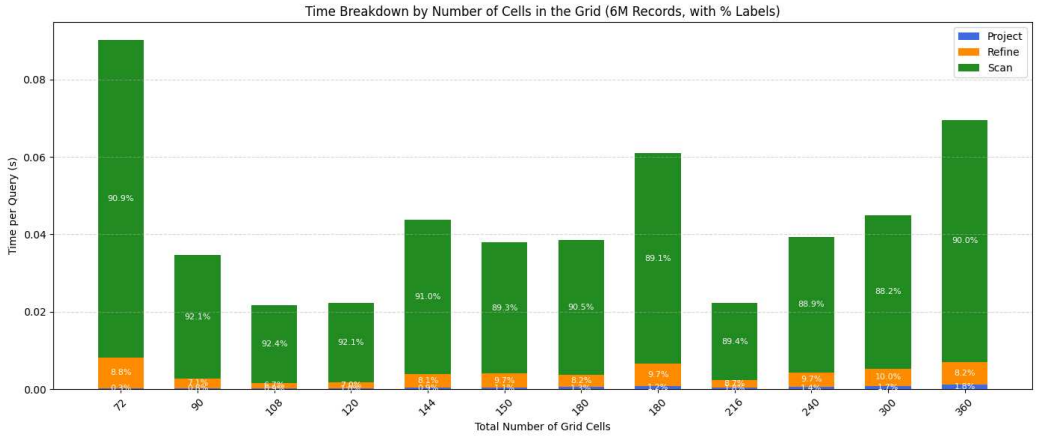

Figure 5.1: Time breakdown per query by total number of grid cells in the layout. Green: scan time; orange: refinement; blue: projection. As can be seen, Scan dominates across all configurations. Percentage labels represent each component's share of total time.

Across all configurations and datasets, the scan phase consistently appears as the principal bottleneck in Flood's query processing pipeline. In most queries, scan accounts for around **90%** of the total execution time, confirming that reducing the number of scanned records (`Ns`) is essential for performance optimization.

Figure 5.2 shows that scan time increases approximately linearly with `Ns` on a log-log scale. Each point in the figure represents a query, with color encoding average selectivity and bubble size indicating result length. As query selectivity ratios increase, the number of scanned records and the corresponding scan latency increase as well. This reinforces that `Ns` is the most critical factor influencing total query time.

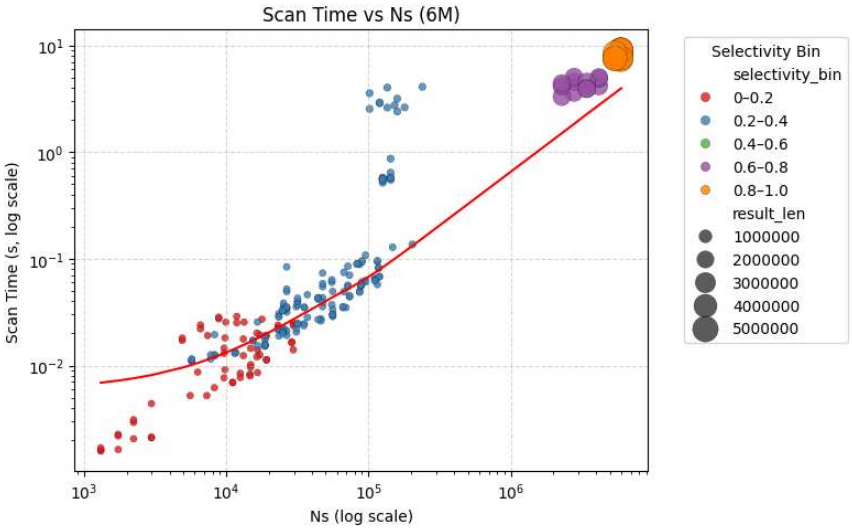

Figure 5.2: Scan Time vs. Number of Scanned Records (Ns) on the 6M dataset. Axes are log-scaled. Bubble size reflects result length; color represents query selectivity. The red curve shows a trend line capturing the overall correlation.

In addition to Ns, we investigate how projection and refinement times vary with the number of intersecting cells per query (Nc). As shown in Figure 5.3, projection time increases moderately with Nc, while refinement time rises sharply, especially for queries with high selectivity ratio. Since Nc is influenced by the total number of grid cells, this highlights a trade-off: finer grid resolutions improve filtering accuracy but introduce higher overhead in both projection and refinement phases.

