Author/s:
Liu, Guanli

Title:
Learning Spatial Indices Efficiently

Date:
2023-06

Persistent Link:
https://hdl.handle.net/11343/337761

Learning Spatial Indices Efficiently

by

Guanli Liu

ORCID: 0000-0002-3298-5524

A thesis submitted in total fulfillment for the
degree of Doctor of Philosophy

in the
School of Computing and Information Systems
Faculty of Engineering and Information Technology
THE UNIVERSITY OF MELBOURNE

October 2023

Abstract

Machine learning and database management systems have been extensively researched for many years. A database management system, while supporting data persistence and stable queries, can often encounter efficiency issues when building indices for querying big data.

In recent years, machine learning is gradually used in databases to solve efficiency issues such as knob tuning, index selection, cost estimation, and index building. Most problems are solved by using regression models and reinforcement learning. Here, using machine learning techniques for index building leads to a new type of index structures named the *learned index*, which has shown better query performance than traditional indices, e.g., B-trees and R-trees.

However, the efficiency of building learned indices is still a key challenge, especially when facing large-scale datasets. The reason is that index building is based on the whole dataset, which requires a full dataset scan in each epoch, and there is at least one epoch for building a learned index. Existing learned indices suffer from this issue, which hinders the application of indices in the database management system.

In this thesis, we address this efficiency issue for index learning over a special type of data, the spatial data, i.e., data associated with geographical location information, the volume of which is rapidly growing due to the prevalence of smart mobile devices, the Internet of Things, and 5G networks. We study four research problems on effective and efficient spatial data indexing using machine learning techniques.

The first problem is an empirical study of two learned spatial indices RSMI and ZM, which support point, range, and kNN queries. These indices have reported better query performance than a traditional spatial index, the R-trees. However, there is no open-source code or extensive basis that supports testing and evaluating these learned indices against other traditional spatial indices for large real-world datasets. We address such an issue by offering an implementation of these learned indices. Based on the implementation, we present a thorough empirical analysis of the advantages and disadvantages of learned spatial indices, highlighting their significant build times and motivating our studies to optimize the build time efficiency of learned spatial indices.

In the second research problem, we address the efficiency issue in learning spatial indices by proposing an index learning framework called ELSI. The key idea of ELSI is that learning from a smaller dataset can derive similar query performance to that using the full input dataset. Experiments on real datasets with over 100 million points show that

ELSI can reduce the build times of four different learned spatial indices consistently and by up to two orders of magnitude without jeopardizing query efficiency.

In the third research problem, we propose to pre-train index models offline and only fine-tune them online for index learning to accelerate the building of learned (spatial) indices. The results show that we improve the build time of learned one-dimensional indices by 30.4% and improve lookup efficiency by up to 24.4% on real datasets and 22.5% on skewed synthetic datasets. When this technique is applied to spatial data, it speeds up learned spatial index building by two orders of magnitude, while the lookup efficiency can also be increased by up to 13%

While learned spatial indices have shown strong query performance, their structure and query algorithms differ drastically from the traditional indices, which are well supported by off-the-shelf database systems. To reduce the additional overhead of replacing traditional indices with learned spatial indices, in the fourth research problem, we study applying learning-based techniques to optimize the structure of traditional spatial indices. We focus on indices based on space-filling curves (SFC). SFCs are a classic technique to transform multidimensional (e.g., spatial) data into one-dimension values for data indexing. The choice of SFC for indexing over a particular dataset and query workload has a significant impact on the query performance of the resultant index. We propose algorithms that enable estimating the query performance of an SFC-based index without actually building the index, thus enabling efficient computing of a query-optimized SFC-based index. Experiments show that our cost estimation algorithms are over an order of magnitude faster than naive methods. The computed SFC indices outperform competing indices based on two classic types of SFCs, i.e., Z-curves and Hilbert curves, in nearly all settings considered.

Declaration of Authorship

I declare that this thesis titled, 'LEARNING SPATIAL INDICES EFFICIENTLY' and the work presented in it are my own. I confirm that:

- the thesis comprises only my original work towards the Ph.D.;
- due acknowledgment has been made in the text to all other material used; and
- the thesis is fewer than the maximum word limit in length, exclusive of tables, maps, bibliographies, and appendices as approved by the Research Higher Degrees Committee.

Signed: Guanli Liu

Date:

Preface

Portions of this thesis are based on published papers:

- Part of the contents of Chapter 3 has been published in the experimental section of following paper, to which I have made the majority of the contributions:

  Jianzhong Qi, Guanli Liu, Christian S. Jensen, Lars Kulik. *Effectively Learning Spatial Indices*, Proceedings of the VLDB Endowment (**PVLDB**) 2020.

- Part of the contents of Chapter 4 has been published in the following paper:

  Guanli Liu, Lars Kulik, Christian S. Jensen, James Bailey, Jianzhong Qi. *Efficiently Learning Spatial Indices*, IEEE International Conference on Data Engineering (**ICDE**) 2023.

- Part of the contents of Chapter 5 has been published in the following paper:

  Guanli Liu, Jianzhong Qi, Lars Kulik, Kazuya Soga, Renata Borovica-Gajic, Benjamin I. P. Rubinstein. *Efficient Index Learning via Model Reuse and Fine-tuning*, International Workshop on Databases and Machine Learning (**DBML**) 2023.

- Part of the contents of Chapter 6 is in preparation for submission to *International Conference on Management of Data* (**SIGMOD**) 2024.

Acknowledgements

Throughout the writing of this thesis through out my Ph.D. candidature, I have received a great deal of guidance and assistance. I would like to express my deepest gratitude to everyone for their invaluable support. Without their help and encouragement, this thesis would not have been possible.

First of all, I would like to thank my supervisors Dr. Jianzhong Qi and Prof. Lars Kulik, whose expertise was invaluable in formulating the research questions and methodology. Their insightful feedback pushed me to sharpen my thinking and brought my work to a higher level. Their continuous and meticulous help in polishing the papers greatly benefited me. Their thoughtful questions push me to prepare better and think deeper in the weekly meetings. I have learned a rigorous work attitude and meticulous and comprehensive ways of doing things from them. I also want to thank my industry supervisor Catherine Lopes from the IMNIS project, who is very generous and kind. She gave me a lot of advice on career development.

I am particularly grateful for the support and help from my Advisory Committee Chairs Prof. Shanika Karunasekera and Prof. Alistair Moffat. I am thankful to Alistair for always giving me thoughtful suggestions during my progress reviews and helping me to set up a clear and detailed plan to finish the thesis.

I would like to thank Prof. Christian S. Jensen for his patience and kind support. Christian is a researcher with significant contributions in the field of database management. His suggestions take our joint work to the next level.

I also acknowledge all my collaborators, Prof. Benjamin I. P. Rubinstein, Prof. James Bailey, Mr. Kazuya Soga, Dr. Renata Borovica-Gajic, Asst. Prof. Tianyi Li, and Assoc. Prof. Xingjun Ma, for their excellent collaboration and support.

I would like to thank Daisy Wang's family, Richard's family, Shiquan Yang, Xinyu Su, Yanchuan Chang, Yang Wang's family, Zhuo Zhang, and Zhuowei Zhao, for the joyful moments. During the journey of study, we spend a lot of time together. We enjoy meals, card games, and sports together. I also want to thank John Thomas for being my tennis partner, who keeps an optimistic attitude and enlightens me in many fields. I am delighted to have them during the study period.

In addition, I would like to sincerely thank my family, including my parents, my sister, brother in law, and my niece, for their wise counsel and sympathetic ear. Since coming to Australia, I have not had a chance to return home to stay with my family. I can only FaceTime with them every weekend morning. During the pandemic, their care sustained

me during my loneliest moments. Thanks to their help, I can live here comfortably and focus on studying and working. I sincerely hope we can reunite soon.

Last but not least, I would like to thank my girlfriend, Sonia Song. We went to many fun places and made many big decisions together. I could not have had a wonderful and happy life without her company. I was lucky enough to meet her and taste the desserts and birthday cakes she made for me. I wish we could always be happy.

Contents

List of Figures

List of Tables

Abbreviations

AI	Artificial Intelligence
ALEX	Adaptive Learned Index
BMC	Bit-merging Curve
CDF	Cumulative Distribution Function
CL	Clustering
CPU	Central Processing Unit
DBMS	Database management system
DP	Dynamic Programming
DQN	Deep Q-network
DRL	Deep Reinforcement Learning
DT	Decision Trees
ELSI	Efficient Learning of Spatial Indices
EMD	Earth Mover's Distance
FFN	Feedforward Neural Network
GPU	Graphics Processing Unit
HC	Hilbert Curve
kNN	k Nearest Neighbor
KS	*Kolmogorov–Smirnov*

LBMC	Learned Bit-merging Curve
LC	Lexicographic-order Curves
LISA	Learned Index Structure for Spatial Data
LUD	Learn Update Distribution
LUDB	Learn Update Distribution with Bound
MBR	Minimum Bounding Rectangle
MDP	Markov Decision Process
ML	Machine Learning
MLP	Multilayer Perceptron
MMI	Multi-staged Model Index
MR	Model Reuse
NN	Nearest-Neighbor
OG	OriGinal
PDF	Probability Density Function
PG	PostgreSQL
PGM-index	Piecewise Geometric Model index
POIs	Points of Interests
QUILTS	QUery-Intensive Linearization Tolerating data Skew
RL	Reinforcement Learning
RMI	Recursive Model Index
RF	Random Forests
RS	Representative Set
RS-index	RadixSpline
RSMI	Recursive Spatial Model Index

SBMC	Selected Bit-merging Curve
SBR	Sort-Based R-tree
SP	Sampling
SFC	Space-filling Curves
URLs	Uniform Resource Locators
UUIDs	Universally Unique IDentifiers
ZC	Z-curve

Chapter 1

Introduction

Spatial data keeps increasing at an unprecedented pace. Key drivers are remote sensing, GPS, and the omnipresence of mobile devices. The volume and rate of data growth challenges traditional index structures.

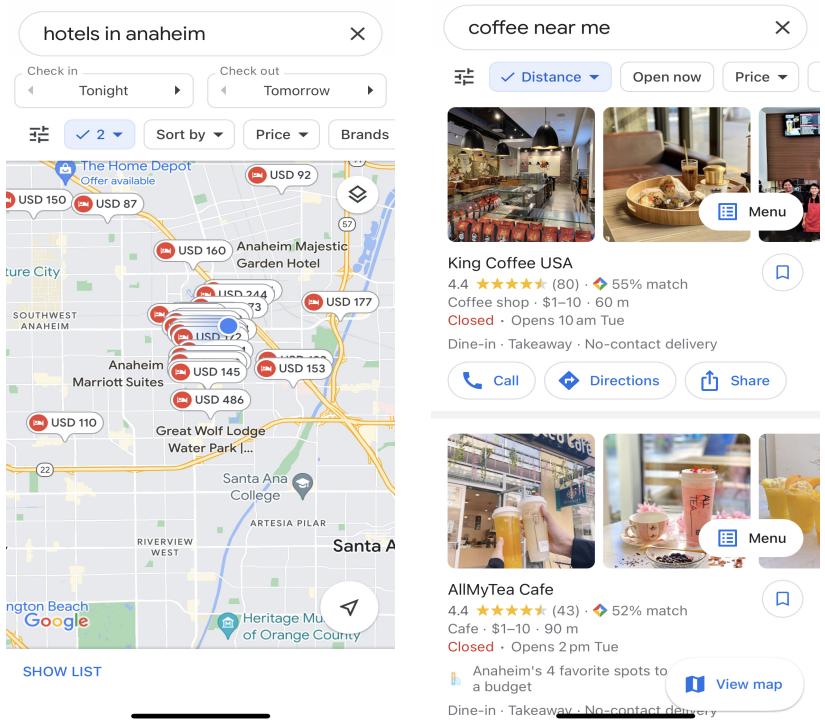
To improve query efficiency on spatial data, there is a paradigm shift: traditional spatial indices are being replaced by learned spatial indices [58, 84, 103], because they show significant performance improvements. The idea of using learning-based techniques is inspired by a *learned index* that uses Machine Learning (ML) techniques to build one-dimensional indices to replace traditional indices (e.g., B-trees) [53].

However, directly adapting ML faces a number of challenges. The most important one is that the index-building cost of learned spatial indices is expensive. For example, Learned Index Structure for Spatial Data (LISA) [58] reported a few hours to build an index structure over a dataset of 50 million points, which may not be practical in real applications.

In this thesis, we address the efficiency challenges for spatial index learning.

1.1 Background

Spatial data is increasingly important in a wide range of applications, including location-based services, urban planning, and public health. Mobile applications, e.g., TikTok, Twitter, and Google Maps are interacting with hundreds of millions of spatial data



(A) Range query (B) kNN query

FIGURE 1.1: Spatial queries

points. To provide better service based on the huge volume of Points of Interests (POIs), multiple spatial queries are designed. For example, range query and k Nearest Neighbor (kNN) query are two typical spatial queries that we normally use in Google Maps. The query "hotels in anaheim (mobile window view)" (Figure 1.1(a)) is a typical example of spatial range query, which lists all the returned hotels with the price on the map. Here, Anaheim is the query range. As another example, "coffee near me" (Figure 1.1(b)) returns coffee shops sorted by their distances to the app user. This is a kNN query, where the spatial objects are coffee shops.

With the growing availability of spatial data, it has become crucial to effectively and efficiently manage and query big spatial data (at billion scale or more). Spatial indices have been designed to efficiently store and manage spatial data. They are used to organize spatial data and speed up spatial queries by helping prune the search space at query processing. *R-trees* [37], *kd-trees* [13], and *quadtrees* [31] are classic spatial indices. These indices have a hierarchical structure. When processing spatial queries with such indices, a tree traversal is always required, which may still have to access many tree

nodes and yield decreasing performance as the index size grows with the data, especially when the indices reside on external memory.

1.2 Motivation and Challenges

In recent years, Artificial Intelligence (AI) (especially ML) techniques have been introduced to enhance database systems. ML studies use training data to build models via learning algorithms, which are typically used to improve the efficacy in terms of accuracy, for example, for email filtering, speech recognition, and computer vision [24, 79, 92, 102]. With the support of ML, many traditional research questions in database fields are being solved with a new generation of approaches that have been summarized with the term "AI4DB" [57].

The reason why ML is now widely used in databases is that ML models can learn from data and return predictions that are likely to be better than what traditional heuristic methods can offer. With an advanced learning technique called Reinforcement Learning (RL), ML models can further learn to make progressive decisions aiming to optimize some tasks in the end, which helps improve the decision-making efficiency and quality by focusing on the most promising sub-space of the solution space.

ML has also been introduced into index building, resulting in a new family of index structures called the *learned indices* [30, 53, 58, 70]. The key idea of learned indices is that a model can learn a sorted order of data records and use the model to predict the record positions, and hence the time cost of locating the records relevant to a query key is decreased. Also, the index memory footprint is reduced because the model size is just the number of parameters to represent the model. Based on the improvement of learned indices (e.g., Recursive Model Index (RMI) [53]) against B-tree, researchers consider extending this technique to other research questions, e.g., learning spatial indices.

To enable learning-based techniques for spatial data indexing, a number of learned spatial indices have been designed, which are essentially reduced to two difference types of approaches that obtain one-dimensional values for index learning [23, 58, 103]. One approach is to order the data records and map each data point to a key [23, 103]. Another approach is to partition a dataset of d dimensions onto $d-1$ dimensions and learn a one-dimension index on the last dimension [26, 58, 70].

However, learning spatial indices still suffer from *the efficiency of index building*. For example, LISA [58] reported a few hours to build an index structure over a dataset of 50 million points. The reason is that learning over a large dataset can be costly since the learning process requires a number of epochs, and the dataset will be fully scanned in each epoch. Models like Feedforward Neural Network (FFN) can be slow if learning over large datasets.

In this thesis, we are going to address the efficiency issues in spatial index learning, which is challenging due to the following reasons.

- **Challenge 1**: *There is no comprehensive basis that supports testing and evaluating learned spatial indices against traditional indices for large real-world datasets.* Using learning-based techniques to build spatial indices is inspired by the key idea of learned indices, which learn a model based on an ordered dataset to predict the data position in storage. While such an ordering is natural for data that can be ordered by a single parameter, there is no single universal technique that takes multi-dimensional data and returns a linearly ordered dataset while ensuring a high query performance.

  Recently, two learned spatial indices, *ZM* index [103] and RSMI [84], use the Space-filling Curves (SFC), e.g., the Z-curve (ZC) [78] and Hilbert Curve (HC) [29] to order spatial data linearly and thus enabled us to learn spatial indices. The experimental results of ZM and RSMI show that they both can improve the query performance compared to tree-based indices like R-trees.

  However, there is a range of traditional indices that outperform R-trees, and there is no comprehensive comparison as of now that shows to what extent ZM and RSMI can outperform these traditional indices in terms of index building cost query time, and index size. This is a major drawback. Without such a study, it is unclear if learned indices can indeed outperform traditional indices for large spatial datasets under realistic settings. As there is no open-source code available, all learned spatial index structures were re-implemented from scratch. Our work provides a comprehensive basis for future studies that wish to compare learned spatial indices against traditional indices. It also offers a thorough analysis of the advantages and disadvantages of learned spatial indices.

- **Challenge 2**: *The current techniques for learning spatial indices incur high computational cost.* Learned indices offer a number of benefits. Most notably, they can deliver better query performance than traditional indices over spatial data. A key reason is that they can leverage the high prediction accuracy and efficiency of modern deep learning techniques. Recent studies demonstrate that we achieve query-efficient learned indices for spatial data by partitioning and subsequently transforming spatial data to one-dimensional values, after which existing techniques can be applied.