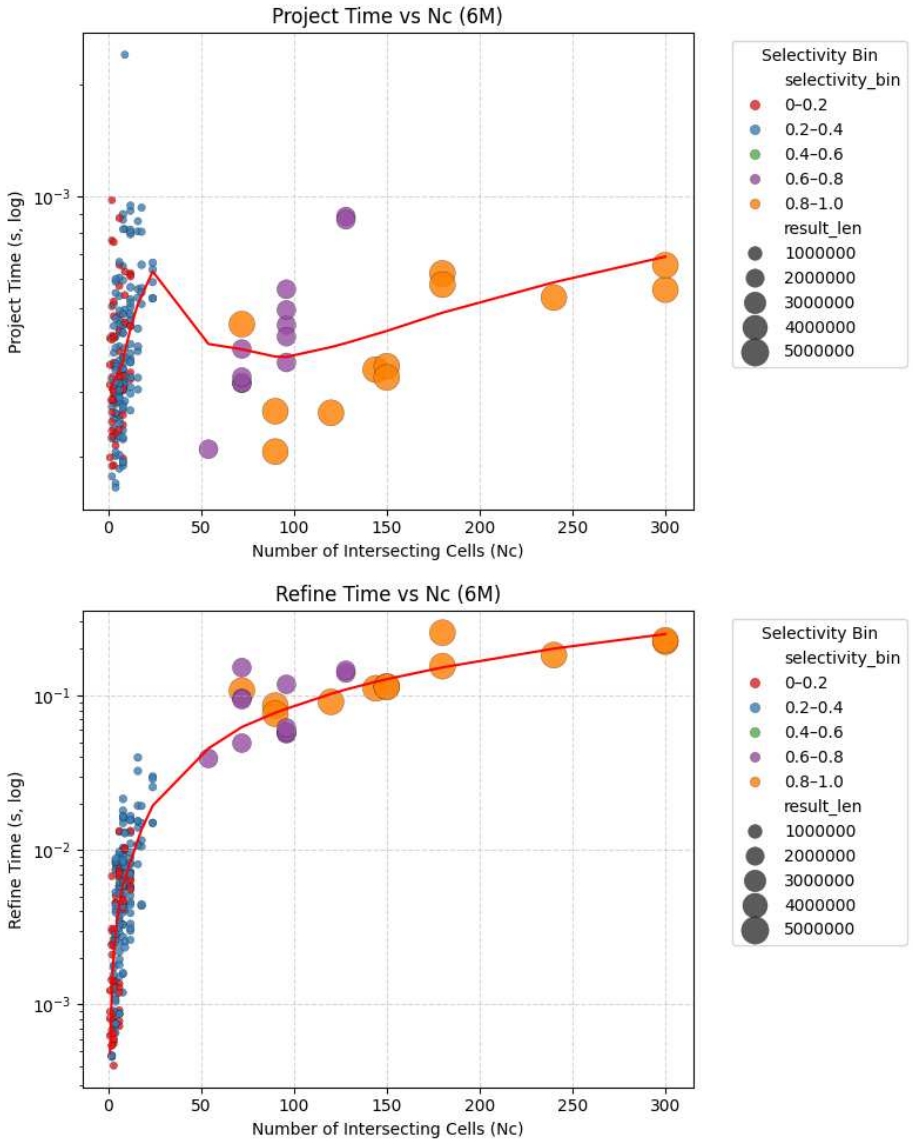

Figure 5.3: Top: Projection time vs. number of intersecting cells (Nc). Bottom: Refinement time vs. Nc. Each point represents a query; bubble size reflects result length, and color encodes average selectivity. Refinement time increases sharply with both selectivity and the number of intersecting cells.

The total number of grid cells in the layout plays a pivotal role in determining this balance. As grid granularity increases and the average number of records per cell decreases, each cell covers a smaller range of values, which improves projection precision and increases the chances of applying refinement or exact skipping. However, this also raises Nc, leading to higher overhead in earlier stages.

We observed that query performance peaked, for example, on the TPC-H dataset with a standard query workload, when each cell contained approximately **1,000 to 2,000** records. Below this threshold, the overhead of visiting many small cells dominated. Above it, the scan volume increased due to coarse cells containing too many candidate points. However, this sweet spot varies across datasets and workloads, especially depending on average query selectivity. This further emphasizes the need for layout tuning as a formal optimization problem, rather than relying on fixed heuristics.

These observations reinforce Flood's core design principle: query performance is shaped by complex, interdependent factors such as scan volume, grid resolution, and query selectivity, which cannot be effectively captured through fixed rules. Minimizing the number of scanned records (`Ns`) is especially important, as scan time constitutes the most considerable portion of query latency. Increasing the number of grid cells can help reduce `Ns` by narrowing candidate regions. However, it also increases projection and refinement overhead by raising the number of intersecting cells (`Nc`). Effective layout design requires a more careful balance: cells should be fine-grained enough to reduce scan volume but not so numerous that they introduce overhead in earlier stages. These trade-offs highlight the value of data-driven, machine-learned layout tuning.

5.3.2 Refinement Shrinkage vs Cost Trade-off

Refinement is applied after projection and before the scan phase. It aims to reduce `Ns` by pruning records outside the query range on the sort dimension. Our experiments show that refinement is highly effective: on average, it reduced scan volume by 71.7% across both TPC-H and synthetic classes of datasets. However, its benefit is not uniform across the query space.

As shown in Figure 5.4, the effectiveness of the refinement step depends on the number of scanned records. In most cases, refinement significantly reduces scan time while adding only a small overhead. This trade-off is favorable and supports enabling the refinement step.

However, for `Ns` values of about 10,000 or more, refinement becomes less beneficial, indicating diminishing returns and hence increased cost. This is likely due to high cell density and limited filtering power when a large portion of data matches the query range.

These results suggest that refinement is more effective for queries with mid-sized result sets, large enough to benefit from pruning but not so large that refinement overhead outweighs the gain. This insight can be used to employ a dynamic strategy that enables or

disables refinement based on predicted `Ns`.

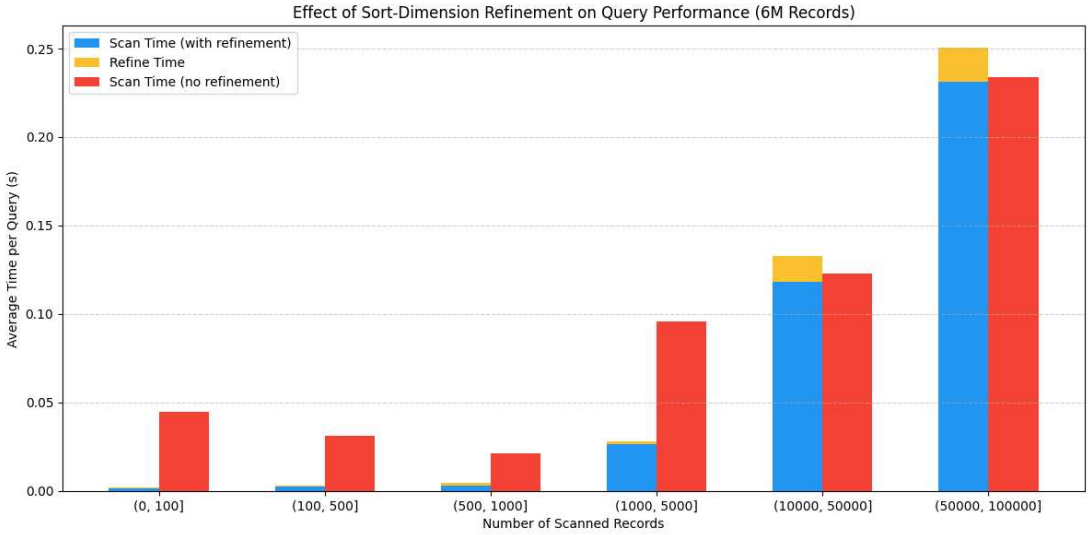

Figure 5.4: Effect of refinement on query time across `Ns` bins for the 6M dataset. Blue: scan time with refinement; yellow: refinement time; red: scan time without refinement. Refinement is most beneficial in mid-range bins where scan cost is high and refinement overhead remains small.