  However, despite the advantages in terms of query performance, learned spatial indices also have a major drawback: building and rebuilding come at a high computational cost. There is little research that proposes algorithms to efficiently build learned spatial indices, which in turn limits their practicality. Generally, deep learning techniques require a long time frame to train a model, especially over very large datasets, e.g., datasets that have over 50 million entries. The training process includes multiple iterations of full scans over the whole dataset, which is time-consuming even with modern Central Processing Unit (CPU) and Graphics Processing Unit (GPU) architectures. Reducing index build times so that they are comparable with traditional spatial indices is one of the major challenges within the database community.

- **Challenge 3**: *Applying classical machine learning techniques such as model reuse for learning spatial indices for new datasets is difficult.* As discussed in Challenge 2, learning spatial indices is slow when we simply learn an index from scratch. To speed up the learning process, a common idea is to use pre-trained models. Ideally, we could build a generic model for every spatial dataset as this has the potential to provide significant savings in terms of index learning time. However, in practice, a model can often only represent a single specific data distribution. While a suboptimal model may still work, the expected retrieval times are high. This is because the prediction accuracy of such an index is generally low, and we may end up with large prediction errors when indexing a different dataset. Therefore, reusing existing learned spatial indices is difficult when considering the variation in the distribution of different datasets.

  To maintain high accuracy in query processing over different data distributions, we need to study the generation of pre-trained models to make sure that they are

generic enough for the variety of different data distributions. However, the generation of pre-trained models for learned indices is still challenging. Furthermore, adapting pre-trained models to different real datasets and aligning the distribution gap between them is also difficult.

- **Challenge 4**: *The optimization of traditional spatial indices through learning-based techniques is computationally hard and lacks a principled approach for learning an optimized spatial index.* While learned spatial indices have shown strong query performance, their structure and query algorithms differ drastically from the traditional indices, which are well supported by off-the-shelf database systems.

  To reduce the additional overhead of replacing traditional indices with learned spatial indices, we consider another approach and apply learning-based techniques to optimize the structure of traditional spatial indices [35, 108], which can avoid the extra cost of learning an index. A common pattern is to transform the index-building process into an optimization problem, e.g., using RL to tune an R-tree based on query workloads. This is because RL offers a way to automatically learn and improve an index based on feedback and interactions with the index structure.

  Currently, not all the indices are able to be optimized as there is no existing principled strategy to formulate the learning procedure. A key idea to learn indices is using SFCs, which is used to map spatial data points into one-dimension values. SFCs have two desirable properties: (1) they are easy to learn because they all have distinctive patterns and (2) SFCs are all based on orderings, which enables an efficient enumeration of all SFC-based indices.

  In recent years, SFC-based indices have been widely studied and used in both academia and industry [2, 85], and ZC as well as HC are two of the most prominent SFCs. However, a *given* SFC cannot be optimized because they are functions that are simply applied to a given space and lead to a fixed enumeration of all data points in that space. ZC and HC are only two of the possible candidates for an SFC, and what can be optimized, is finding an optimal SFC for a given dataset and query workloads. An SFC is optimal if it requires minimal data access to implement query workloads. We will show later that different query workloads (e.g., different range queries) can lead to higher or lower numbers of index accesses.

  Thus, our aim is to find the optimal SFC that orders the given spatial data points so that we improve query performance. Since the candidate space of possible SFCs

is huge, enumerating the whole solution space has a prohibitive cost. Therefore, a key challenge is to make SFCs learnable without relying on a brute-force approach.

In addition, learning to find the optimal SFC, e.g., using RL, requires efficient feedback (i.e., reward function) to measure the intermediate result. For example, the learning cost can be expensive if we build real indices and execute queries to compute the rewards. However, there is no efficient measurement, which preempts efficient learning of SFCs.

1.3 Research Questions

We propose four research questions about spatial index learning in this section, aiming to ascertain the advantages and disadvantages of learned spatial indices and improve the efficiency of spatial index learning.

Previously, we discussed four challenges in spatial index learning in Section 1.2 where Challenge 1 is about the experimental study of learned spatial indices, and the others are about the efficiency in index learning. As these challenges are not independent of each other, we summarize the last three challenges into two types: (1) *learning an index model to replace traditional index structures* (Challenge 2 and Challenge 3), and (2) *learning to optimize the structure of traditional index structures* (Challenge 4).

When learning a spatial index to replace a traditional index structure, the key problem is that the index building requires a long time to train a model, e.g., FFNs [53]. This is because a large number of model parameters are inside a model, and the learning process relies on multiple iterations to fully scan the dataset. Replacing the FFNs with linear models can decrease the number of parameters, which saves training time, but linear models can suffer from underfitting, which can lead to subpar query performance. Thus, building efficiency issues need to be addressed.

When learning to optimize the structure of traditional index structures, the cost of the learning process is also an obstacle. This is because most of the techniques use a time-consuming reward function, which needs constant building of the indices to measure the improvement of query performance. Currently, there is no work that can efficiently compute an optimal SFC-based index. There are two reasons: (1) there is no efficient method to measure SFCs, and (2) the candidate space of SFCs is huge, which prohibits

the brute-force approach. These issues show that learning to optimize SFC-based indices is also worth to be resolved.

In this thesis, we will investigate four key research questions in learning spatial indices.

- **Research Question 1**: *To what extent can learned spatial indices outperform traditional spatial indices?* The potential of learned indices has been experimentally studied [50] over a series of indices [25, 30, 49, 53, 95]. For spatial data, specific learned spatial indices have been proposed to enhance query processing efficiencies. Compared with traditional spatial indices, e.g., R-trees, learning-based techniques have shown superior performance in point and range queries [84, 103].

  However, a significant challenge is the absence of a comprehensive experimental study, which should be able to thoroughly evaluate traditional spatial indices alongside learned spatial indices. In addition, the lack of available source code hinders investigations into the implementation details of learned spatial indices. The limited insight into their potential drawbacks further complicates matters. Therefore, the efficiency of learned spatial indices in query processing remains difficult to ascertain.

  To shed light on these issues, we conduct an experimental study on the performance improvements offered by learned spatial indices over three widely used query methods. Additionally, the thesis seeks to provide clarity in relation to potential disadvantages associated with learned spatial indices, such as the cost of index building. By addressing these aspects, a more comprehensive understanding of the capabilities and limitations of learned spatial indices can be gained.

- **Research Question 2**: *How can spatial indices be learned most efficiently?* After the investigation of learned spatial indices and observing their high query efficiency, there are several issues shown in the building process. For example, the index-building time can be about two orders of magnitude longer than traditional spatial indices [58, 84, 103]. The building cost consists mostly of model training, which is time-consuming but unavoidable.

  Furthermore, the query performance of learned spatial indices suffers from data insertions. When new data points are inserted, the data distribution is changed, whereas the models of RSMI are built for the original data distribution. This results in larger prediction errors and increases the search time. One straightforward

solution would be to frequently retrain the index, but this approach is prohibitively expensive due to the high index-building cost.

Currently, there is no universal method to efficiently build learned spatial indices and maintain query performance at the same time. Sampling is a common method to shrink the dataset, which can improve the building efficiency as the model training cost is decreased. However, the downside is that the query performance will be jeopardized. Moreover, different learned spatial indices require different strategies to adapt sampled datasets due to their inconsistent index structures. Thus, a unified method is missing and an ad-hoc adaption is required.

Therefore, it is worthwhile to explore: (1) how to sample a small dataset and maintain the query performance; (2) how to use a unified framework for efficiently learning spatial indices. To address these issues, we propose to learn indices by sampling representative datasets, which can significantly reduce training costs while maintaining high query performance. We also design a unified framework to learn spatial indices where the sampling is decoupled from index learning.

- **Research Question 3**: *How can we effectively design pre-trained models that reduce building costs while maintaining query performance?* The main idea to address Research Question 3 is inspired by a broader research field known as *domain adaptation* [11]. It involves reusing a pre-trained model, denoted as $\mathcal{M}_S$, which is trained on a known source dataset $\mathcal{D}_S$, and adapting it for a new target dataset $\mathcal{D}_T$. After fine-tuning $\mathcal{M}_S$ on $\mathcal{D}_T$, the need of training a new model from scratch on $\mathcal{D}_T$ is avoided, thus reducing the overall computational cost.

  Specifically, we will address Research Question 3 by building on our approach for Research Question 2. To effectively build a smaller, representative training set, we use certain methods in addressing Research Question 2. One of these methods is suitable (called Model Reuse (MR)) in effectively reducing the building cost.

  The overall idea is to first generate representative synthetic datasets to simulate real datasets, then train models for these datasets, afterwards we adapt these pre-trained models to effectively build an index and finally fine-tune these models for optimal query performance.

  However, as discussed in Challenge 2, there are certain challenges associated with utilizing pre-trained models and fine-tuning. These challenges include generating

appropriate pre-trained models and adapting pre-trained models over different datasets. To address these challenges, the presented research in this thesis will present novel techniques for the generation of pre-trained models, model adaption, and data range alignment.

- **Research Question 4**: *How can we efficiently learn a space-filling curve-based index that ensures optimal query performance?* SFCs are important for spatial data indexing as they have been used for both traditional and learned spatial indices, where they are used to order spatial data. The performance of SFC-supported indices is highly dependent on the ordering of SFCs. Thus, learning an optimal SFC to index spatial data is a feasible way to bring better spatial indexing.

  As discussed in Challenge 4, however, learning to optimize SFCs is challenging because the candidate space of possible SFCs is huge, and efficient SFC measurement is not available. An existing approach QUery-Intensive Linearization Tolerating data Skew (QUILTS) [73] uses a heuristic method to design an SFC and measures the query cost with query workloads against the designed SFC. However, the cost estimation is not efficient since QUILTS measures the performance of each candidate SFC overall query workloads, which can cost hours over 1,000 queries. This shortcoming hinders the optimality of QUILTS.

  To address this challenge, we focus on developing an efficient cost estimation algorithm that can accurately measure SFCs with low computational time complexity. This can be achieved by pre-storing all the fixed intermediate results in a look-up table, which helps avoid high computation costs. This algorithm should be adaptable to the learning process and keep measuring different SFCs. In addition, we will propose a strategy to make SFCs learnable because they originally follow a static pattern and should be dynamically changed in the learning process.

1.4 Our Contributions

We investigate four important research questions in spatial index learning.

- **Contribution 1.** *Efficiency of Spatial Index learning: An Empirical Study.* We start with an empirical study on recently proposed learned spatial indices named

RSMI [84] and ZM [103], to confirm the effectiveness of such indices to achieve a high query efficiency, and study their limitations in terms of index building and update efficiency. We test RSMI and ZM against traditional spatial indices over both real and synthetic datasets with more than 100 million points. Our study offers one of the first sets of empirical evidence on the high effectiveness and efficiency of using learned spatial indices for query processing, which outperform traditional spatial indices such as the R-trees for more than an order of magnitude. Meanwhile, learned spatial indices like RSMI are constrained by their build times, which can be two orders of magnitude slower than traditional indices. This motivates the thesis to study highly efficient techniques for machine learning-based spatial index optimization.

- **Contribution 2.** *A Framework for Efficiently Learning Spatial Indices.* To advance the practicality of learned spatial indices, we propose a system named ELSI that enables the efficient building and rebuilding of a class of learned spatial indices that follow two simple design principles. Our core idea is to reduce the model (re-)building times by engineering reduced training sets that preserve key data distribution patterns. ELSI encompasses a suite of methods for constructing small and distribution-preserving training sets from input datasets. Further, given an input dataset, ELSI can adaptively select a method that produces a learned index with high query efficiency. Experiments on both synthetic and real datasets of 100+ million points show that ELSI can reduce the build times of four different learned spatial indices consistently (by up to two orders of magnitude) without jeopardizing query efficiency.

- **Contribution 3.** *Efficient Index Learning via Model Reuse and Fine-tuning.* We further propose a model reuse and fine-tuning technique to save index build times and avoid training from scratch, inspired by the aforementioned *domain adaptation* technique which is a classic machine learning technique to reduce model training times when working on a new domain. The reused models are pre-trained over synthetic datasets, which are generated based on a heuristic method. To reuse the pre-trained models, we use the earth mover's distance to measure the similarity between the target dataset $\mathcal{D}_T$ and the synthetic datasets. Extensive experiments on synthetic and real datasets show that model reuse and fine-tuning can accelerate the building of learned spatial indices by 30.4% and improve lookup efficiency by

up to 24.4% on real datasets and 22.5% on skewed synthetic datasets. We further show how this technique can be extended to spatial data. Experimental results show that, using model reuse, the time to build a learned spatial index can be reduced by one or two orders of magnitude, while the lookup efficiency of the resultant learned spatial indices can be improved by up to 13%.

- **Contribution 4.** *Efficient Cost Modeling for Spatial Index Learning.* Our last contribution considers optimizing the structure of a traditional spatial index – the SFC-based indices. We propose efficient query cost estimation techniques for SFC-based indices that enable both *online* and *offline* query cost estimation for SFC-based indices. Given a query and a set of available SFC indices, our techniques support the online selection of the most efficient SFC index in a minimally intrusive manner, enabling practical application in database systems that support different SFC indices. When a query workload is given beforehand, our techniques leverage reinforcement learning to enable the selection of an SFC-based index for processing the workload efficiently. Experiments show that our cost estimation is over an order of magnitude faster than naive methods. Moreover, the query performance of the SFC-based indices that adopt learned SFCs is better than using ZCs and HCs.

1.5 Thesis Organization

The rest of the thesis is organized as follows.

- In Chapter 2, we review both traditional and learned indices, which will be further discussed based on dimensionality, i.e., one-dimensional indices and spatial indices. We also discuss the application of space-filling curves in spatial data indexing, as well as datasets, query methods, and measurements that are commonly used in the literature on learned spatial indices.

- In Chapter 3, we study the empirical performance of learned spatial indices, which are compared with traditional spatial indices. We show that learned spatial indices are efficient in query processing, however, they suffer from a high building cost.

  This chapter is based on the experimental section of the following paper, to which I have made the majority of the contributions: Jianzhong Qi, Guanli Liu, Christian

S. Jensen, Lars Kulik. Effectively Learning Spatial Indices, *Proceedings of the VLDB Endowment* (**PVLDB**) 2020.

- In Chapter 4, we propose a system called *ELSI* that enables the efficient building and rebuilding of a class of learned spatial indices.

  This work has been published in the following paper: Guanli Liu, Lars Kulik, Christian S. Jensen, James Bailey, Jianzhong Qi. Efficiently Learning Spatial Indices, *IEEE International Conference on Data Engineering* (**ICDE**) 2023.

- In Chapter 5, we use pre-trained models and fine-tune the models to build learned indices. It achieves competitive query performance to learned indices that are built from scratch, whereas our building cost is much cheaper.

  This work has been published in the following paper: Guanli Liu, Jianzhong Qi, Lars Kulik, Kazuya Soga, Renata Borovica-Gajic, Benjamin I. P. Rubinstein. Efficient Index Learning via Model Reuse and Fine-tuning, *International Workshop on Databases and Machine Learning* (**DBML**) 2023, held together with ICDE 2023.

- In Chapter 6, we propose efficient query cost estimation algorithms for space-filling curves that are orders of magnitude faster than naive methods, thus enabling online SFC selection and offline SFC learning.

  This work is in preparation for submission to *International Conference on Management of Data* (**SIGMOD**) 2024.

- In Chapter 7, we summarize our contributions and discuss future directions.

Chapter 2

Related Work

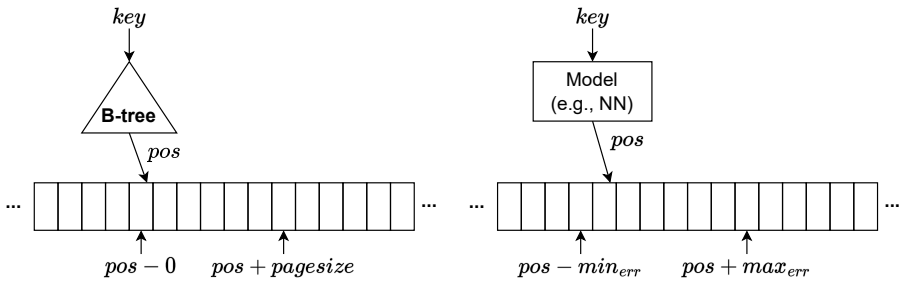
Traditional indices such as B-trees [8] and R-trees [37] have been widely used in database systems for decades [28, 64, 81, 82]. Recently, machine learning-based techniques have been introduced to database indexing, which have attracted strong interests in the database community, resulting in the so-called *learned indices* [53]. A number of learned indices have been built and reported attractive query performance [25, 30, 39, 49, 53, 58, 70, 84].

In this chapter, we review studies on both traditional and learned indices to show how learned indices improve query performance. We cover indices for both one-dimensional and spatial (multidimensional) data. We also discuss SFC-related techniques as SFCs are widely used in both traditional and learned spatial indices. Finally, we summarize the query types and datasets commonly used in the empirical studies of learned indices.

2.1 Overview

Indices are a core component of database systems to enable efficient query processing. For traditional indices such as the B-trees which have a hierarchical structure, a tree traversal is typically needed to locate the leaf nodes containing the data records that satisfy a given query. A binary search may follow to locate the final query target from the leaf nodes (cf. Figure 2.1(a)).

Learned indices take a function invocation-based query paradigm. They train machine learning models (i.e., index models/functions) to predict the location of the data record



(A) A B-tree (B) A learned index

FIGURE 2.1: A B-tree vs. a learned index.

relevant to a given search key directly. Due to the inaccurate nature of machine learning models, a search in the range of $[pos - min_{err}, pos + max_{err}]$ is often needed to locate the final query result, where *pos* is the predicted location by the index models, while min_{err} and max_{err} are bounds of the prediction errors (cf. Figure 2.1(b)).