5.3.3 Flattening Options and Evaluation

To prevent data skew from concentrating records into a few large cells, Flood uses flattening during layout construction. We compared the RMI and empirical CDF-based flattening. Both approaches produced relatively balanced layouts, but empirical CDF offered superior runtime performance due to its lightweight access and more uniform distribution.

Although empirical CDF flattening incurs higher index creation cost (due to the need to sort and estimate quantiles), it slightly reduces query time, especially in high-selectivity workloads. Our experiments confirmed that CDF-based flattening maintained tighter control over cell sizes, leading to more consistent query behavior.

For one representative layout, the final grid consisted of 7,700 cells, with the following statistics:

- **Mean data points per cell:** 1,818
- **Standard deviation:** 135.5 (about 7.45% of the mean)
- **Minimum / Maximum:** 1,129 to 2,598

- **Interquartile range (IQR):** 567

The statistics above reflect a uniform distribution of records across cells, avoiding overloaded regions and ensuring that most queries interact with cells of similar size. The low standard deviation and absence of extreme outliers improve the predictability of scan time and reduce refinement variance. These properties imply that CDF-based flattening is an effective strategy for layout regularization.

Overall, while both RMI and CDF-based flattening are viable, we found the empirical CDF to be more robust and useful in practice. Equal-width binning, which is equivalent to disabling flattening, led to significantly higher cell size skew and increased query time consistency, especially in the presence of heavy-tailed or clustered attribute distributions.

5.3.4 Modular Performance Analysis

To understand the contribution of individual components within Flood's query processing pipeline, we performed a systematic, modular performance evaluation. This included toggling on/ off the flattening, refinement, exact range skipping, and refinement components (PLM vs BST), then measuring their impact on scan, projection, refinement, and total query processing time. The measurement results are shown in Table 5.1.

Configuration	Scan	Projection	Refinement	Total
Flattening on	25.60	0.223	0.191	26.05
Flattening off	28.30	0.071	0.261	28.67
Refinement off	28.70	0.152	0.052	28.91
Refinement on	25.30	0.141	0.401	25.81
Exact range check on	28.70	0.149	0.225	29.13
Exact range check off	25.20	0.144	0.228	25.60

Table 5.1: Average query time by component and configuration (all times in milliseconds).

Considering these results, we make several observations as follows:

- **Skipping exact range cells significantly reduces total query time.** Enabling this optimization allows the system to skip the scanning step for cells that fully satisfy the query predicates. If a cell lies entirely within the query range, all of its records are guaranteed to match and can be returned directly, avoiding unnecessary refinement.

This reduces the total query time from 29.1 ms to 25.6 ms (a 12% improvement), with most of the benefit observed in scan time (reduced from 28.7 ms to 25.2 ms).

- **Refinement yields a meaningful time reduction.** With refinement enabled, total query time drops from 28.9 ms to 25.8 ms (11% gain). This is despite a 0.4 ms refinement overhead, which is outweighed by the scan time reduction from 28.7 ms to 25.3 ms.

- **Flattening improves overall performance.** Our experiments showed that using empirical CDF flattening leads to a 9.1% decrease in total query time (28.7 ms to 26.1 ms). This is largely due to scan time improvements, as the flattening step balances data across cells, hence reducing worst-case scan volume. Given the overall gain, a slight increase in projection time overhead (from 0.07 ms to 0.22 ms) is reasonable.

Table 5.1 shows the impact of each feature in isolation. On the other hand, Figure 5.5 illustrates how combinations of flattening, refinement, refinement method, and exact range checking affect the overall query performance.

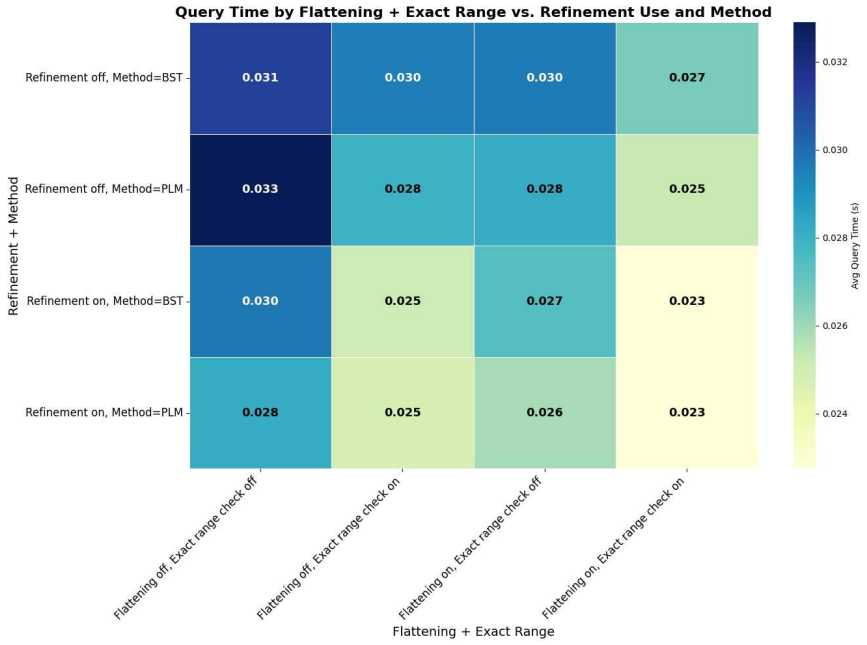

Figure 5.5: Query time (s) across combinations of flattening, refinement (on/off), refinement method (PLM/BST), and exact range checking.

The heatmap in Figure 5.5 confirms that the fastest configurations consistently use flattening, enable refinement, and use PLM over BST. Exact range checking is also beneficial across all datasets and query workload combinations.

This analysis reinforces that Flood's performance is highly sensitive to scan-time optimization. Techniques that prune candidate records early (via flattening, refinement, or skipping unnecessary checks) yield the most significant performance benefits.

5.4 Cost Model Evaluation

5.4.1 Motivation and Design Rationale

Flood's layout optimization relies on accurately predicting query execution cost across candidate layouts. Recall that the cost model proposed for the Flood framework is based on a linear combination of three subcomponents—projection, refinement, and scan—weighted by the coefficients w_p, w_r, and w_s, respectively. However, our findings revealed a critical shortcoming: these weights are only useful if the estimates of N_c and N_s are closely accurate.

Flood's use of downsampling heuristics to approximate N_c and N_s led to significant estimation error, particularly for N_s, which contributes mostly to the total query time due to the cost of a full scan. As a result, even when the weights w_p, w_r, and w_s were closely estimated, the total predicted time was often inaccurate, which in turn led to undermining the optimizer's ability to find desired layouts.

To address this, we redesigned the model pipeline as two separate learning tasks:

(1) A **neural network** with two hidden layers (10 neurons each) to predict N_c and N_s.

(2) An **XGBoost regressor** to estimate the cost weights w_p, w_r, and w_s, and the total query processing time.

5.4.2 Feature Engineering and Data Preparation

Both variants of the cost models were trained using feature vectors derived from the index and data characteristics. These features were computed using a lightweight analysis version of Flood that simulates grid creation and projection without full index construction. This made the training process scalable and layout-independent. Input features included:

- **Per-column statistics**: They include `min`, `max`, `skewness`, and `kurtosis`.

- **Grid-level statistics**: They include the number of cells, average points per cell, median cell size, 75th and 90th percentiles.
- **Layout shape**: This includes flattening type, bin counts per dimension, and total cell volume.
- **Density metrics**: They include standard deviation and interquartile range of cell sizes and sparsity estimates.

The cost models were trained and evaluated using over 6,000 queries spanning three different datasets:

- **TPC-H Lineitem (scaled)** — a semi-structured benchmark with temporal and numerical skew.
- **Sales dataset** — a transactional dataset with strong seasonality.
- **Synthetic benchmark** — designed with configurable skew and overlap characteristics.

5.4.3 Model Accuracy and Evaluation Metrics

For a comprehensive evaluation of the model, we used three complementary measures:

- **Mean Absolute Error (MAE)**: captures average deviation in original measure units (e.g., number of records or time(seconds).
- **Mean Absolute Percentage Error (MAPE)**: normalizes error relative to ground truth, especially useful for comparing variables with small magnitudes.
- **R^2 Score**: measures the proportion of variance explained by the model, which indicates how well the model fits the data.

Table 5.2 summarizes the predictive performance. The Category column groups related prediction targets: intermediate variables, internal cost weights, and the final output. The Target column lists the specific variable being predicted by the model. N_c and N_s represent the number of intersecting grid cells and the number of scanned records, respectively. w_p, w_r, and w_s are internal weights learned to estimate projection, refinement, and scan costs. Total Time refers to the final predicted query latency in milliseconds.

Category	Target	R^2 Score	MAPE (%)
Intermediate Estimates	N_c	0.9998	1.11
	N_s	0.9561	2.87
Cost Weights	w_p	-0.051	5.92
	w_r	0.014	3.12
	w_s	0.9355	12.69
Final Output	Total Time	**0.99999**	**0.13**

Table 5.2: Prediction accuracy of cost model components grouped by output type.

The **R^2 Score** measures how well the model explains the variance in each target, with values closer to 1 indicating a better fit. The **MAPE (%)** column reports the Mean Absolute Percentage Error, showing the average relative prediction error in percentage terms. Bold values highlight particularly high accuracy for total query time estimation.

5.4.4 Model Comparison and Impact

Our experiments confirmed that:

- The neural network outperformed Flood's original downsampling-based heuristic, which estimates N_c and N_s by querying a small uniform sample of the dataset, by approximately 36%, enabling more reliable cost estimation.
- XGBoost showed a minor but consistent improvement in estimating w_s compared to Random Forest, with lower MAPE and better R^2.
- For the weights w_p and w_r, both models performed similarly, with low R^2 scores but minimal real-world impact due to their small magnitude in total computation time.

Despite limited improvements in individual weight estimation, the learned model predicted total query time with exceptional accuracy (**R^2 = 0.99999**, **MAPE = 0.13%**). This underscores the importance of accurate N_s modeling—when scan dominates query time, predicting N_s well outweighs precise weight calibration.

5.4.5 Conclusion and Outlook

Our proposed cost model advances the initial Flood framework by isolating and addressing the core estimation bottlenecks. Instead of focusing exclusively on modeling

cost weights, we identified N_s as the primary determinant of overall latency and used a lightweight neural network to model it explicitly. The result is a modular and robust cost model that enables better layout optimization and is adaptable to new workloads or data distributions.

Future work may focus on integrating uncertainty estimates into cost predictions and extending the model to capture the impact of query dimensionality and join workloads; these ideas could further enhance Flood's capability and adaptability to real-world indexing scenarios.

5.5 Layout Optimization Analysis

This section presents the empirical results of layout optimization in the Flood indexing framework. We evaluate and compare different optimization strategies in terms of convergence behavior, robustness, and their effect on end-to-end query performance.

5.5.1 Optimization Algorithm Comparison

We benchmarked three optimization strategies for tuning Flood's layout: the original gradient descent approach proposed in the original Flood framework, greedy coordinate descent with random restarts, and simulated annealing. Each one of these strategies was evaluated based on convergence speed, cost reduction quality, and stability.

Flood's gradient descent method attempts a global layout update using finite difference approximations of the gradient. While it achieved rapid initial improvements in estimated cost, its trajectory frequently oscillated due to conflicting updates across dimensions. Additionally, the use of a discrete step size and integer constraints often caused it to get trapped in local optima. As a result, this layout optimization algorithm exhibited unstable convergence and was sensitive to initialization, making it unreliable in practice.