The advantage of learned indices comes from multiple factors. It replaces scans (or binary search) over a tree node in the index traversal process with function invocations which can be much faster. Meanwhile, an index model can provide predictions with small (or no) errors for a data range, which is typically much larger than what is covered by a tree node of a traditional index. This means a much lower hierarchy if multiple index models are needed in a learned index structure, e.g., RMI [53].

In what follows, we provide a review of traditional and learned indices in Section 2.2 and Section 2.3, respectively. We focus on the learned indices as there are many surveys on the traditional indices already [15, 32]. To show the evolutionary history of the index structures reviewed, we plot them in chronological order in Figure 2.2. Here, *traditional indices* refer to index structures built by deterministic or heuristic algorithms, and *learned indices* refer to those built with machine learning-based techniques. For the traditional indices, we only show the original versions without enumerating their variants. For the learned indices, we show the representative ones.

2.2 Traditional Indices

We discuss one-dimensional indices in Section 2.2.1 and spatial indices in Section 2.2.2.

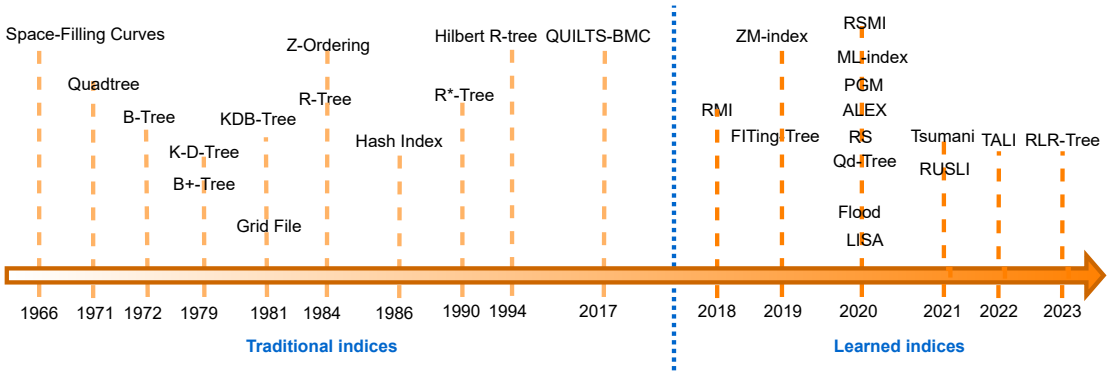


FIGURE 2.2: Index structures reviewed in this chapter

2.2.1 One-dimensional Indices

A one-dimensional index is a fundamental data structure that enables efficient data access operations over database tables, using search keys in a single dimension. Traditional one-dimensional indices can be classified into four main categories: *tree-based*, *hash-based*, *bitmap-based*, and *trie-based* indices.

Tree-based indices employ a hierarchical structure to store index entries and facilitate efficient search, insertion, and deletion operations. The most commonly used tree-based one-dimensional indices are arguably the B-tree family [6, 12, 20, 86, 95, 104], where the B^+-trees and the B^*-trees are among the most well-known variants.

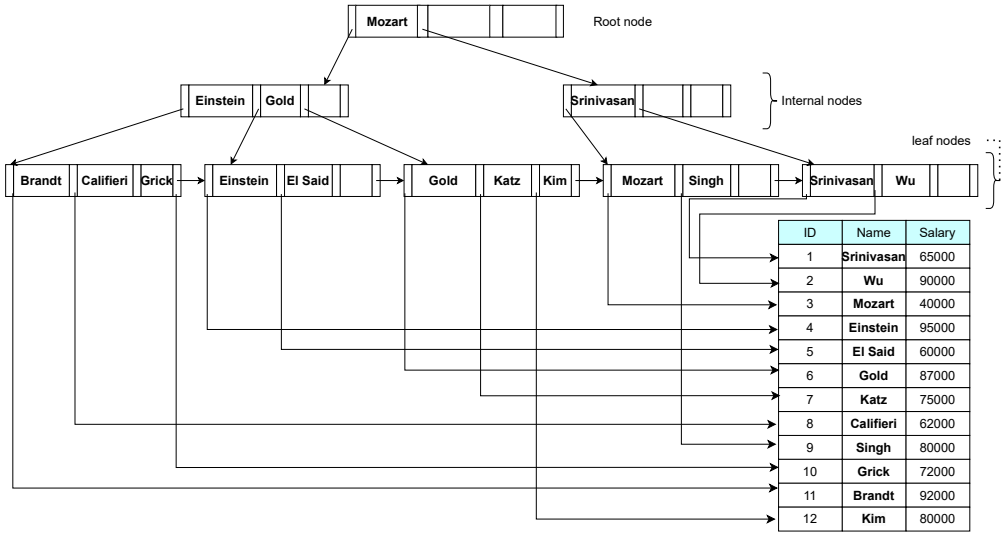


FIGURE 2.3: B^+-tree [94]

The query process of a tree-based index usually requires tree traversal. Take Figure 2.3 as an example, which plots a B^+-tree that indexes the *name* column of some database

table. Each node in this tree has a capacity $B = 4$, i.e., there are up to four entries in each tree node. When querying for a key "Kim", the search starts from the root node. The query key is compared with (using a binary search) the key values stored in the root node to find the first value greater than or equal to the query key, which is "Mozart" as shown in the figure. The node pointed to by the pointer on the left of the entry"Mozart" (i.e., the left internal node) is the next to be queried. The same tree node querying procedure repeats recursively until a leaf node is reached. By further examining the key values stored in the leaf node, we can either locate the data record corresponding to the query key "Kim", or determine that no relevant data record exists. Typically, $O(\log_B n)$ nodes are accessed in the process, given a database table (i.e., a dataset) with n records.

Insertion and deletion operations follow a similar procedure, but with additional node splitting or merging operations to maintain a compact and balanced tree structure.

Hash-based indices leverage hash functions to map data items into buckets of a *hash table* using their key values. Given a query key, it is also mapped to a bucket using the same hash functions, which are then compared with the data key values stored in that bucket to locate the matched key and data record.

Hash-based indices have been implemented in DBMSs, e.g., PostgreSQL [40], where the hash index supports one-dimensional indexing only. Each hash index entry stores just a 4-byte hash value, which makes the resultant index much smaller than B-trees, when the raw index keys are of a large size, e.g., Universally Unique IDentifiers (UUIDs) or Uniform Resource Locators (URLs).

One significant advantage of hash-based indices is the speed of query processing in ideal case. The computational cost of the hash functions is low, which can directly lead to the data associated to a specific key. However, when there is hash collision, i.e., when different keys are hashed to the same bucket, the search cost can increase substantially, e.g., degrading to a scan over all data records sharing the bucket. In addition, hash-based suffer in range query processing and do not typically preserve the order between the data records.

Bitmap-based indices use bitmaps to store the index entries [17]. Each unique index key value is assigned a corresponding bit in the bitmap. Such a structure can be highly

efficient to process exact-match queries (i.e., a bit lookup over the bitmap). However, its insertion operations can have a high rewrite cost for frequently updated tables. When a new data record is added with a new key value that has not appeared before, the bitmap index may need to be fully updated. For example, consider a column named *month* in a database table, and there are 11 different values, i.e., from "January" to "November" (and not "December"). If a bitmap index is built over the *month* column, only 11 bits are needed, e.g., 00000000001 will represent "January" and 10000000000 will represent "November". When a new data record arrives with a value of "December" in the column *month*, the bitmap index needs to be reconstructed to allow for this new value.

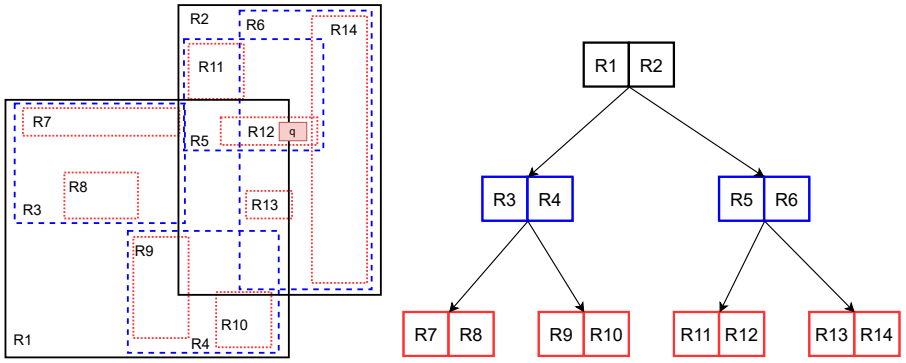
Trie-based indices, also known as prefix trees or trie indices, where the keys are often strings [91]. In a trie-based index, each node represents a character, while a key value corresponds to a sequence of tree nodes whose sequence of characters equals the key value. To search for a given query key (i.e., a string), a sequence of nodes whose corresponding characters match the query key is visited (or the query key does not exist in the database table). The node corresponding to the last character in the query key stores the data value corresponding to the key, e.g., a word frequency. Insertion and deletion operations over a trie-based index involve adding and removing nodes from the tree as needed to maintain the required tree structure. A common issue of tries-based indices is their high space costs, especially when there is a large number of unique keys which do not share common prefixes. In such a case, each parent node in tries can have many child nodes and hence requiring a large storage space.

2.2.2 Spatial Indices

Spatial indices, such as *R-trees* [37], *Grid-files* [72], and *kd-trees* [13] are proposed to enable efficient processing of queries over spatial data, such as point queries (i.e., exact-match queries), range queries (a.k.a. window queries), and kNN queries (i.e., similarity queries). The core design goals of these indices (and the one-dimensional indices discussed above) are the same, i.e., to enable efficient pruning of the search space at query processing. However, their pruning strategies and hence construction algorithms are quite different, based on which they are categorized into three types: *data partition-based*, *space partition-based*, and *mapping-based* indices.

Data partition-based indices, such as *R-trees* [9, 10, 37, 93], employ a clustering-based approach to partition data. The underlying principle of R-trees is to group spatially close data objects together and enclose them within a Minimum Bounding Rectangle (MBR). When processing queries on an R-tree, if the query object does not satisfy the given query predicate (e.g., intersecting for range queries) with the MBR of a tree node, that tree node can be safely pruned from further consideration.

In Figure 2.4, we plot an R-tree over a dataset of eight data objects $R7$ to $R14$. Each node in this tree holds up to two entries (i.e., MBRs and node pointers/data records). Entries $R3$ to $R6$ correspond to MBRs that bound the leaf nodes formed by the data objects, while $R1$ and $R2$ further bound the internal nodes formed by $R3$ to $R6$. There is also a range query q in Figure 2.4(a), which aims to fetch all the data objects overlapping with q. A top-down R-tree traversal is done to process the query. At start, the two MBRs $R1$ and $R2$ (the two solid blue rectangles) in the root node are examined, which both overlap with q. Thus, both child nodes of the root node need to be examined, and MBRs $R3$ to $R6$ (the four dashed blue rectangles) are compared with q. Only $R5$ overlaps with q, and its corresponding child node is visited. Data objects $R11$ and $R12$ (the dotted red rectangles) are then compared with q, and $R12$ is returned as the query answer as it is the only object overlapping with q. As this example reveals, an issue with the R-tree structures is that there are overlaps between the MBRs at the same level of the tree. Multiple branches need to be visited during query processing, which negatively impacts the query efficiency.



(A) MBRs in data space (B) R-tree structure

FIGURE 2.4: An R-tree example

A common approach to build an R-tree is by dynamic object insertions. The resultant R-tree structures depend on the order of data insertion, which may form R-trees with substantial MBR overlaps.

An alternative approach bulk-loads R-trees in a bottom-up manner, where data objects are grouped to form the leaf nodes first. Most bottom-up methods [5, 34, 47, 56, 89] require some sorted order over the data objects, with the goal to minimize MBR overlaps among the resultant leaf nodes, e.g., sorting by the x-coordinates [89] of the data objects. Every B consecutive data objects in the sorted order are simply packed into a leaf node. Subsequently, a recursive packing over the resultant nodes is done until only a node is formed, which becomes the tree root.

Space partition-based indices recursively partition the space in which the data is embedded until the data objects in a partition fit into an index node, where *kd-trees* [13], *quadtrees* [31], and *Grid-files* [72] are typical examples.

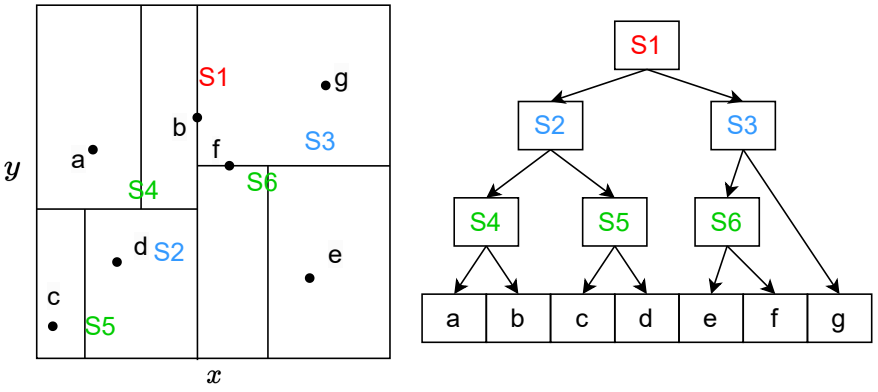


FIGURE 2.5: A kd-tree example

A kd-tree (i.e., k-dimensional tree) recursively partitions the data space on each data dimension, until there is only one point in each sub-space. This partitioning of the data space is based on the median i-coordinate along dimension i among all points in the sub-space, after they are sorted.

For example, in Figure 2.5, there are seven data points (i.e., a to g) in a data space $S1$, i.e., the full data space. Space $S1$ forms the root of the kd-tree. The first round of partitioning is done on dimension x, where the x-coordinate of b is the median. Then, $S2$ and $S3$ are the two sub-spaces formed by a vertical line through b. All points with an x-coordinate less than or equal to that of b (i.e., a, b, c, d) are assigned to partition $S2$, while the rest of the points (i.e., e, f, g) are assigned to partition $S3$. Then, $S2$ and $S3$ are partitioned by dimension y, respectively. Partition $S2$ is divided into $S4$ and $S5$, where the partition line the based on the median of y-dimension value. Next, $S3$ is divided into $S6$ and g. The process repeats until there is only one point per partition.

As the partitions formed by a kd-tree are non-overlapping, they may offer more efficient pruning at query processing. However, they can become unbalanced after a series of insertions at a small region. The cost of maintaining a balanced kd-tree is high, while query processing over a skewed kd-tree also has a high cost.

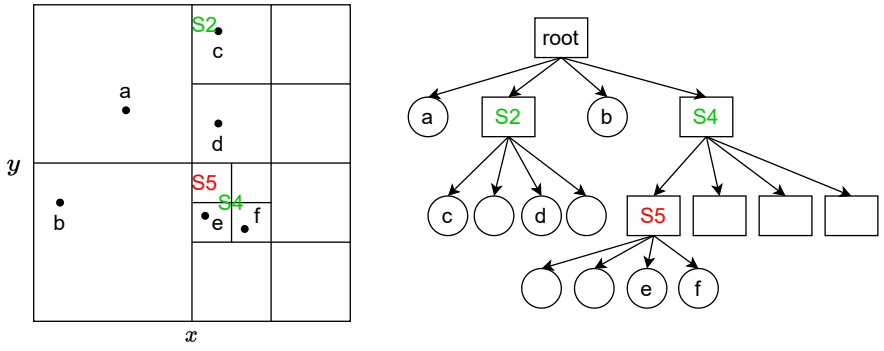


FIGURE 2.6: A quadtree example

Instead of binary partitions as done in kd-trees, *quadtrees* partition a space into four quadrants of equal size recursively, until the number of points in a partition does not exceed some threshold (e.g., node size B). Thus, for a quadtree, a node has either four child nodes if it is partitioned or no child nodes at all. When inserting a point, a quadtree starts from the root node and recursively finds the quadrants that enclose the inserted point until a leaf node is reached. If the leaf node is empty, the insertion is completed by adding the point into the leaf node. Otherwise, the leaf node may need further partitioning to make room for the inserted point, following the index building procedure. Quadtrees are widely used for indexing point data [81, 82], attributing to their simple structure.

Figure 2.6 shows a quadtree over six points a to f. The root node of the tree indexes the four quadrants of the full data space. The two quadrants on the left each contains a point (i.e., a and b), and no further partitioning is needed. The top right quadrant contains two points c and d, which take another level of partitioning to be placed into separate quadrants (assuming $B = 1$). The bottom right quadrant also contains two points (e and f), which take two more levels of partitioning as these two points are closer. As the example shows, an issue with the quadtrees is that they may create a highly unbalanced structure if the data distribution is skewed.

A *Grid-file* partitions the data space with a grid such that each grid cell stores a pointer pointing to the storage space of the data points falling into the cell. Grid-files provide an

initial attempt to map spatial data into one-dimensional values. After partitioning the two-dimensional data space into a series of cells, each cell in the space can be numbered with a one-dimensional value, while it corresponds to a range of x- and y-coordinates, forming a relationship between a one-dimension value and a two-dimensional range. All the points that fall in a cell can be mapped to the same one-dimensional value corresponding to the cell.

Grid-files are typically built with a regular grid where equi-sized partitioning is done on each dimension. Figure 2.7 shows an example with a 3×3 grid. The mapping between a data point and the grid cell enclosing the point can be easily calculated based on the point coordinates.