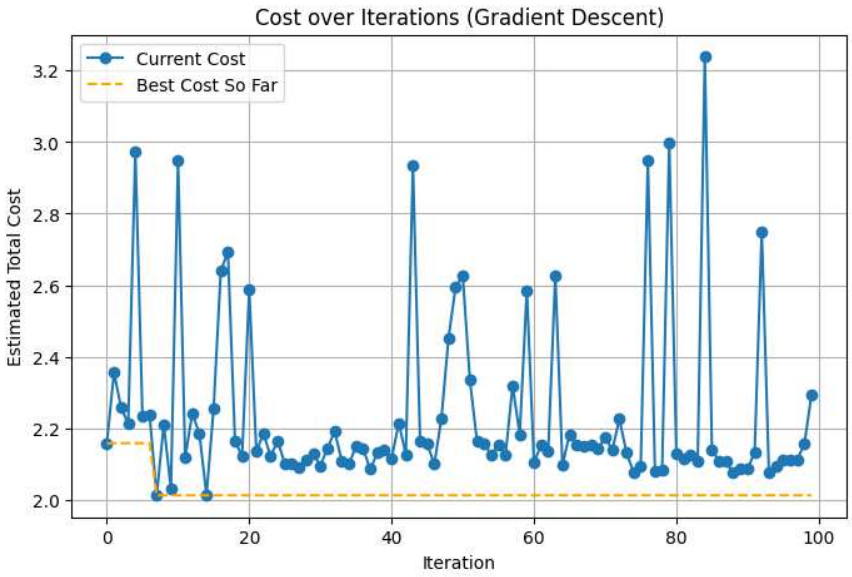

Figure 5.6: Optimization using Gradient Descent: Estimated cost over iterations. Rapid early drop is followed by oscillations and stagnation due to conflicting updates across dimensions.

Figure 5.6 visualizes the estimated total cost over 100 iterations of Flood's gradient descent algorithm. The solid blue line represents the cost at each iteration, while the dashed orange line shows the lowest cost observed so far. Although the initial iterations show a steep cost reduction, subsequent updates exhibit erratic oscillations without meaningful progress. These fluctuations highlight the instability of the optimization trajectory, likely due to conflicting updates across dimensions and the inability to perform fine-grained tuning with discrete integer steps. The algorithm fails to consistently improve the cost, getting stuck in poor local optima, which underscores its sensitivity to initialization and limited reliability.

Simulated annealing introduced stochastic updates, allowing cost-increasing steps to escape local minima. While the initial iterations led to a sharp drop in estimated cost, the best cost plateaued early and failed to improve further. The trajectory remained noisy and flat for the rest of the run, as shown in Figure 5.7. This suggests limited exploration capability in later iterations and difficulty in navigating the high-variance, non-convex cost landscape.

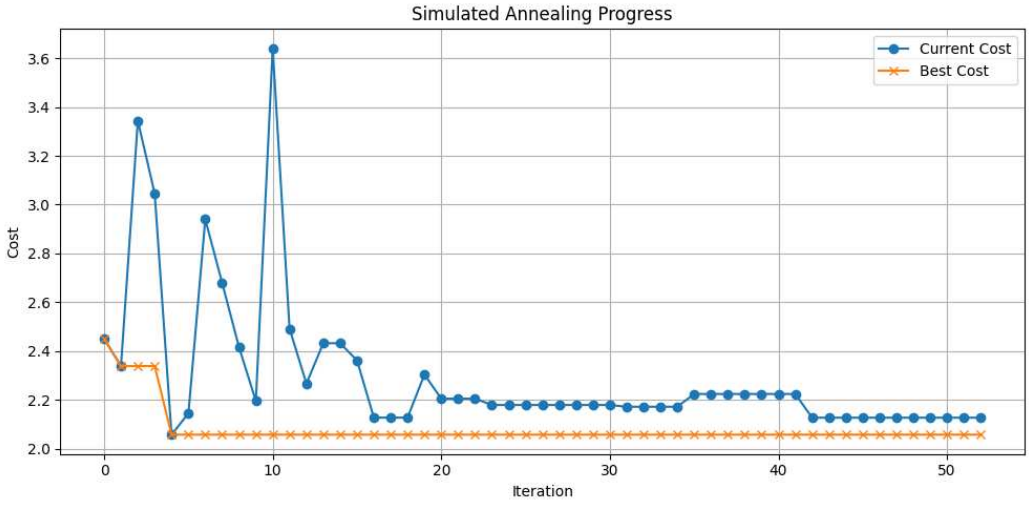

Figure 5.7: Simulated Annealing: Current cost vs. best cost over 100 iterations. The best cost flattens early, indicating limited progress after initial gains.

The most effective optimizer we used in our evaluation was the greedy coordinate descent method with random restarts. This method incrementally adjusts one bin count per iteration in the direction that yields the greatest cost reduction. Random restarts from diverse initial layouts help prevent the algorithm from getting stuck in poor local minima. As shown in Figure 5.8, all three restarts consistently converged to similar low-cost regions, and one of them achieved the lowest cost among all tested methods. Although the per-iteration cost trajectory appears noisier compared to gradient descent, the *best cost found so far* improves steadily over time, which demonstrates exploration capability and reliable convergence.

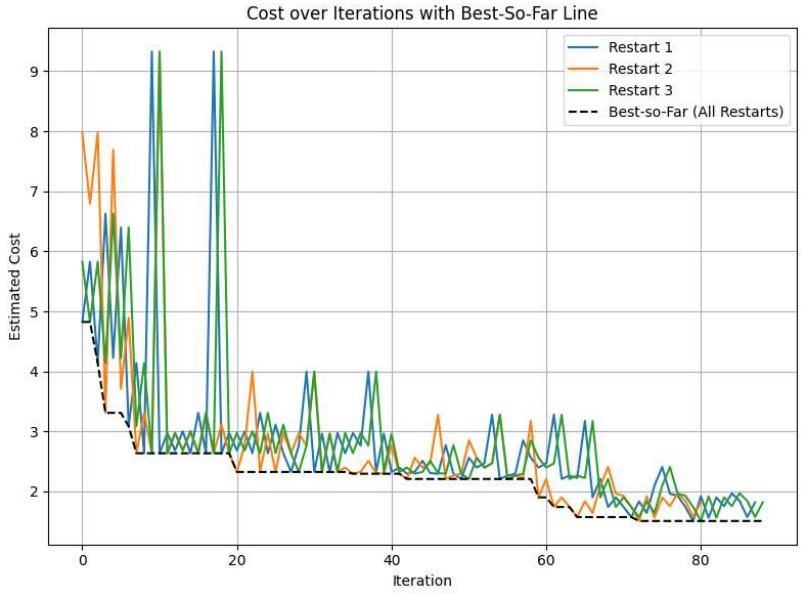

Figure 5.8: Greedy Coordinate Descent: Cost over iterations across three random restarts. All runs converge to similarly low-cost regions, with one achieving the best observed cost. Despite per-iteration fluctuations, the best cost consistently improves.