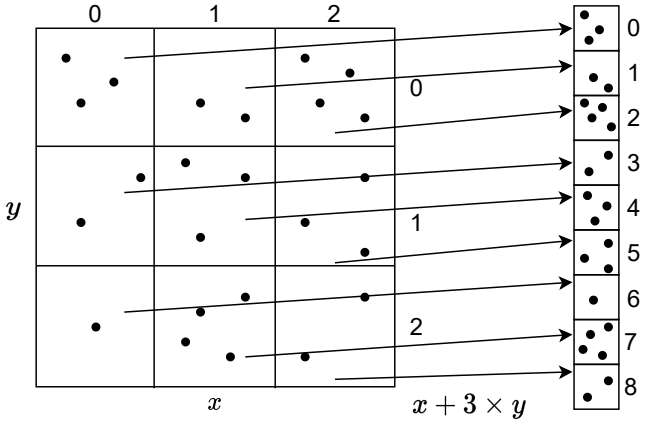


FIGURE 2.7: A Grid-file example

Mapping-based indices map spatial data points to one-dimensional values. This mapping enables an ordering of the data points based on their mapped values, which in turn enables using one-dimensional indices to organize the data [7, 14, 45]. Different mapping and ordering schemes have been proposed in the literature. A simple strategy is to just take the coordinates from one of the data dimensions. There are also more sophisticated schemes such as the space-filling curves (SFCs), iDistance [44], and the pyramid technique [14]. SFCs are a core concept in this thesis, and we will detail it in its own section (Section 2.4).

The iDistance technique is proposed to facilitate kNN query processing. Its key idea is to select a set of reference points in a spatial data space, where each reference point is derived from some partition (or a data cluster). Each data point is then mapped to a one-dimensional value by computing its distance (plus an offset to differentiate points

in different partitions) to the reference point that shares the same partition of the data point. The mapped values are then indexed with a one-dimensional index, e.g., a B-tree, for query processing.

2.3 Learned Indices

The core intuition behind learn indices is that a database index can be represented as a function f that maps a given query key *key* to the storage address *pos* of the corresponding data record. Such a function f can be learned via a machine learning model (i.e., an index model). When such a function is learned, exact-match queries can be processed through a function invocation in a constant time (in an ideal case), without having to perform tree traversals.

Earlier, we showed with Figure 2.1 that B-trees can be seen as index models (i.e., index function f). When given a query key, B-tree requires a tree traverse to local the leaf node relevant to the query key. Then, a search algorithm (e.g., binary search) is run to locate the queried data record in the node. Query processing with machine learning models follows a similar paradigm, except that function invocations replace tree traversals. This is because typically a hierarchy of machine learning functions is needed to cope with the large size of a real dataset, when each individual machine learning model has a limited learning capacity.

A seminal study [53] uses an FFN to learn function f for indexing one-dimensional data. This study leverages the capability of FFN to learn complex non-linear functions. The learning is based on storing the data records in ascending order of their (one-dimensional) key values. The learned function f is then essentially an approximation of the Cumulative Distribution Function (CDF) of the dataset to be indexed.

As Figure 2.8 shows, the CDF of a dataset reflects the distribution of the dataset. Given a value x as the input, the CDF returns the percentage of data records with key values less than or equal to x. Let the CDF of a dataset $\mathcal{D}$ of size n be $cdf_{\mathcal{D}}$. Then, the storage address of a data record with key value *key* is just $cdf_{\mathcal{D}}(key) \cdot n$, if the dataset is stored in ascending order of the data values. The target of index learning is then to learn a function f to fit the CDF of a dataset, and $f(key) \cdot n$ approximates the storage address relevant to query key *key* for query processing.

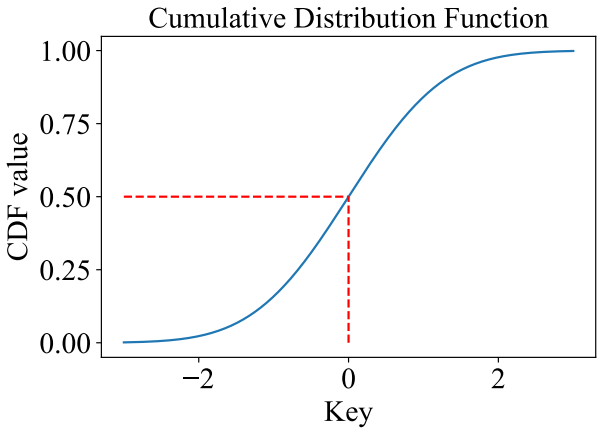


FIGURE 2.8: A cumulative distribution function (CDF) is a function that describes the probability that a random variable will take on a value less than or equal to a given value. This figure plots the CDF of a dataset that follows a normal distribution. When given a $key = 0$, which is the mean of all the keys in this dataset, the CDF value is 0.5 because half of the keys in the dataset are less than or equal to 0.5.

Given the inaccurate nature of machine learning models, studies have focused on reducing the prediction errors of the learned index function f over one-dimensional data. There are studies that extend the idea to spatial data, forming learned spatial indices. A core challenge to address in such indices is how to order spatial data objects, as there is no universal ordering of such objects. In what follows, we review both types of studies in Section 2.3.1 and Section 2.3.2, respectively.

2.3.1 Learned One-dimensional Indices

Learned indices have been proposed as an alternative for tree-based indices [53], hash-based indices [48], and Bloom filters [66]. We focus on the learned indices to replace tree-based indices, for their more general applicability.

The learned one-dimensional indices can be largely grouped into two categories, based on how the index models are formed: (1) *prediction-based* and (2) *interpolation-based*. Prediction-based indices learn the index models through machine learning techniques, such as linear regression models or neural networks. Empirical error bounds are derived after the index models have been computed (by invoking the models over every key value of the input dataset $\mathcal{D}$). Interpolation-based indices, on the other hand, take a single-pass scan over $\mathcal{D}$ and compute a series of index models. Each index model (e.g., a linear function) covers input data in a range until the prediction error of the function

exceeds a given threshold ϵ. Here, ϵ is a hyperparameter to balance the index size and query efficiency. We summarize some representative indices in Table 2.1.

TABLE 2.1: Representative learned one-dimensional indices

Type	Name	Build method	Updatable
Prediction-based	RMI [53]	Recursion + neural networks	No
	ALEX [25]	Linear regression + tree	Yes
	CDFShop [62]	RMI tuning	No
Interpolation-based	FITing-tree [33]	Linear interpolation + B-tree	Yes
	PGM [30]	Linear interpolation + recursion	Yes
	RadixSpline [49]	One pass + prefix table	No

Prediction-based learned indices. Recursive Model Index (RMI) [53] is a representative prediction-based learned index. It uses FFN to learn the index models, which are regression models. For small datasets, a single index model (and a single FFN) may already be sufficient. Larger datasets require multiple index models (and hence multiple FFNs) to form a hierarchy.

As shown in Figure 2.9, a single index model $f_{1.1}$ is computed at the top level, which learns only a rough estimation of the CDF of the dataset. The prediction value of the model given a query key *key*, $f_{1.1}(key)$, is used to determine the index model in the next level to be invoked for the prediction task. Heuristically, each index model in the RMI structure learns to approximate the CDF of a subset of the input dataset $\mathcal{D}$. By recursively invoking the index models, one can progressively narrow down the search for the query key.

In the RMI structure, the number of levels and the number of models in each level are hyper-parameters. When learning the index models, the L_2 loss is used to minimize the prediction errors. The building cost of the RMI structure depends on the number of models and the training cost of each single model. For example, using neural nets can take minutes per model, depending on the complexity. In the experimental part of RMI [53], the index building cost is not reported, but we find that neural nets with one and two hidden layers are studied. Therefore, the index building cost is expected to be high when using neural nets.

Adaptive Learned Index (ALEX) [25] further considers data updates. It uses a tree structure where there are two kinds of nodes which are *internal nodes* (i.e., non-leaf nodes) and *data nodes* (i.e., leaf nodes), respectively. The internal nodes are almost the

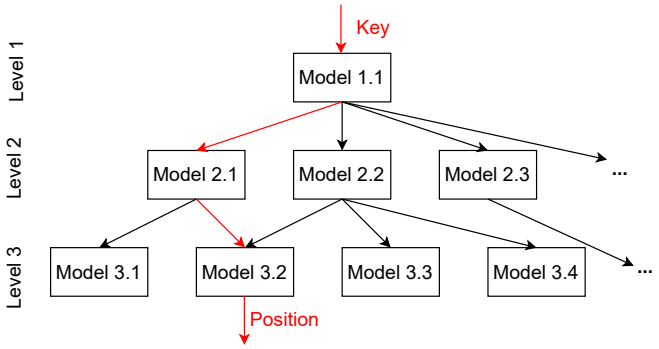


FIGURE 2.9: The RMI structure

same as those in RMI, but with a target to partition the key space and store the pointers to data nodes. The data nodes are like the leaf nodes of B^+-trees, which support data insertions. The difference is that the data nodes use a *gapped array*, with the intention to reduce the data shifting needed for data insertions. In a data node, queries and insertions are both handled by a linear regression-based index mode and a bitmap to record whether a key is in the gapped array. To avoid large internal nodes and data nodes, a *max node size* is imposed on both types of nodes to help decide when to split the nodes.

Compared with RMI, ALEX can handle updates with the support of data nodes. The use of the gapped array helps reduce data shifting for insertions and leads to better insertion performance than that of the B^+-trees.

For query processing, the internal nodes help narrow down the search space like those in RMI. When the query process reaches the data nodes, an exponential search in the data nodes is done for point queries. For range queries, they are replaced by point queries using the lower bounds of the query ranges. Once the lower bounds are located, a scan is done on the data nodes starting from the lower bounds until reaching the upper bounds of the query ranges.

For insertions, there are two cases:

1. Inserting into a non-full data node. The insertion position is first located with a point query. If the position is a gap, the new data record and key are simply placed at the position. Otherwise, a gap needs to be created by shifting the existing data records and keys by one position in the direction that is nearer to a gap. Then, the insertion can proceed with the created gap.

2. Inserting into a full data node. For a data node, when the ratio of gaps in the gapped array reaches a lower bound, the data node is considered "full", even though the data node may not be exactly 100% full, such that the gapped arrays do not get too full and trigger much data shifting at insertions.

   When a data node is full, an expansion operation is required if the expanded size is less than the max node size. The expansion process includes reconstructing the gapped arrays with enlarged sizes and scaling the model instead of retraining. When the number of keys in the gapped array is larger than the max node size, the liner model in the data node will need to be retrained.

Deletions are done straightforwardly by a lookup to find the location of the key and remove it and its payload in the gapped arrays, and by marking the record as deleted in the bitmap. If a data node reaches a lower bound of the ratio of keys in the gapped array, the node will be merged. Intra-node cost models are used to determine whether two data nodes should be merged together.

CDFShop [62] does not present another newly learned index but allows users to experiment with different RMI configurations and observe how these configurations affect the lookup latency and the memory footprint. Comparing with grid search, which is a simple yet inefficient strategy, CDFShop presents an automatic optimization technique for RMI, which can search for configurations that achieve strong lookup performance while minimizing the memory footprint. The automatic optimization consists of four steps: (1) Assume a set of M candidate models to be used to build an RMI of h levels. CDFShop focuses on the type of models used in each level, and so there are M^h model combinations (i.e., configurations) considered for the h levels. Index building and query processing are done for the RMI of the M^h configurations, and a subset of M' configurations with index size or query time that are predicted (by a model) to dominate those of the other configurations are kept. (2) The M' configurations kept are re-used on datasets that are different from those used in Step 1. Again, the configurations predicted to yield promising results are kept. (3) Indices are built and queries are now actually run with the configurations kept to identify the configurations that have high practical efficiency in either index size or query cost. (4) Step 3 is repeated, by varying the number of models in each RMI level, to further filter the candidate configurations. Finally, a smaller set of configurations are derived, which can be used to build Pareto-efficient RMIs.

Interpolation-based learned indices. FITing-tree [33] computes a Piecewise Linear Approximation (PLA) of the key value series of the input dataset $\mathcal{D}$, with a user-given maximum error bound $\epsilon \geq 1$. It aims to minimize the number of linear functions needed in the PLA to constrain the index size, while satisfying the maximum error bound.

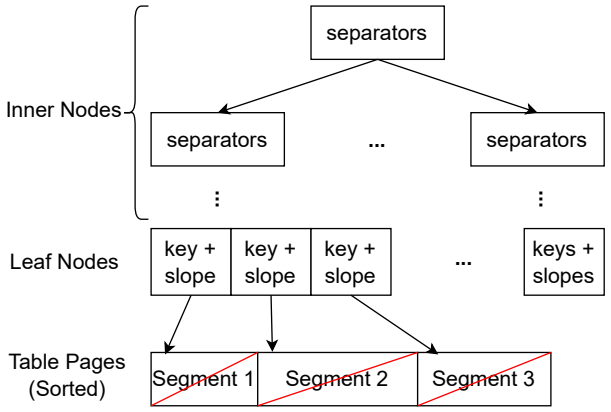


FIGURE 2.10: The FITing-tree structure

As shown in Figure 2.10, the bottom level of the FITing-tree structure is the leaf nodes. Each leaf node handle a corresponding table page, which stores the data records. The key and slope define a linear model to index the page, where the key is the first item in the table page. The inner nodes of a FITing-tree are identical to those of a B+tree (built over the start keys of the leaf nodes).

The point query and insertion procedures of the FITing-tree are the same as those of the B^+-tree at the inner node levels. At the leaf node level, the predicted location of the search key in the table page is calculated by the key and slope in the leaf node. Then, the true location of the query key is guaranteed to be in the range of ϵ-distance from the predicted location.

For insertions, the FITing-tree also requires additional work upon reaching a leaf node since a given error bound needs to be satisfied. Two insertion strategies are proposed. The first strategy performs an in-place insertion when there is an empty slot for the inserted data record. The second strategy uses a buffer of a fixed size to store the inserted data records in sorted order to support efficient query and merge operations. When the volume of the buffer reaches a given threshold, it will be combined with the data in the table pages to rebuild a new FITing-tree.

The Piecewise Geometric Model index (PGM-index) [30] enhances the FITing-tree by computing the minimum number of segments needed to cover the input dataset $\mathcal{D}$ given

a maximum error bound ϵ. It recursively packs data records into segments bottom up in its index-building process, i.e., the start keys of the data segments for a one-dimensional dataset, for which data segments are computed recursively, as shown in Figure 2.11. The cost of constructing the index primarily depends on the process of learning segments through linear interpolation. Thus, this cost is inherently constrained by ϵ, which decides the number of segments. However, a trade-off exists between query performance and construction cost as ϵ varies. Specifically, a smaller ϵ improves query efficiency by allowing for execution over a reduced range. Conversely, it inflates the index construction cost, as a smaller ϵ requires more segments and a more complex PGM-index structure.

To process point queries, the PGM-index starts from the top level to predict segment that covers the query key in the next level. This is refined by a binary search to ensure that the correct segment in the next level is located. The process continues recursively until the data segment covering the query key is identified to answer the query.

We use a query example to illustrate how the PGM-index works, with $\epsilon = 1$ and a query key $key = 76$. As shown in Figure 2.11, we get a predicted node number of 1 from the root node (level 0), given an input query key $key = 76$, i.e., $\lfloor sl_0^0 \times (key - 2) + ic_0^0 \rfloor = 1$ (the exact values of sl_0^0 and ic_0^0 are not of particular concern and hence are not specified here). Since $\epsilon = 1$, we also need to examine one node before and after node 1 at level 1. This means a binary search over an array $\{2, 31, 102\}$ to locate the largest number that is smaller than or equal to 76 (recall that the numbers are the start keys of a data segment), which is 31. We then visit the node with the start key 31 and compute another prediction using 31, sl_1^1, and ic_1^1 as the linear function parameters. This procedure is repeated, and finally, the query procedure terminates when 76 is found at a data segment.

For insertions, again there are two strategies for the PGM-index. The first follows that of an LSM-tree [75] and defines m static PGM-index structures with increasing sizes. The i-th PGM-index is built over a dataset $\mathcal{D}_i$ and should be either empty or contains $2^i - 1$ keys. When inserting a new key key, the first empty PGM-index, e.g., $\mathcal{D}_j$ will be the insertion target, and all the keys in the previous PGM-index structures will be merged to build $\mathcal{D}_j$. Then, a new PGM-index is built over $\mathcal{D}_j$.

The second strategy appends new keys to the end of the sorted dataset. If the new key key can be fitted by the last segment while maintaining the ϵ error guarantee,

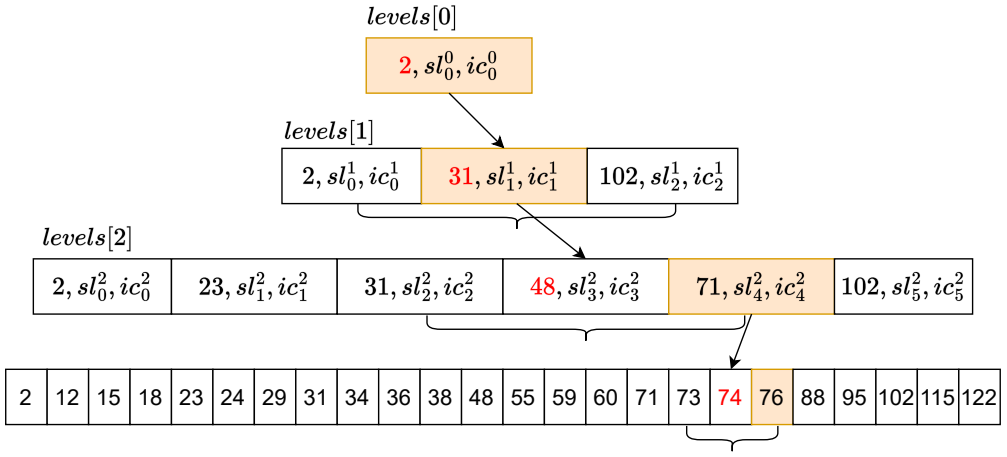


FIGURE 2.11: The PGM-index structure

the insertion process is completed. If the last segment cannot fit *key*, a new segment covering key *key* is formed. Then, the insertion process is propagated bottom-up in the PGM-index.

RadixSpline (RS-index) [49] is a single-level index that can be built in a single pass over sorted datasets. There are two key steps in building RS-index. First, all the data points are scanned to select the *spline points*. Every two consecutively selected spline points form an interpolation function f, where the location of any point between the two spline points can be predicted by f with a maximum error bound e. A set of spline points $\mathcal{D}'$ can be derived after scanning dataset $\mathcal{D}$. Second, a radix table is built over $\mathcal{D}'$ to help efficiently retrieve the pairs of consecutively selected spline points. The radix table consists of 2^r entries, where the b^{th} entry points to a spline point with a value of k. Here, b equals to the leftmost r bits of the binary form of k.

When using RS-index for point query processing, we first locate two consecutive pointers pointing to spline points that bound the query key in the radix table. When given a query key *key*, we take the leftmost r bits of the binary form of *key* can calculate b. We read the bth pointer in the radix table, which points to the spline point that is not larger than *key*. Then, we read the $(b+1)$th pointer in the radix table, which points to the spline point that is larger than *key*.

As shown in Figure 2.12, when querying key 47183, the binary form of the query key is **101**1100001001111. The $r = 3$ leftmost bits (i.e., 101) equals 5 (i.e., $b = 5$). Thus, the two spline points pointed to by the 5^{th} and the 6^{th} entries of the radix table are

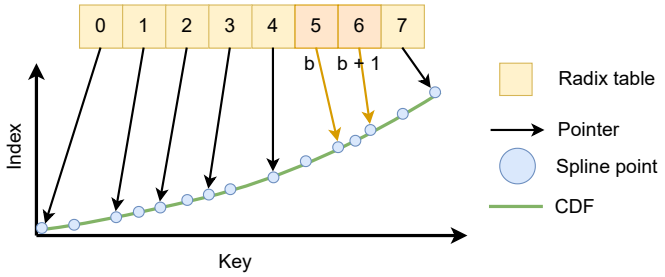


FIGURE 2.12: The Radix Spline index structure

fetched. Then, the interpolation function corresponding to these two spline points is used to predict the storage location of the query key.

2.3.2 Learned Spatial Indices

Almost all existing learned spatial indices are built based on model prediction instead of interpolation. We classify the learned spatial indices based on whether they are built based on known query workload, i.e., *query-aware* or *query-agnostic*.

The query-aware indices learn patterns of the queries or compute the query costs to guide index building. Decision making, e.g., deciding data partitions or tree splits, are based on the queries, i.e., to minimize the cost of the queries. Query-aware indices have a strong potential to obtain competitive query performance. However, their performance can also be sensitive to how close the actual query workload is to what is known during index building. The query-agnostic indices, on the other hand, do not assume any knowledge of the query distribution. They mostly are built based on data distribution. A common limitation shared by both types of indices is their capability to retain high query performance when the data distribution has been changed by data updates. We summarize representative learned spatial indices of both types in Table 2.2.

TABLE 2.2: Representative learned spatial indices

Type	Name	Build method	Updatable
Query-agnostic	ZM [103]	ZC + RMI	No
	ML-Index [23]	Clustering+iDistance+RMI	No
	LISA [58]	Partition + mapping + linear models	Yes
	RSMI [84]	ZC + Partition + FFNs	Yes
Query-aware	Flood [70]	RMI + Grid-file	No
	Qd-tree [108]	RL + kd-tree partition	No
	Tsunami [26]	Flood + Grid Tree	No
	RLR-Tree [35]	RL + R-tree	Yes

Query-agnostic learned spatial indices. Z-order Model (ZM) [103] is a direct extension of a learned one-dimensional index RMI to spatial data. It uses a ZC to order the data points and map spatial coordinates to one-dimensional values. The index function f then learns to predict the rank of a point p given its Z-value as the query key. For point query processing, ZM needs to transform the query point to a Z-value to predict the storage position of the data record corresponding to the query point by f. For range query processing, ZM computes the minimum and maximum Z-values of the query range and runs two point queries with the two Z-values to locate the data records corresponding to the two values. All data records in between are the query answer.

However, it is important to note that the index building cost for ZM can be significant for large datasets. This is because ZM follows a tree structure (see Figure 2.9), which requires more models to maintain query efficiency on large datasets and brings the computational overhead of learning models.

Experiments show that ZM can be up to 2.5 times faster than R-trees in query processing and while achieving a 97% reduction in index size on a spatial dataset with 100,000 objects. The drawback of using a ZC (or some other SFCs) is that it is difficult to determine the optimal grid resolution to obtain a smooth CDF to be approximated by the index function f. This limits the effectiveness of the resultant learned indices, especially for large and skewed datasets.

ML-index [23] is a two-layer index that follows the idea of iDistance [44]. Its upper layer contains reference points to help map spatial data points to one-dimensional values for data ordering. The lower layer is again a learned one-dimension index, e.g., RMI (cf. Figure 2.13). ML-index computes the distance between each data point and its closest reference point and uses this distance plus an offset as the one-dimensional value for index learning. The offset here is added such that the data points with different closest reference points can be differentiated. The offset value for the data points that are closest to the same reference point is the same, and it is the sum of all data points that have been considered already.

For point query processing with ML-index, the first step is to compute the scaled value (by adding the offset value and the mapped value) based on the reference point closest to the query point. Then, the computed value is used to query the learned one-dimensional index to fetch the query answer.

For range queries, ML-index iterates over the reference points. For each reference point, ML-index calculates the closest and the furthest points from the query range. These two points correspond to a one-dimensional query range, the lower and upper bounds of which are used to perform two point queries. All data points in between are fetched and filtered by the query range in two-dimensional space to compute the final query result.

For kNN queries, ML-index runs a series of range queries with increasing query range size until k points are found, while no other points can be closer to the query point than these k points. To process a kNN query at point q, the query ranges are generated based on the location of q and an initial search radius r. Each query range (i.e., a circle), is denoted by c_q. For each reference point p_i, ML-index also computes a circle c_i centered at q that tightly encloses all data points allocated to the partition of p_i. We denote dis_i as the distance from q to p_i. Then, there are two cases to consider based on the position relationship between c_q and c_i: (1) Circle c_i fully encloses c_q. In this case, the points that are nearer to p_i (with the distance larger or equal than $dis_i - r$) and further from p_i (with the distance is smaller or equal than $dis_i + r$) are considered. This is to ensure that for each reference point p_i, no point that is further than distance r from q is retrieved. (2) Circles c_i and c_q intersect. In this case, only the points nearer to p_i are considered. This is because points that are farther from p_i do not fall within circle c_i and, therefore, do not need to be retrieved. There is a third case where circles c_i and c_q do not intersect. All data points in c_i are ignored in this case – such points may still be considered later on when the radius r of c_q is enlarged.

For index building, ML employs the k-means algorithm to identify suitable reference points for iDistance [44]. However, this method becomes computationally expensive as the dataset size and the number of reference points (k) increase. There exists a trade-off between query efficiency and the computational cost of building the index. Specifically, a smaller k allows for faster index building but results in fewer reference points. Consequently, each reference point ends up associating with a larger number of data points, potentially leading to less efficient queries.

LISA [58] considers larger datasets and uses machine learning models to generate searchable data layouts under external data storage settings. It has three main components: (1) a mapping function that transforms spatial points into one-dimensional values, (2) a shard prediction function that divides the mapped space into shards using machine

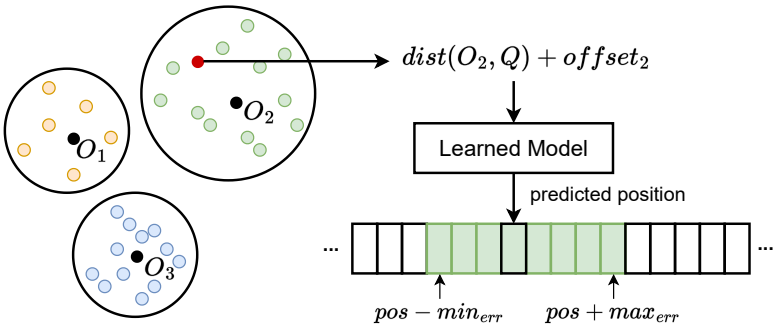


FIGURE 2.13: The ML-index structure

learning techniques, and (3) a set of "local models" that group shards into pages (cf. Figure 2.14). LISA partitions the data space using a grid and maps the spatial data points to one-dimensional values. It learns a shard prediction function to predict a shard ID for each point given its mapped value. The points are sorted and indexed by their shard IDs and grouped in the data pages by a set of local models.

LISA supports range queries and kNN queries (and hence point queries implicitly). For range queries, LISA decomposes a large query into a series of smaller query ranges, where each one at least interacts with one cell in the grid space. For each cell, LISA selects the shards that overlap with the query range. Then, LISA uses the local models to find the candidate pages, which are further filtered by the full query range to obtain all the query results. For kNN queries, LISA combines a lattice regression model and a range query for query processing. Here, the lattice regression model predicts a search range to form the range query. If the query range encloses fewer than k points, the area of the query range is expanded. The expansion factor is determined based on the size of the candidate result set R. If R is empty, each edge of the query range is doubled. Otherwise, the area is expanded by a factor of $\frac{k}{|R|}$.

The structure of LISA also supports insertions and deletions. When inserting a point, LISA finds the target shard with its grid structure and shard prediction function. Then, local models are used to find a target page for the insertion. If the page is full, a new page will be created. When deleting a point, LISA follows the insertion procedure to locate the data page holding the point. Then, the point will be removed from the page. If a page becomes empty after data deletion, or it only contains a few points, two consecutive pages within the same shard are merged.

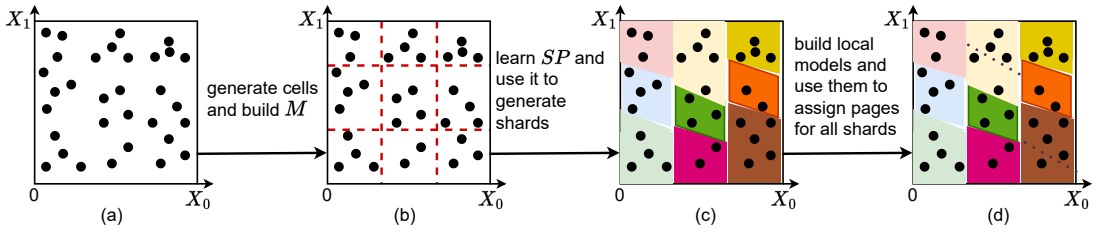


FIGURE 2.14: The LISA index structure

Recursive Spatial Model Index (RSMI) [84] takes a different approach to learn data partitions to cope with large datasets. It performs a recursive partitioning of a dataset $\mathcal{D}$ until each partition has no more than N points – a parameter that depends on the learning capacity of a single index model. Subsequently, for each partition, an index model is learned. This index will be detailed in Section 3.3.

Both LISA and RSMI use prediction to obtain the data partitions based on data distribution, making them more suitable for large datasets. In terms of the dimension reduction technique, RSMI uses SFC while LISA uses a grid to partition the data space and use a monotonic function to map spatial points into one-dimensional values. Additionally, it is important to emphasize that the index building cost of both LISA and RSMI is far from negligible, especially when indexing large datasets consisting of 100 million points (see Section 4.6). In such cases, the time required has been reported to span several hours. Both LISA and RSMI employ a multi-level tree structure, sharing similar index-building cost challenges with RMI [53]. Specifically, these costs are influenced by the number of models in the index structure and the computational cost of training each model.

Query-aware learned spatial indices. Flood [70] performs data partitioning to shrink the data space for index learning, where query workloads are used to estimate the query frequency of each dimension. When Flood partitions a d dimensional data space, it partitions in $(d-1)$ dimensions and leaves one dimension to build one-dimensional indices. For the $d-1$ partitioning dimensions, Flood computes the query frequency of each dimension as well as dimension combinations. For the last dimension, Flood creates an RMI index for each space partition.

Flood learns a cost model based on query workloads, which can estimate the time cost when given queries and a dataset. The cost model is designed to optimize the layout of the grid, i.e., the number of columns assigned to each of the $d-1$ dimensions in the grid. The cost model takes three inputs: a dataset $\mathcal{D}$, a query workload Q, and the grid

layout L. The output of the cost model is the estimated query time of all queries in Q on dataset $\mathcal{D}$ with the grid layout L.

In addition, Flood uses empirical CDF models to project the potentially skewed data space into a more uniform space. Flood models each dimension using an RMI. When given a value k on dimension i. A naive way to locate the partition of k is to record the locations of partitions and compare k with them. However, Flood uses an RMI to learn the partition and uses $\lfloor CDF(k) \cdot p_i \rfloor$ to locate, where p_i is the number of partitions on dimension i. This step flattens the data distribution by RMI, which enables a shorter search range and is the key factor in enhancing query efficiency. The ability to self-optimize allows Flood to outperform alternative techniques in query processing by up to three orders of magnitude.

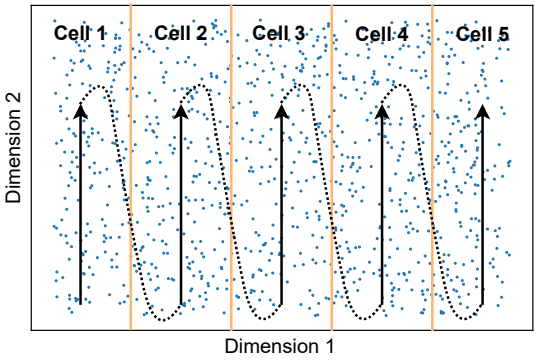


FIGURE 2.15: Partition of Flood in a two-dimesional space

For index building, Flood first ranks the d (2 in Figure 2.15) attributes according to a cost model. Next, it overlay a $(d-1)$-dimensional grid on the data space. In this grid, the ith dimension is split into c_i uniformly spaced columns. Every data point is assigned to a cell in this grid, and the process of locating the cell of a point is like that in the Grid-file [72].

All grid cells (and hence the data points inside) are sorted by their order in the first dimension, then the second dimension, and up to the $(d-1)$-th dimension. The data points within the same cell are further ordered by the coordinates in the last dimension. Figure 2.15 shows the sorted order for a dataset of two dimensions.

kNN queries and point queries are not discussed in the Flood paper, as the work does not focus on geospatial analytics. The paper only details range query processing. As shown in Figure 2.16, range queries over Flood are processed in three steps:

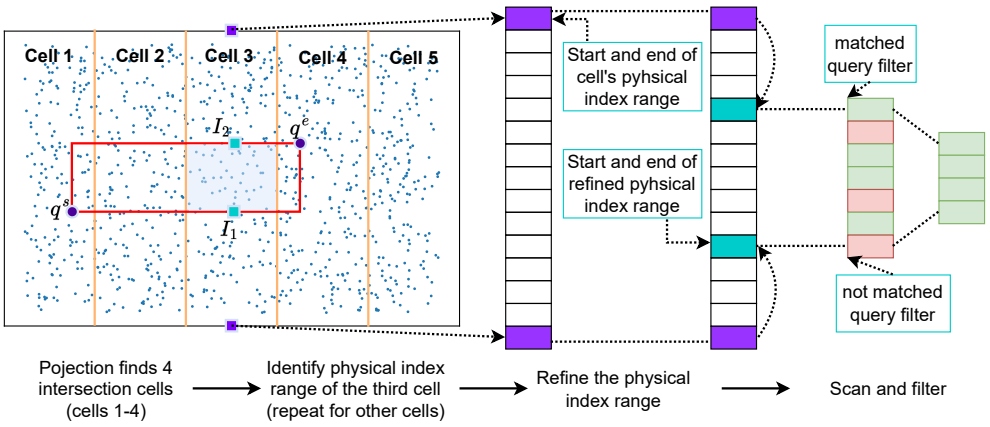


FIGURE 2.16: Query over Flood

1. Projection: Locate the cells in the grid layout that intersect with the query rectangle. For each cell that interacts with the query rectangle, identify the range of the enclosed data points in storage. For the cells that are fully overlapped with the query range in the first $d-1$ dimensions, e.g., Cells 2 and 3, it is straightforward to fetch the points inside the query range in the d-th dimension. For the cells that are partially overlapped with the query range in the first $d-1$ dimensions, e.g., Cells 1 and 4, refinement and scan steps are needed.

2. Refinement: For each partially overlapped cell, the refinement step computes the storage positions of the two data points that bound the query range in the d-th dimension, which can be easily done by binary search. To streamline the refinement step, Flood uses the learned model, i.e., RMI, to predict the positions of the bounding points.

3. Scan: The data points stored between the two bounding data points identified in the refinement step are scanned and compared with the query range to identify the final query result.

Tsunami [26] is an enhanced version of Flood that aims to address two issues. (1) Flood cannot adapt to skewed query workloads, where the query frequencies and selectivity vary across the data space. (2) When the data point coordinates in different dimensions are correlated, the equi-size partitioning strategy of Flood is ineffective.

Tsunami contains a *Grid Tree* and *Augmented Grids*. Grid Tree is a space-partitioning decision tree that partitions the data space into non-overlapping regions.

To optimize the Grid Tree, queries are clustered into different types based on selectivity. When all queries are grouped together, the impact of skew can be obscured. This is because the skewness of different query types may counterbalance each other, thus neutralizing their individual effects. A data structure named the *skew tree* is also proposed to find the partition positions along a dimension. The skew tree is a balanced binary tree encompassing the entire data range of a dimension, with each node representing a specific range along that dimension. As such, the partition positions can correspond to the boundaries between any two contiguous leaf nodes in the skew tree.

An Augmented Grid is constructed for each partition by using functional mappings and conditional CDFs to address the issue of correlated data. Given a pair of dimensions x and y that are correlated. If they are partitioned independently, their correlation is not captured. Thus, the partitions can not fit the query workloads well, which leads to high query costs in filtering out irrelevant points.