In summary, while simulated annealing and gradient descent both demonstrated early gains, they both ultimately suffered from stagnation and instability, respectively. Greedy coordinate descent, in contrast, was both robust and effective, consistently converging to better layouts with fewer failures. On a dataset with 6 million records, the full optimization process—including cost model inference and layout evaluation—completed in approximately 420 seconds on a standard desktop PC we used in our experiments. On the TPC-H dataset with 6 million records and a workload of 20 queries, both Flood's original gradient descent and simulated annealing procedures converged to layouts with an average estimated query cost of approximately **2.1 seconds**. In contrast, greedy coordinate descent with restarts achieved a much lower cost of approximately **1.5 seconds** on the same workload, highlighting its superior optimization capability.

5.6 Comparison with Traditional Indexes

While the original Flood paper reports significant performance gains, such as a **135 times speedup over a full scan**, **4.68 times over a KD-Tree**, and **3.25 times over Z-order indexing**, our implementation of Flood in Python provided more modest but still reproducible and analytically useful improvements. On the TPC-H dataset, using a query

workload adapted from common filters on the `lineitem` table and standardized to span a range of selectivities, our implementation achieved an average **17.1 times speedup over full scan**, **1.64 times over KD-Tree**, and **4.35 times over Z-order indexing** across datasets of varying sizes.

Flood consistently outperformed traditional baselines across all the number of records in the dataset, though the performance gap was more modest on smaller datasets (under 1 million records). As dataset sizes increased, the benefits of Flood became more pronounced, with brute-force scan times rising sharply and traditional indexes like KD-Tree and Z-order exhibiting degraded performance.

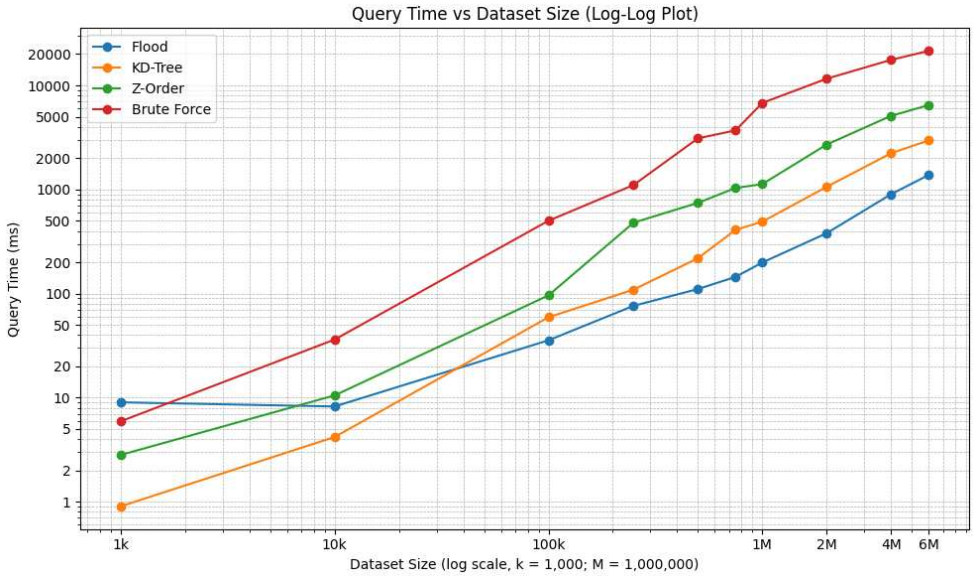

Figure 5.9: Query performance on the TPC-H `lineitem` table across dataset sizes, evaluated on a standardized workload of 100 range queries. Flood maintains low latency as data scales, while KD-Tree, Z-order, and full scan degrade significantly.

This trend is clearly illustrated in Figure 5.9, where query times for Flood grow much slower compared to the exponential rise observed in brute-force scans. Z-order and KD-Tree show better scalability than full scan, but eventually become less efficient as dimensionality and data volume increase. To further highlight this difference, Figure 5.10 focuses on the 6 million record dataset, revealing that Flood achieves the lowest query latency even at scale.

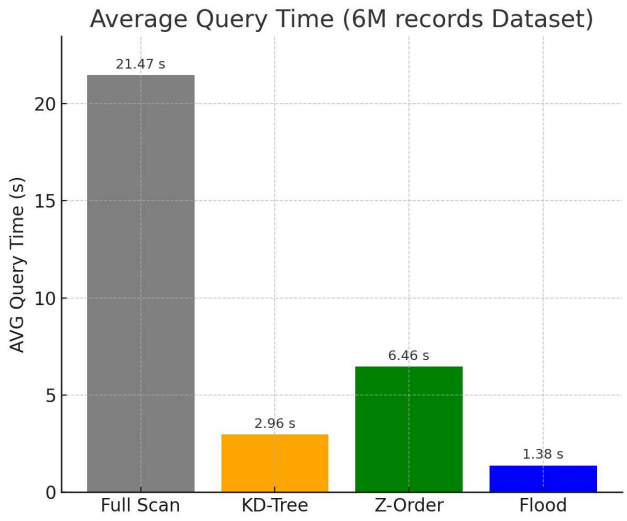

Figure 5.10: Average query time on the TPC-H `lineitem` table with 6 million records. Flood achieves the lowest latency among all baselines.

Although these results fall short of the results reported in the Flood paper, the discrepancy is likely due to several factors. First, the original implementation of Flood was written in C++, benefiting from hardware-level optimizations such as SIMD instructions and memory prefetching. In contrast, our implementation was developed in Python, an interpreted language with higher runtime overhead. Second, their benchmarks were conducted on a much larger scale—for example, using a 30-million-record version of TPC-H—whereas we were limited by our PC to consider datasets with about 6 million records. Third, the KD-Tree and Z-order baselines used in my evaluation were not aggressively tuned, which may have led to lower speedups relative to a brute-force scan.

While Flood's query-time advantages are clear, they come at a higher index creation cost. Figure 5.11 shows that Flood consistently takes longer to build than KD-Tree and Z-order indexing methods, even before accounting for its learning phase. This was expected, as Flood's construction includes CDF-based flattening, grid partitioning, and PLM-based structures for refinement. In addition, the layout learning phase adds around 8 minutes of optimization to support the 6 million record dataset.