Tsunami follows the same partition strategy (i.e., using RMI) to partition dimension x uniformly. Then, the query filters (from query workloads) over y are transformed into filter dimension x using a mapping function. Finally, Tsunami partitions dimension y dependent on dimension x. Therefore, dimensions x and y are partitioned dependently under the supervision of query workloads.

Qd-tree [108] uses reinforcement learning (RL) to guide the space partitioning process of a kd tree, which helps minimize the number of data block accesses given a query workload (assuming that data points are grouped and stored in blocks). RL is used instead of Dynamic Programming (DP) because DP may still be too expensive given the large optimization space in multi-dimensional and large query set settings.

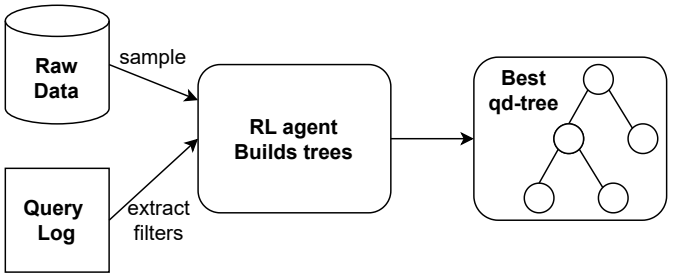


FIGURE 2.17: System architecture of Qd-tree

As shown in Figure 2.17, the Qd-tree constructor is designed to learn tree index over a query log and raw data based on candidate cuts of queries (for reward calculation) and sampled datasets (for data indexing), respectively.

In Qd-tree, each internal node has two children, and each leaf node corresponds to data blocks. Each internal node corresponds to a data space partitioning at some dimension. A typical strategy is to partition with the median data point coordinate value in a data dimension considered for the partitioning – think of the partitioning strategy of a kd-tree. Considering that the query workload may be skewed towards a certain side of the dimension, e.g., mostly querying points between the 25^{th} and the 75^{th} percentiles, a better partition position should be either at the 25^{th} percentile or the 75^{th} percentile. Qd-tree learns such a partitioning strategy based on the query workload. It uses a model to predict the reward (i.e., the query cost) of different cuts to construct the Qd-tree, which helps avoid the cost of actually running the queries over the intermediate trees.

RLR-Tree [35] uses machine learning models to replace heuristic rules and optimize the R-tree structure. It models the decision marking in sub-tree choosing and node splitting for inserting a data point into an R-tree as Markov decision processes. Such processes are then optimized using RL, where the reward is defined as the difference in query time between the current and the previous query. The overall architecture of the RLR-Tree is shown in Figure 2.18.

Although the Qd-tree and the RLR-tree both use RL to improve the performance of traditional spatial indices, there are significant differences between them in the RL model. For the Qd-tree, the query workload is assumed to be known, while RLR-Tree has no prior knowledge of the query workload. Qd-tree cannot compute the rewards until an index is built, while RLR-tree can compute the rewards after each node splitting.

2.3.3 Update Handling

We have discussed the key ideas of index building for both traditional and learned indices. For static datasets, index-building is a one-off task as the data will not be changed. For dynamic datasets where there are changes to the dataset, *update handling* is an important issue to retain high query performance for the index structures.

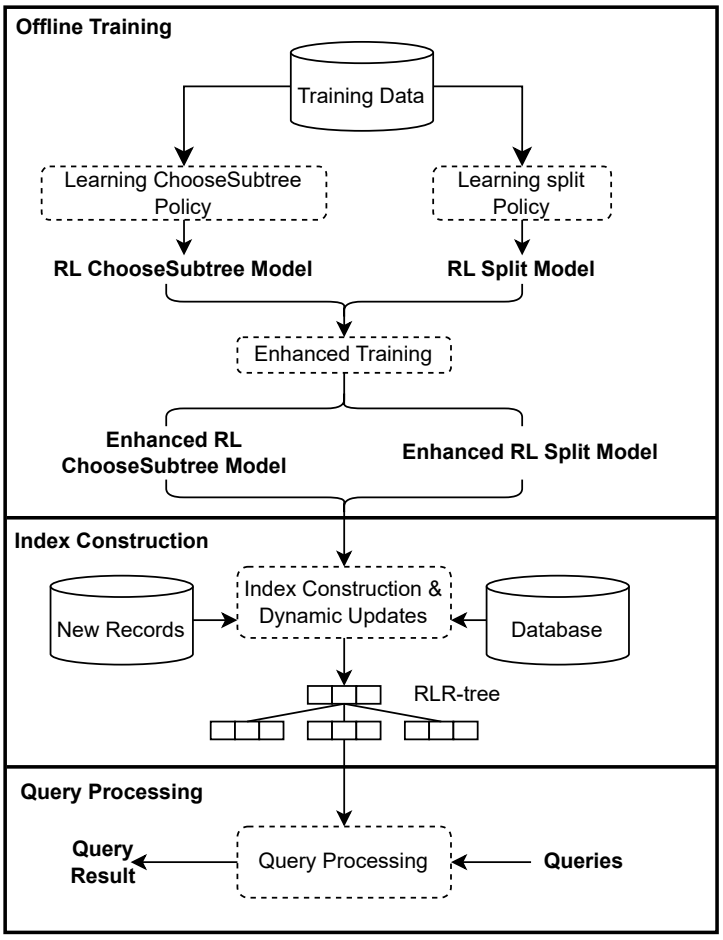


FIGURE 2.18: RLR-Tree overview

For the traditional indices, e.g., B-trees and R-trees, update handling is simple as these structures come with building algorithms based on dynamic data insertion already. For the learned indices, their main components are machine learning models, which can be difficult to update when the dataset changes. This is because the learned index function f comes with prediction errors. Such errors may change with data updates, and training a index model typically means at least a scan over the full dataset, which can be too expensive to be done on a per data update basis.

In Section 2.3, we have discussed the update handling of a few representative learned indices. For example, FITing-tree and PGM-index use extra space to store new insertions. When the number of new insertions reaches some threshold, FITing-tree will be rebuilt over the updated full dataset. PGM-index builds a series of small PGM-index structures following the LSM-tree technique. The i-th PGM-index is built over a dataset $\mathcal{D}_i$ and should be either empty or contains $2^i - 1$ keys. Upon the insertion of a new key, denoted

as *key*, the first empty PGM-index, for example, $\mathcal{D}_j$, is selected as the target for insertion. All keys from preceding PGM-index structures are merged to $\mathcal{D}_j$. Following this, a fresh PGM-index is built over $\mathcal{D}_j$. For LISA and RSMI, update handling is done directly on the index structure. LISA finds the target insertion position in a data page, which follows a point query procedure. Splitting and merging of data pages are supported, such that retraining of the index models is not required. RSMI also uses point queries to find the locations for data insertions. However, a rebuild of a few sub models may still be needed when the size of a partition reaches the learning capacity of a sub model such that further new data partitions need to be formed.

A few other update handling strategies have been proposed in the literature. To handle insertions, a trivial way is to increase the maximum error by one after each update. However, this approach can soon lead to a high query cost if there are frequent insertions. A tighter estimation of the updated maximum error [38] is achieved by tracking the error drifts of a number of *reference points*. At query time, the closest reference points on both sides of the query point are fetched, and their error drifts are used to estimate the updated maximum error of the query point with linear interpolation.

RUSLI [65] is a learned index that can be updated in constant time. It is based on the RS-index [49], which is a learned index that can be built with a single pass over the dataset. RUSLI extends RS by adding a temporary *overflow array* to store the inserted data. For each spline point, RUSLI also maintains a corresponding *insertion position*, which is continually updated as new data points are inserted. During the insertion of a new data point, a point query is initially executed to find the corresponding spline point. The new data point is then inserted into the overflow array based on the recorded insertion position. Thus, there are no additional costs associated with searching for an insertion position or moving data items within the overflow array. Once the model has been updated, the temporary memory is discarded.

TALI [36] is an update-distribution-aware learned index for social media data. There are two core methods to handle updates, namely Learn Update Distribution (LUD) and Learn Update Distribution with Bound (LUDB). The LUD method is proposed to address an issue in ALEX, where elements need to be shifted to create space for insertion when conflicts arise. LUD avoids such conflicts by learning the distribution of the updates and calculating the sum of the initial keys and the keys being inserted. This

strategy leads to more efficient space allocation and reduces the amortized cost of data shifting for insertions. However, LUD suffers when there is a large number of insertions, which gradually occupy most of the insertion gaps and bring element shifts. To overcome this issue, LUDB is proposed to set a boundary to limit the capacity of node, rather than directly allocating space based on the sum of initial and inserting keys. If this sum exceeds the boundary, the capacity of node is determined by initial keys divided by initial density. Otherwise, space is allocated based on the sum. LUDB also employs a merge algorithm that sets a minimum data amount, helping to reduce index size and space occupation. This improves index performance and achieves a smaller index size with a negligible time cost.

2.3.4 Other Machine Learning-Based Index Structures

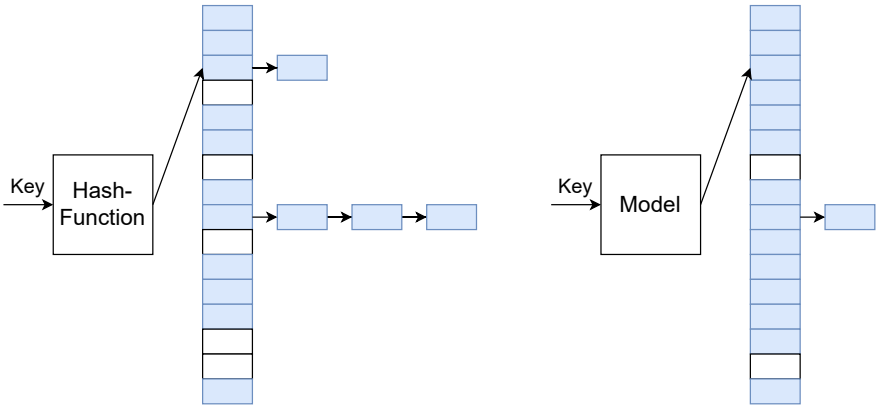


FIGURE 2.19: Traditional Hash-map vs. learned Hash-map

We briefly discuss a few other machine learning-based structures to support efficient data retrieval.

Learned hash function. Traditional hash functions are typically based on random number generators. They are designed to distribute data keys evenly across the hash buckets. However, collisions are inevitable, where multiple data keys are mapped to the same hash bucket, resulting in lower query performance.

Learned models have been proposed to serve the purpose of hash functions [53]. Unlike learning an index to replace B-trees, the *learned hash function* does not require storing keys compactly or in a fixed order. Instead, it aims to scale the CDF of a dataset by a target size M and uses $h(k) = F(k) \cdot M$ as the learned hash function. In a situation where F flawlessly captures the empirical CDF of keys, conflicts can be eliminated and

the query performance can be improved (cf. Figure 2.19). This hash function is not tied to the Hash-map architecture. A recursive model architecture (e.g., RMI) can be utilized for this model.

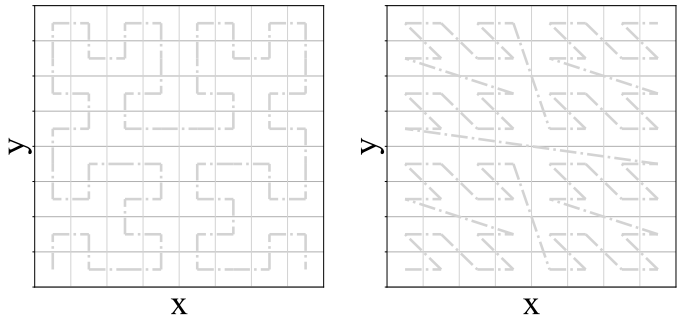
Learned Bloom filters. Bloom Filter is a probabilistic data structure that efficiently checks if a key exists within a dataset. It utilizes a bit array and various hash functions to map a search key onto bits in the array.

To verify if a query key exists in the dataset, the key is run through the hash functions, and the corresponding bits in the bit array are examined. If all bits are set to one, the Bloom filter suggests the key might be in the dataset. However, if any bit is zero, the Bloom filter confirms the key is not in the dataset. The primary drawback of the Bloom filter is the possibility of false positives.

To mitigate false positives, *Learned Bloom Filters* [53] are introduced. The fundamental concept is the employment of a classifier f that outputs the probability of a given query key k. If $f(k)$ is greater than or equal to a threshold value τ, the key is deemed to exist in the dataset. If not, a traditional Bloom Filter is used as a fallback. This two-step query process is anticipated to decrease the false positive rate.

A learned Bloom Filter combines the space-saving benefits of a Bloom Filter with the predictive power of a learned model, thereby minimizing the occurrence of false positives. This unique fusion makes it a highly suitable choice in scenarios where the need for space efficiency is of utmost importance.

Learned database system. SageDB [52] is a learned database system where learned components such as indices and sorting algorithms are designed to replace their traditional counterparts. It includes a learned spatial index. To learn this index, the data points are sorted and partitioned along a sequence of dimensions into equal-sized cells. Then, the points are ordered by the cells in which they lie, and an index function f is learned to predict the rank of a point given its coordinates. The dimensions used for sorting the points and the partition granularity are learned. However, no details are available on how this learning is done and how the cells are ordered.



(A) Hilbert curve (B) Z curve

FIGURE 2.20: Examples of space-filling curves

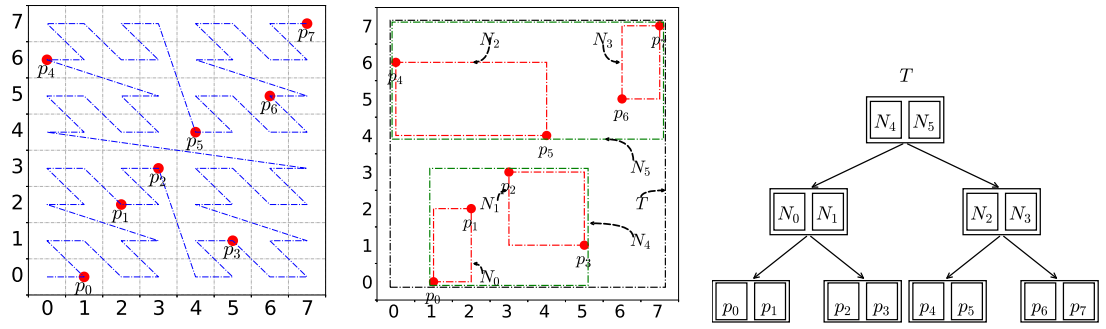
2.4 Space-filling Curves

Spatial data can be ordered in various ways, e.g., by coordinates in a single dimension or in a combination of multiple dimensions. SFCs are a commonly used approach. They form a regular grid over the data space and go through each grid cell exactly once, and the order that an SFC goes through the cells gives an order over the points in the data space. HC and ZC (cf. Figure 2.20) are two types of SFCs widely used in relational databases [81] and data warehouses [4] for managing spatial (multidimensional) data.

The order that an SFC visits a grid cell gives a curve value to the cell (and the data points in the cell), e.g., the Z-value for a ZC. Using the curve values of the data points in a spatial dataset $\mathcal{D}$, one can build a dimensional index (e.g., a B-tree) over $\mathcal{D}$.

In Figure 2.21(a), eight data points (from p_0 to p_7) in an 8×8 grid space are ordered by a ZC. For example, point p_3 (5, 1) = (101, 001) is mapped to 010011 = 19, which is the Z-value of the point and is calculated by interleaving the bits of the column number of the point in both dimensions. Then, all the data points are packed into pages by size B ($B = 2$). Next, in Figure 2.21(b), each leaf node N_0 to N_3 contains two data points, which are bulk-loaded into N_4 and N_5. Finally, in Figure 2.21(c), a B-tree tree index is built for these points, which can also be seen as an R-tree if the MBRs of the tree nodes are computed and stored.

SFCs are used in not only traditional indices [47, 54, 77, 106] but also the learned ones. For example, ZM and RSMI are both built based on ZC. SFCs have also been studied in other fields such as data mining [16] and machine learning [42, 100]. We discuss optimization and cost estimation techniques for SFCs next.



(A) Order data points by Z-values.

(B) Pack data points.

(C) Build tree structure.

FIGURE 2.21: An example of ZR-tree

2.4.1 SFC Optimization

Traditional SFC-based indices are built in a bulk-loading manner where the loaded data points follow the ZC order or HC order [47, 77]. Sort-Based R-tree (SBR) [1] first considers the query workloads to help improve the query performance of SFC-based indices. The key idea of SBR is that for any determined sorted order of the data objects, it can heuristically generate a partitioning for all the data objects (i.e., to cut a sorted array into segments of different sizes). The partitioning is guided by a cost function, which is calculated by measuring the sum of the area of the MBR of partitions, and the size of each partition must be in a given size range.

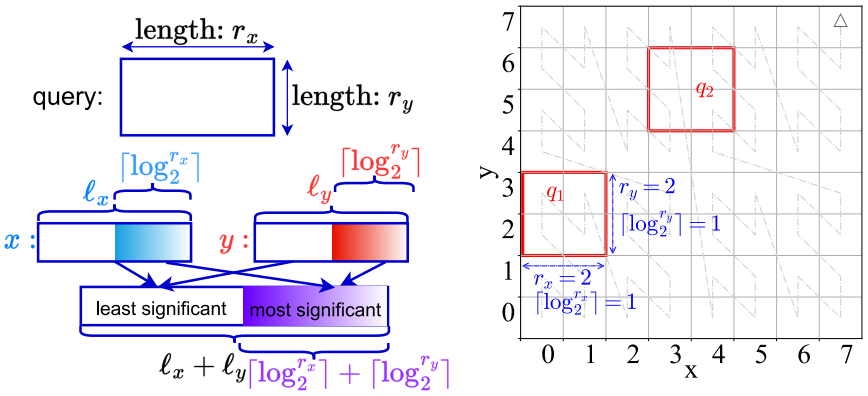
QUILTS [73] is another heuristic method to search for an SFC optimized for a given query workload. It focuses on the Bit-merging Curve (BMC), which is a special type of SFCs constructed as below. Given a regular grid partitioning of a d-dimensional data space, where each dimension has 2^{ℓ} columns, a BMC goes through each grid cell once. The order that the BMC visits a cell (i.e., the curve value of the cell) is calculated by merging the $d \cdot \ell$ bits of the column indices of the cell for all d dimensions, e.g., a ZC is a BMC where the bits of the d column indices are interleaved to form the curve value.