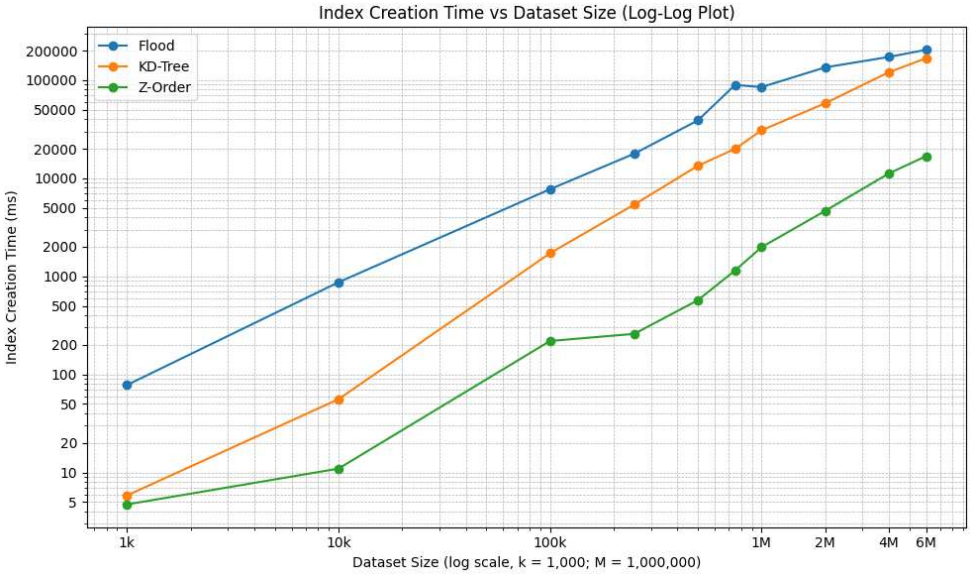

Figure 5.11: Index creation time on the TPC-H `lineitem` table across dataset sizes. Flood incurs higher construction costs due to its multi-stage design, even before layout learning is applied.

Despite this overhead, the significant gains in query latency—especially on large datasets—make Flood particularly suitable for analytical scenarios where fast, repeated query execution justifies the upfront indexing cost. The trade-off between indexing time and query performance becomes especially favorable as data volume increases.

5.7 Limitations and Open Questions

Despite its strengths, the Flood indexing framework has several limitations:

- **Sensitivity to heavily skewed data:** When the data is highly skewed, it becomes harder to learn balanced layouts. This makes it challenging to find an effective trade-off between reducing scan volume and avoiding high projection and refinement overhead.
- **High index creation and optimization cost:** Flood requires significant time for index construction and layout tuning, which may be impractical for real-time or latency-sensitive applications.
- **Limited support for highly dynamic workloads:** In highly dynamic environments

where data or query patterns change frequently, Flood's need for re-optimization and full index rebuilding limits its applicability.

- **High memory usage:** The current implementation operates entirely in-memory and assumes sufficient RAM is available, which restricts scalability on resource-constrained systems.
- **No support for joins or aggregations:** Flood is currently optimized for range selection queries over numeric attributes. Extending its capabilities to support join operations or aggregate functions (e.g., `SUM`, `AVG`) remains a challenge.

5.8 Summary of Contributions

This chapter presented a detailed experimental evaluation of the Flood indexing framework, analyzing its internal query-time behavior, component-level contributions, optimization strategies, and performance relative to traditional multi-dimensional indexing techniques.

We found that scan operations dominate query execution time, accounting for approximately 90 percent of the total time across a wide range of dataset sizes and query selectivities. Reducing the number of scanned records (`Ns`) is therefore essential for improving performance. While the cost of projection and refinement steps is smaller, they increase with finer grid resolutions due to the growing number of intersecting cells (`Nc`), thus creating trade-offs to manage which affect the overall efficiency.

Our systematic study and analysis demonstrated that flattening, refinement, and skipping exact ranges significantly improve performance by reducing scan volume or eliminating unnecessary filtering. Among flattening strategies, empirical CDF-based flattening provided the most consistent results, producing balanced grid layouts and stable query latency.

To support layout optimization, a machine-learned cost model was developed. It accurately predicted query time by separately estimating intermediate quantities like `Ns` and `Nc` using a neural network, and final query cost using XGBoost regressors. The resulting model achieved near-perfect accuracy, with R^2 score of 0.99 and MAPE of 0.13%.

We compared three layout optimization algorithms. We observed that Gradient Descent provided early gains but often became unstable due to conflicting updates and integer constraints. Simulated annealing performed better in escaping local minima, but its progress plateaued early and remained noisy. In contrast, greedy coordinate descent with

random restarts showed robust and reliable convergence. It consistently discovered high-performing layouts and outperformed the other methods in both consistency and final cost.

On the TPC-H dataset with 6 million records and a workload of 20 queries, both gradient descent and simulated annealing converged to layouts with an average estimated query cost of approximately 2.1 seconds. Greedy coordinate descent achieved a lower cost of 1.5 seconds, confirming its effectiveness in this optimization setting.

Although Flood incurred higher index creation time compared to KD-Trees and Z-order indexing due to its multi-stage pipeline and layout learning phase, it delivered significantly faster query execution. For analytical workloads where queries are executed repeatedly, the upfront indexing cost is justified by the improved runtime performance.

These results highlight the importance of scan-time optimization, data-balanced layouts, and cost-aware learning strategies in enabling scalable and efficient multi-dimensional indexing.

Bibliography

[1] V. Nathan, J. Ding, M. Alizadeh, and T. Kraska, "Learning multi-dimensional indexes," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, Portland, OR, USA, Jun 2020, pp. 985–1000.

[2] A. Pavlo, G. Angulo, J. Arulraj, H. Lin, J. Lin, L. Ma, P. Menon, T. C. Mowry, M. Perron, I. Quah *et al.*, "Self-driving database management systems," in *Proc. Conf. Innovative Data Systems Research (CIDR)*, vol. 4, 2017, p. 1.

[3] R. Marcus, P. Negi, H. Mao, C. Zhang, M. Alizadeh, T. Kraska, O. Papaemmanouil, and N. Tatbul, "Neo: A learned query optimizer," *Proc. VLDB Endow.*, vol. 12, no. 11, pp. 1705–1718, Jul. 2019.