Recently, the Instance-Optimal Z-Index [80] is proposed, which is another SFC-based index for 2-dimensional data. It uses a quadtree-like strategy to recursively partition the data space. It creates four sub-spaces of a (sub-)space, which may be of different sizes. The four sub-spaces are each ordered by a ZC with a different grid resolution, that follows a '$\mathbf{\Sigma}$' or an 'N' shape. At the bottom level of the space partitioning hierarchy, the ZCs of sub-spaces that come from different parent sub-spaces are connected following

the order of the ZC that traverses the parent sub-spaces. This way, a curve is formed that traverses all bottom-level sub-spaces, and the data points are indexed in that order.

2.4.2 Cost Estimation for SFCs

Cost estimation approximates the cost of processing a query without actually computing the query. A couple of studies [68, 107] offer theoretical estimations on the number of SFC curve segments covered by a query range, which is an indicator of the query costs of SFC-based indices. These studies do not offer empirical results or guidance on how to construct a query-efficient SFC index.



(A) Bit merging in QUILTS. (B) Curve design over q_1.

FIGURE 2.22: The curve design method of QUILTS.

QUILTS constructs a BMC that aims to optimize the costs to process a query workload as follows. Given a query range, QUILTS calculates the number of columns r_i needed to cover the query span in dimension i (the minimum is taken when there are multiple query ranges in a given query workload). The lowest $\lceil \log_2^{r_i} \rceil$ bits (called *the most significant bits*) of the coordinates of dimension i are then merged into the final bit sequence of the targeted query-optimized BMC. This is done for every data dimension. Intuitively, this allows data points covered by a query span to be more likely to reside in the same data blocks, for query cost optimization. The rest of the bits (called *the least significant bits*) of the coordinates in each dimension can then be merged in any order into the final bit sequence of the targeted query-optimized BMC. Figure 2.22(a) illustrates the QUILTS curve design method with two dimensions x and y.

An issue with QUILTS is that it only considers the size but ignores the location of the query range, which can lead to sub-optimal curves. Figure 2.22(b) gives an example,

where two queries q_1 and q_2 of the same size ($r_x = r_y = 2$ and hence $\lceil \log_2^{r_x} \rceil = \lceil \log_2^{r_y} \rceil = 1$) lead to the same curve (e.g., a ZC, as its last two bits of the curve value consist of one bit from each dimension). Query q_1 covers a single consecutive curve segment while q_1 covers four, which will lead to rather different query costs when the data points are stored following the curve order.

QUILTS formulates the query cost $\mathcal{C}_t$ for a BMC-based index over a set of queries as $\mathcal{C}_t = \mathcal{C}_g \cdot \mathcal{C}_l$, where $\mathcal{C}_g$ is a *global cost* and $\mathcal{C}_l$ is a *local cost*. The global cost is the length of a continuous BMC segment to cover a query range q fully minus the length of the BMC segments in q, for each query. The idea is to count the number of segments outside q that may need to be visited to compute the queries. The local cost is the entropy of the relative length of each segment of the BMC curve outside q counted in the global cost, which reflects how uniformly distributed the lengths of such segments are.

However, these two costs rely on the accumulated length of the curve segments outside q, which is expensive to compute. Given n range queries, it takes $O(n \cdot c_t)$ time to compute $\mathcal{C}_t$, where $O(c_t)$ is the average estimation cost per query. Further, they can only be used to estimate the query costs of a given BMC index and cannot guide an efficient search for a query-efficient BMC index.

2.5 Spatial Queries, Datasets, and Evaluation Metrics

This section presents basic types of spatial queries (Section 2.5.1) and commonly used datasets (Section 2.5.2) and evaluation metrics (Section 2.5.3), to provide a basis for the empirical studies of the thesis.

2.5.1 Spatial Queries

Point query, range query, and kNN query are arguably the most common types of spatial queries, which can be used to form more complex query types such as spatial keyword queries [18] or moving kNN queries [74].

Point queries return the record(s) with index keys that exactly match a query key. In a spatial data setting, a point query finds spatial objects at given query point locations.

Definition 2.1 (Point Query)**.** Given a query point q and a spatial dataset $\mathcal{D}$, a point query PQ returns every object $p \in \mathcal{D}$ at q:

$$PQ(q) = \{p \in \mathcal{D} | q \cap p = q\}. \tag{2.1}$$

Without the support of indices, a point query requires a full dataset scan to find all the data objects that satisfy the query. Indices like R-trees help reduce the query time by enabling fast pruning of the search space, i.e., only nodes with MBRs overlapping the query point q need to be visited.

Range queries return the records with index keys in a given query range. In a spatial dataset, a range query finds spatial objects that satisfy some query predicate with a query range (a.k.a, a query window), which is a (hyper)rectangle. Different query predicates have been studied in the literature. We consider a common one in this thesis, i.e., "enclosure", which can be used for, e.g., finding all restaurants in some region of interest.

Definition 2.2 (Range (Window) Query)**.** Given a d-dimensional interval $I^d = [l_1, u_1] \times [l_1, u_1] \times \cdots \times [l_d, u_d]$, a predicate g, and a (d-dimensional) dataset $\mathcal{D}$, a range query WQ returns every object $p \in \mathcal{D}$ that satisfies g with I^d:

$$WQ(I^d) = \{p \in \mathcal{D} | g(I^d, p) = \text{true}\}. \tag{2.2}$$

Like point queries, range queries require a full dataset scan to find all the data objects that satisfy the query predicate with the query ranges. Indices such as the R-trees help reduce the query time by enabling fast pruning of the search space, i.e., only nodes with MBRs that satisfy g with the query range needs to be visited.

K Nearest-Neighbour queries (kNN) are generalised versions of the Nearest-Neighbor (NN) query (i.e., $k = 1$). They return the k data records that are the "nearest" to a given query point, where a distance function *dist* is used to help define the nearness relationship. Such queries are often used to find the nearest points of interest, e.g., charging stations for electric vehicles.

Definition 2.3 (K Nearest-Neighbor Query)**.** Given a query point q, a distance function *dist*, and a dataset $\mathcal{D}$, a kNN query $KNNQ$ returns a subset S of $\mathcal{D}$ of k objects, such

that no object in $\mathcal{D} \setminus S$ is closer to q than any of the objects in S.

$$KNNQ(k, q) = S : S \subset \mathcal{D} \cap |S| = k \cap \forall p' \in \mathcal{D} \setminus S, p \in S : dist(p, q) \leq dist(p', q). \quad (2.3)$$

KNN queries can be processed by a dataset scan with the help of a size-k heap. When tree-based index structures are available, kNN queries can be processed by the well-known depth-first or breath-first algorithms [55]. A priority queue is used in these algorithms to help order the visit of the tree nodes and data objects, and to support early termination once no more promising nodes or data objects are available. When no particular ordering can be established based on object distances, a commonly used strategy is to fall back to range queries, i.e., to query the dataset with a sequence of range queries centered at the query point q and with increasing sizes, until k data objects are found and no closer objects can be found.

2.5.2 Datasets

Large one-dimensional and spatial datasets with over 100 million data records have been used in recent studies on learned indices. We summarize the commonly used ones in this subsection.

One-dimensional datasets. Four real datasets are commonly used to test the learned one-dimensional indices [30, 49, 50], as listed below. They have at least 200 million data records, and osm has up to 800 million data records. We plot the CDFs of these datasets in Figure 2.23.

1. **amzn**: This is a collection of book popularity data from Amazon, where each key represents the popularity of a particular book.
2. **face**: This is a collection of distinct Facebook user IDs [101].
3. **wiki**: This is a collection of article edit timestamps from Wikipedia.
4. **osm**: This is a collection of cell IDs from Open Street Map [76], where each key is an embedded location.

Synthetic one-dimensional datasets have also been used, commonly with *uniform* and *skew* distributions. Such datasets have up to 128 million records. A skewed dataset

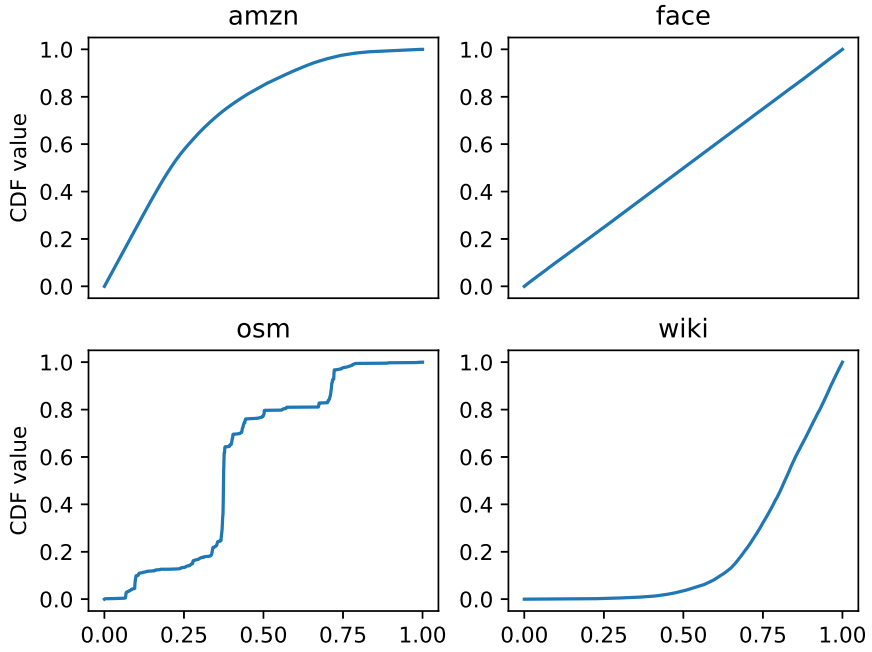


FIGURE 2.23: CDFs of four real one-dimensional datasets

is formed by a uniform dataset by replacing the key x in the uniform dataset with x^s where s can vary from 1 to 9 [85]. Here s is the skewness parameter, and $s = 1$ means a uniform dataset.

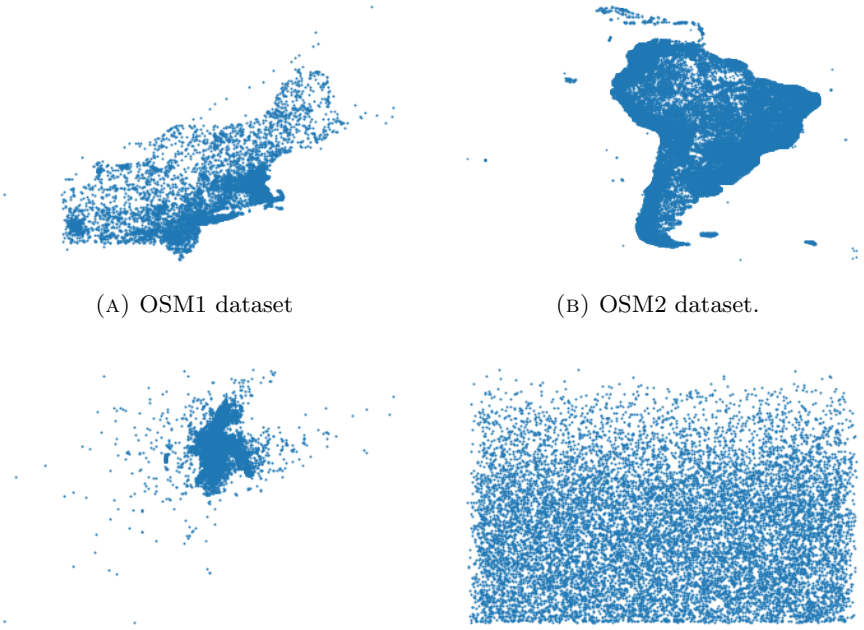
Spatial (multidimensional) datasets. There are also four commonly used multidimensional datasets: **OSM1**, **OSM2**, **TPC-H**, and **NYC**, as summarized below. The dataset sizes range from 100 to 180 million. We plot these datasets in Figure 2.24.

1. **OSM1**: This dataset contains some 100 million points (2.2 GB) in North America.

2. **OSM2**: This dataset contains over 180 million points (4.4 GB) in South America. Both OSM1 and OSM2 are extracted from OpenStreetMap [76], which is a free, open geographic database with a topological data structure to store the data points.

3. **NYC**: This dataset contains some 150 million taxi transaction records, including pickup and drop-off locations of each trip in New York City in 2015. The pickup locations (i.e., spatial points) alone are 5.6 GB in size.

4. **TPC-H** [99]: This dataset contains 120 million records from the data table `lineitem` of the TPC-H benchmark. This is a multidimensional dataset.



(A) OSM1 dataset

(B) OSM2 dataset.

(C) NYC dataset.

(D) TPC-H dataset.

FIGURE 2.24: Visualizations of four multidimensional datasets

Another smaller real dataset named **Tiger** has been used, which contains some 17 million rectangles (950 MB in size) representing geographical features in 18 Eastern states of the USA [97]. This dataset comes for the TIGER/Line Shapefiles [97] which have been commonly used in studies on traditional spatial indices.

Synthetic spatial datasets typically used uniform, normal, and skewed distributions, with up to 128 million points (2.5 GB in size).

2.5.3 Evaluation Metrics

There are a few key measurements to consider when evaluating the performance of database indices:

1. *Index size* measures the space cost of an index structure either in memory or on an external storage device. Smaller index size is usually preferred, as it helps reduce the memory footprint (and potentially disk I/O) in both index building and query processing. However, query-efficient structures often also have large index sizes (i.e., to trade space costs for time costs). How to obtain query-efficient and small index structures is a challenge.

2. *Index build time* measures the time cost required to build an index structure, and it plays a key role in the practical usability of an index structure. An index structure with a high query performance but requires a long time to build may not be very practical for indexing large datasets.

3. *Query time* measures the time cost required to process certain types of queries with an index structure. Typically, query time is composed of index lookup time, data retrieval time, and data transmission time.

4. *Disk I/O (block access)* measures the I/O cost required to process certain types of queries with an index structure, assuming external storage of the index structure and/or the actual data records. This metric is particularly important when external storage is used, as the time for I/O accesses is much higher than that for in-memory computation.

5. *Query accuracy* measures how accurate the results returned by an index structure are. Result accuracy is often traded for query efficiency over large datasets and complex queries, e.g., high-dimensional kNN queries. In the learned index context, query accuracy is also important as machine learning models often return inaccurate results.

6. *Update time* measures the time cost required to process updates (e.g., insertion or deletion) to the dataset upon which an index structure has been built. This is another essential metric to measure the practical usability of an index structure as efficiently updatable index structures are needed to cope with the dynamic nature of datasets in many real application scenarios.

2.6 Summary

In this chapter, we reviewed both traditional and learned indices, over both one-dimensional and spatial (multidimensional) data, with a focus on learned spatial indices which are the most relevant to this thesis. We discussed learned one-dimensional indices by two different index building strategies, i.e., prediction-based and interpolation-based. For learned spatial indices, we categorized them based on whether query workloads are used in the index building procedure.

We observe that the aim of existing learned spatial indices has focused on the query efficiency, while the index building efficiency has been largely understudied, which impacts the practicality of such structures. We will present a detailed experimental study of learned spatial indices on both aspects in Chapter 3.

We further reviewed SFC-based spatial indices and discussed how such indices are optimised in the literature, where the focus has been on query cost estimation to guide the optimization. Our proposed optimization technique will be presented in Chapter 6.

We also summarized the spatial queries, datasets, and evaluation metrics commonly used in recent learned spatial index studies, to set to technical background for the thesis.

Chapter 3

Efficiency of Spatial Index Learning: An Empirical Study

In this chapter, we contribute the first detailed empirical evaluation of learned spatial indices, seeking to measure their viability in connection with traditional spatial indices, e.g., R-trees. We study two learned spatial indices, ZM [103] and RSMI [84], which are among the first machine learning-based index structures for spatial data, to provide a comprehensive basis for both our later chapters and for future studies on learned spatial indices. We will show that these learned spatial indices outperform the traditional ones such as R-trees, kd-trees, and quadtrees in spatial query processing, confirming the strength of the learned spatial indices. Meanwhile, we will show that such indices take a long time to build, because they require repeated scans over the datasets to be indexed, motivating our later chapters to study efficient algorithms to build learned spatial indices.

3.1 Overview

Recently, two learned spatial indices, *ZM* index [103] and *RSMI* [84], have utilized Space-Filling Curves (SFC), such as the ZC [78] and HC [29], to order spatial data linearly, thereby enabling us to develop learned spatial indices. Initial results indicate that both

The work reported in this chapter has been published in the following paper: Jianzhong Qi, Guanli Liu, Christian S. Jensen, Lars Kulik. Effectively Learning Spatial Indices, *Proceedings of the VLDB Endowment* (**PVLDB**) 2020.

ZM and RSMI can significantly boost query performance, surpassing tree-based indices like R-trees.