[4] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis, "The case for learned index structures," in *Proceedings of the 2018 ACM SIGMOD International Conference on Management of Data*, Houston, TX, USA, Jun 2018, pp. 489–504.

[5] R. Weber, H.-J. Schek, and S. Blott, "A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces," in *Proceedings of the 24th International Conference on Very Large Data Bases (VLDB)*. New York City, NY, USA: Morgan Kaufmann, Aug 1998, pp. 194–205.

[6] A. Guttman, "R-trees: A dynamic index structure for spatial searching," in *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*. Boston, MA, USA: ACM, Jun 1984, pp. 47–57.

[7] G. M. Morton, "A computer oriented geodetic data base; and a new technique in file sequencing," *IBM Technical Report*, no. GA36-0038, 1966.

[8] R. Bayer and E. M. McCreight, "Organization and maintenance of large ordered indices," in *Proceedings of the 1970 ACM SIGFIDET Workshop on Data Description, Access and Control*. San Diego, CA, USA: ACM, 1970, pp. 107–141.

[9] D. Comer, "The ubiquitous b-tree," *ACM Computing Surveys*, vol. 11, no. 2, pp. 121–137, Jun 1979.

[10] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The r*-tree: An efficient and robust access method for points and rectangles," in *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*. Atlantic City, NJ, USA: ACM, May 1990, pp. 322–331.

[11] I. Kamel and C. Faloutsos, "Hilbert r-tree: An improved r-tree using fractals," in *Proc. 20th Int. Conf. on Very Large Data Bases (VLDB)*. San Francisco, CA, USA: Morgan Kaufmann, 1994, pp. 500–509.

[12] PostgreSQL Global Development Group, "Sp-gist indexes," https://www.postgresql.org/docs/current/spgist.html, 2023.

[13] PostGIS Project, "Postgis indexing," https://postgis.net/workshops/postgis-intro/indexing.html, 2023.

[14] MySQL Developers, "Creating spatial indexes," https://dev.mysql.com/doc/refman/8.0/en/creating-spatial-indexes.html, 2023.

[15] Z. Christopherson, "Amazon redshift engineering's advanced table design playbook: Compound and interleaved sort keys," https://aws.amazon.com/blogs/big-data/amazon-redshift-engineerings-advanced-table-design-playbook-compound-and-interleaved-sort-keys/, 2016.

[16] SciPy Developers, "scipy.spatial.kdtree," https://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.KDTree.html, 2023.

[17] J. Ding, V. Nathan, M. Alizadeh, and T. Kraska, "Tsunami: A learned multi-dimensional index for correlated data and skewed workloads," in *Proceedings of the VLDB Endowment*, vol. 14, no. 2, Oct. 2020, pp. 74–86.

[18] P. Li, H. Lu, Q. Zheng, L. Yang, and G. Pan, "Lisa: A learned index structure for spatial data," in *Proc. ACM SIGMOD Int. Conf. on Management of Data*, Portland, OR, USA, 2020, pp. 2119–2133.

[19] H. Abu-Libdeh, D. Altınbüken, A. Beutel, E. H. Chi, L. Doshi, T. Kraska, X. Li, A. Ly, and C. Olston, "Learned indexes for a google-scale disk-based database," Presented at the Workshop on ML for Systems at NeurIPS, 2020.

[20] T. K. Sellis, N. Roussopoulos, and C. Faloutsos, "The r+-tree: A dynamic index for multi-dimensional objects," in *Proc. 13th Int. Conf. on Very Large Data Bases (VLDB)*, P. M. Stocker, W. Kent, and P. Hammersley, Eds. Morgan Kaufmann, 1987, pp. 507–518.

[21] L. Arge, M. de Berg, H. J. Haverkort, and K. Yi, "The priority r-tree: A practically efficient and worst-case optimal r-tree," in *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 2004, pp. 347–358.

[22] L. Arge, M. de Berg, and H. Haverkort, "Cache-oblivious r-trees," in *Proc. 21st Annual Symposium on Computational Geometry*, 2005, pp. 170–179.

[23] Oracle Corporation, "Oracle database concepts 12c release 1 (12.1)," https://docs.oracle.com/database/121/CNCPT/indexes.htm, 2013.

[24] M. Stonebraker and L. A. Rowe, "The design of postgres," in *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*. Washington, D.C., USA: ACM, 1986, pp. 340–355.

[25] J. Ding, R. Marcus, A. Kipf, V. Nathan, A. Nrusimha, K. Vaidya, A. van Renen, and T. Kraska, "Sagedb: An instance-optimized data analytics system," in *Proceedings of the VLDB Endowment*, vol. 15, no. 13, Sep. 2022, pp. 4062–4078.

[26] Z. Jiang, J. Huang, Q. Zhang, H. Liu, and C. Wu, "Lilis: Enhancing big spatial data processing with lightweight distributed learned index," *arXiv preprint arXiv:2504.18883*, 2025.

[27] R. Finkel and J. Bentley, "Quad trees: A data structure for retrieval on composite keys," *Acta Informatica*, vol. 4, no. 1, pp. 1–9, 1974.

[28] Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *IEEE Transactions on Neural Networks*, vol. 5, no. 2, pp. 157–166, 1994.

[29] C. Blum and A. Roli, "Metaheuristics in combinatorial optimization: Overview and conceptual comparison," in *Handbook of Metaheuristics*, F. Glover and G. A. Kochenberger, Eds. Springer, 2003, pp. 3–35.

[30] H. Fang, G. Fang, T. Yu, and P. Li, "Efficient greedy coordinate descent via variable partitioning," in *Proceedings of the 24th International Conference on Artificial Intelligence and Statistics (AISTATS)*, ser. Proceedings of Machine Learning Research, vol. 130. PMLR, 2021, pp. 1419–1427.

[31] D. Henderson, S. H. Jacobson, and A. W. Johnson, "The theory and practice of simulated annealing," in *Handbook of Metaheuristics*, F. Glover and G. A. Kochenberger, Eds. Springer, 2003, pp. 287–319.

[32] Transaction Processing Performance Council (TPC), "Tpc-h," http://www.tpc.org/tpch/, 2019.