However, there is a variety of traditional indices that perform better than R-trees. Currently, there is a lack of comprehensive comparison studies that demonstrate the degree to which ZM and RSMI outperform these traditional indices in metrics such as index building costs, query times, and index sizes. This creates an important gap in understanding the strength and limitations of learned spatial indices. Without a thorough comparative study, it is unclear whether the learned indices truly have an non-trivial performance gain comparing with the traditional indices when dealing with large spatial datasets in real-world settings. This chapter aims to fill this gap. As there is no open-source code available, all learned spatial index structures were re-implemented from scratch for this chapter. We aim to lay a comprehensive groundwork for future studies comparing learned and traditional spatial indices, while also delivering an in-depth evaluation of the strengths and weaknesses of learned spatial indices.

3.2 Preliminaries

We start with a few core concepts that will be used throughout the thesis.

Spatial data. Spatial data, also referred to as geospatial data, are data that come with geo-location information. They represent the locations of physical objects in space. Spatial data can take various forms, such as coordinates, addresses, or zip codes. Such data are crucial for a range of applications, from navigation to urban planning. In this thesis, we consider a set $\mathcal{D}$ of n spatial points. We focus on point data for their simplicity and generality. Non-point data, e.g., spatial objects with non-zero extends, can be converted to point data, while the query sizes can be expanded to ensure correct query results [96].

Space-filling curves (SFCs). A space-filling curve applies a regular grid partitioning over a data space and forms a curve that traverses every cell of the grid exactly once. The order of the traversal corresponds to a surjective function that maps a higher dimension value into a one-dimensional value. Examples of Space-Filling Curves include the ZC and HC. These curves are critical in various applications, including image processing, data

compression, and spatial databases, due to their ability to reduce the data dimensionality while maintaining spatial locality.

Point query. Given a d-dimensional dataset $\mathcal{D}$ ($d > 1$), a point query returns the data objects located at a given query point $q = (x_1, x_2, \ldots, x_d)$, where x_i denotes the query point coordinate in dimension i.

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

KNN query. Given a d-dimensional dataset $\mathcal{D}$, a kNN query returns the k points in $\mathcal{D}$ that are the nearest to a given query point $q = (x_1, x_2 \ldots x_d)$, where x_i denotes the query point coordinate in dimension i.

Formal definitions of these queries are included in Section 2.5.1.

Cumulative distribution function (CDF). The CDF of a dataset reflects the distribution of the dataset. Given a value x as the input, the CDF returns the percentage of data records with key values less than or equal to x. Let the CDF of a dataset $\mathcal{D}$ of size n be $cdf_{\mathcal{D}}$. Then, the storage address of a data record with key value key is just $cdf_{\mathcal{D}}(key) \cdot n$, if the dataset is stored in ascending order of the data values. The target of index learning is then to learn a function f to fit the CDF of a dataset, and $f(key) \cdot n$ approximates the storage address relevant to query key key for query processing.

Index function f: In learned indices, an index function is a function that is used to predict the storage location of a data object given its corresponding query key. It is often represented by some machine learning model, e.g., a linear model or a feed-forward neural network (FFN). The accuracy of the index function is measured by prediction errors, i.e., the difference between the predicted and actual locations of a data object.

Learned spatial indices. Given a d-dimensional dataset $\mathcal{D}$, an index over $\mathcal{D}$ is a data structure built to support efficient queries of $\mathcal{D}$. Learned spatial indices are a type of index structures that leverage machine learning techniques to predict the storage location of spatial data, with a strong potential to outperform the traditional spatial indices that perform traversal and search processes for query processing.

Problem statement. In this chapter, our goal is the perform a comprehensive study on the empirical performance of learned spatial indices. We focus on the following aspects:

1. Query efficiency: How do learned spatial indices perform comparing to traditional spatial index structures in terms of query efficiency?
2. Scalability: How do learned spatial indices scale with the increase in dataset size?
3. Index building efficiency: How do learned spatial indices perform comparing to traditional spatial indices in terms of index building time and space costs?

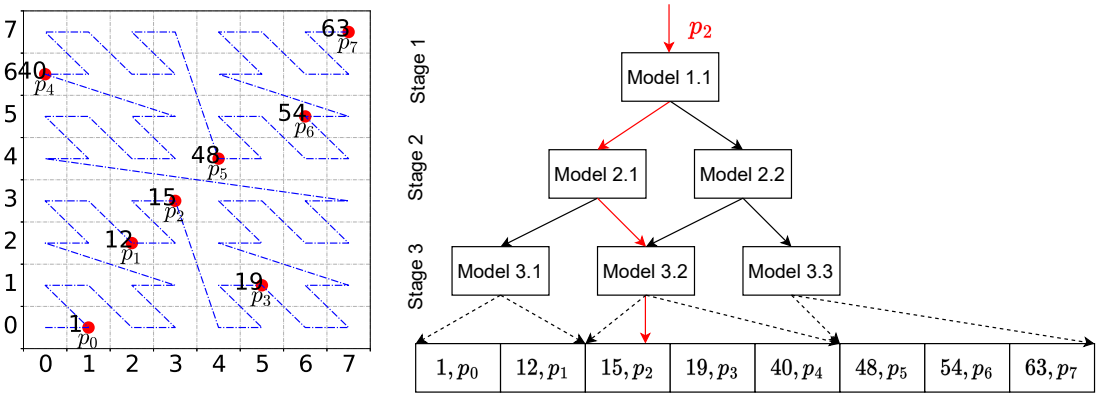
3.3 The ZM and RSMI Indices

We briefly review the structures, query processing, and update handling of ZM and RSMI in this section.

3.3.1 ZM: A Learned Z-order Model Index

To learn a spatial index, the *Z-order model* (ZM) [103] leverages the ZC to order the spatial points. This is because SFCs like ZC can preserve the locality and map points nearby to (mostly) adjacent values in the one-dimensional space. To apply ZC, the underlying data space is partitioned by a grid such that each data points is located in some cell. Then, ZC visits all the cells in Z-order such that all the data points are ordered based on the visited sequence. Once the spatial data points are ordered, the corresponding CDF is generated to be learned.

To learn the CDF, ZM applies a Multi-staged Model Index (MMI), which follows RMI [53] to partition the (one-dimensional) data space recursively to reduce the prediction error. Each model in ZM is an FFN, which is trained with the L_1 loss function. As Figure 3.1 shows, the upper-stage models of the MMI predict a model to be used for the prediction in the next stage. Intuitively, each model is supposed to focus the prediction of a partition of the dataset. When reaching the final stage, the prediction result is the expected storage location of the data object corresponding to the query object. A binary search over an error range enclosing the predicted location is run to find the exact query result.



(A) Ordering with Z-values. (B) Point query over ZM-index.

FIGURE 3.1: An example of ZM-index

In Figure 3.1(a), there are eight two-dimensional points p_1 to p_8. The data space is partitioned into 64 cells, where each point is located in a cell. We use ZC to go through all the cells such that the eight points are ordered, and each point is assigned a curve value. In Figure 3.1(b), a three-stage learned index structure, i.e., MMI, is built over the eight ordered points. When querying p_2 with the index, ZM first maps it to the Z-value 15. Then, the query interacts with Model 1.1, Model 2.1, and Model 3.2 to reach the storage location of p_2. Here, Model 1.1 and Model 2.1 predict a model in the next stage, whereas Model 3.2 predicts the location of p_2.

3.3.2 RSMI: A Recursive Spatial Model Index

While a single index function might be effective for small datasets, its ability to maintain high prediction accuracy diminishes as the dataset size increases. To address the issue of limited learning capacity of individual index functions, the Recursive Spatial Model Index (RSMI) was proposed.

RSMI recursively divides an input dataset $\mathcal{D}$ until each partition has no more than N points – a parameter that depends on the learning capacity of a single index model. RSMI considers external data storage for large dataset settings, where every B points are stored in a data block, and an index model predicts the block ID for each data point. This means that RSMI assumes an index model that can make effective predictions for $\lceil N/B \rceil$ block IDs.

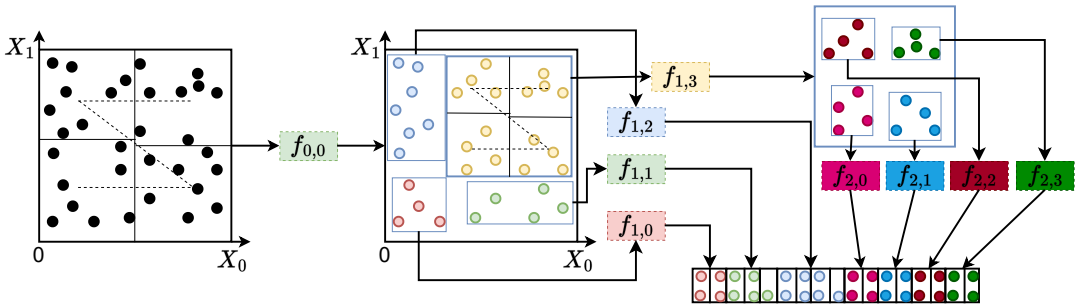


FIGURE 3.2: RSMI index structure ($N = 8$ and $B = 2$)

To satisfy the learning capacity of an index model, RSMI first partitions the dataset $\mathcal{D}$ (with n points) by using a $2^{\lfloor \log_4 N/B \rfloor} \times 2^{\lfloor \log_4 N/B \rfloor}$ grid where there are $4^{\lfloor \log_4 N/B \rfloor} \leq N/B$ cells. Each cell corresponds to an intended partition, which means there are at most $\lceil N/B \rceil$ partitions. To achieve balanced partitions, the grid cells do not have the same area but are intended to enclose the same number of data points. Such cells are obtained by first sorting and partitioning by the x-coordinates of the data points to form $2^{\lfloor \log_4 N/B \rfloor}$ columns each with the same number of data points (except possibly for the last column). Then, each column is further partitioned into $2^{\lfloor \log_4 N/B \rfloor}$ cells by the y-coordinates of the data points, again each with the same number of data points. In Figure 3.2, we illustrate the partitioning where $N = 8$ and $B = 2$ ($\lfloor \log_4 N/B \rfloor = 1$), and a 2×2 cell grid is used to partition the space. Since there are 32 data points in the space, eight points are grouped together in each cell.

Since the resultant grid is non-regular, locating the partition (i.e., a cell) that encloses a given query point is non-straightforward unless the boundaries of the cells are stored. Instead, RSMI assigns each partition an ID (the Z-value was used by RSMI) and trains an index model (called a "sub-model", which is a function) $f_{0,0}$ to predict the partition ID given a point. Not all points in a partition are required to be mapped to their assigned partition ID, as function $f_{0,0}$ may make errors in its prediction. After function $f_{0,0}$ is learned, the points are reassigned to different partitions based on the partition IDs predicted by $f_{0,0}$. In Figure 3.2, eight points are assigned to each partition at the start, while 4 and 16 points are predicted into partitions corresponding to the bottom left and the top right cells by $f_{0,0}$, respectively.

When a partition predicted by $f_{0,0}$ contains more than N points, a further partitioning is required following the procedure above. Repeating this step recursively until each partition satisfies the size limit N, a hierarchy of index models is formed by RSMI.

When a partition has fewer than N points, a leaf level model is trained to predict block IDs. For example, $f_{1,0}$, $f_{1,1}$, and $f_{1,2}$ in Figure 3.2 are all functions to predict block IDs, while $f_{1,3}$ is another function to predict further partitions.

3.3.3 Query Processing with ZM and RSMI

We summarize the point, range, and kNN query algorithms of RSMI in this section. ZM shares similar algorithms for point and range query processing (but it has no kNN algorithms), which are omitted for succinctness.

Point query processing. The point query algorithm of RSMI recursively invokes the models at the next level, which always takes the coordinates of the query point q as input. The process starts from the root sub-model, $f_{0,0}$, and it only visits one sub-model at each level. Let $f_{i,j}$ denote the sub-model visited at level i. At level $i+1$, the algorithm visits the sub-model $f_{i+1,j'}$ where $j' = f_{i,j}(q.cord)$. This procedure continues until a leaf model is reached, which predicts the block ID of q as $f(q.cord)$. The algorithm then examines the predicted block and its neighboring blocks, which are limited by $[f(q.cord) - min_{err}, f(q.cord) + max_{err}]$. Here, min_{err} and max_{err} are the bounds of the prediction errors, which are derived from learning f. If q is found within these blocks, a pointer to the point is returned. Otherwise, the algorithm considers that q is not in the structure.

Range query processing. The range query algorithm of RSMI takes the lower and upper corner points of the query range, q_l and q_h, as input. These points are then used to execute point queries to obtain their corresponding block IDs, $q_l.blk$ and $q_h.blk$. If either of these points is found, its corresponding block ID is used as the lower or upper bound for data scanning respectively. Otherwise, the lower and upper bounds are approximated using the RSMI prediction, $f(q_l.cord) - min_{err}$ and $f(q_h.cord) + max_{err}$, respectively. Then, the algorithm scans the blocks within the lower and upper bounds and filters the points within the query range q.

When either q_l or q_h is not found in the dataset, RSMI may not identify the correct bounds for the query result, even with the help of the query error bounds. Thus, RSMI may return inaccurate results for range queries. For accurate range query processing, RSMI offers a variant where each sub-model is stored with an MBR to bound all points

indexed in the sub-tree corresponding to the sub-model. Range query processing over this RSMI can be done with an R-tree-like query procedure.

***K* nearest neighbor query processing.** The kNN query algorithm of RSMI uses a priority queue Q to store the nearest neighbors (NNs) of the query point q by their distances to q. To estimate the initial search region, the algorithm assumes that a search region of k/n around q will contain k points if the n data points in the dataset $\mathcal{D}$ are uniformly distributed in unit space. Therefore, it uses a rectangular search region centered at q with a width of $\alpha_x\sqrt{k/n}$ and a height of $\alpha_y\sqrt{k/n}$. The skew parameters α_x and α_y adjust the search region size in both dimensions according to data skew, and their values are set to 1 for uniform data.

The algorithm performs a range query using the initial search region to find the data blocks containing the points in the area. For each block $\mathcal{B}_i$ that is closer to q than the current k-th NN, $Q[k]$, the algorithm goes through every point $p \in \mathcal{B}_i$ and adds p to Q if it is closer to q than $Q[k]$. If the number of points in Q is less than k, the algorithm doubles the width and height of the search region. If the distance between q and $Q[k]$ ($dist(q, Q[k])$) is greater than $\sqrt{width^2 + height^2}/2$ (i.e., $Q[k]$ is outside the current search region), the algorithm also enlarges the width and height to be $2 \cdot dist(q, Q[k])$. If neither condition is met, the algorithm terminates and returns the first k points in Q.

Like that in range query processing, RSMI may also offer inaccurate results in kNN query processing, and the MBR-based variant can be used if accurate results are required.

3.3.4 Update Handling for ZM and RSMI

ZM does not come with native update support. We implement an update strategy for it as follows. Since ZM follows a hierarchical structure, we only enable update handling for the bottom level as it is used to predict the storage locations, while the models in upper levels are used to predict a model at the bottom level. When inserting a point p, we follow the point query procedure of ZM, which uses the upper level models to recursively predict a leaf level model f for the final storage location prediction (i.e., a block ID, assuming the same external data block storage as in RSMI). Then, p is inserted into the corresponding block, if the block has the capacity to accommodate p. Otherwise, a data block splitting procedure is needed. This involves creating a new block and moving half

of the existing data points from the existing block to the new block, thereby making room for the new data point p. The ID of the new block is one larger than the existing one. All the following blocks move backward and their IDs are increased by one. To guarantee that p can be retrieved later on, we need to enlarge the error bound for model f because there can be a position shift after inserting p. When deleting a point p, we also use a point query to find the point and simply label it as deleted.

RSMI supports both data insertion and deletion. To insert a new point p, RSMI first performs a point query to locate the target block for p. If there is enough space, p will be added to the target block. Otherwise, RSMI creates a new block $\mathcal{B}$ and inserts it as the next block after the target block (RSMI creates a linked list of all data blocks to enable data scan for query processing). Block $\mathcal{B}$ is marked as an inserted block, which does not contribute to the error bounds during query processing. It then recursively updates the MBRs of the ancestor models that index p to complete the insertion. The time complexity of insertion is $O(hM + (max_{err} + min_{err})B + IB + h)$, which includes the point query time to locate p (i.e., $O(hM)$), the search time on blocks created after the target block by previous insertions (i.e., $O((max_{err} + min_{err})B + IB)$), and the MBR update time (i.e., $O(h)$). Here, h is the height of RSMI, $O(M)$ is the cost of model prediction, and I is the number of new blocks created due to insertions.

To delete a point p, RSMI also performs a point query to locate the target block containing p, swap p with the last point in the block, and mark p as deleted. It then recursively updates the MBRs of the ancestor models. It does not delete a disk block as it is required to maintain the validity of the error bounds. The time complexity of deletion is $O(hM + (max_{err} + min_{err})B + IB + h)$, which includes the point query time to locate p and the MBR update time.

Insertions and deletions affect the layout of the learned index and therefore impact query performance. The cost of each point query is increased by $O(IB)$. To maintain a high query efficiency, a periodic index rebuild may be performed, for example, overnight.

3.4 Implementation

Both the ZM and RSMI indices are implemented through a series of sub-models (such as $f_{i,j}$ in the case of RSMI). Each sub-model is structured as an FFN formed by an input

layer, a hidden layer, and an output layer. The number of nodes in the hidden layer is calculated as the average of the number of input attributes and output classes. For example, in RSMI, with two coordinates as the input and 100 different block ID values as the output, the hidden layer uses 51 nodes. The sigmoid function is employed as the activation function within the hidden layer.

The training process for these FFNs commences from the root sub-model and progresses level by level. Each FFN goes through 500 training epochs with a learning rate set at 0.01. RSMI has a partitioning threshold (N), which is initially set at 10,000, implying that a leaf model can accommodate a maximum of 10,000 data points.

The design of RSMI allows it to adaptively learn the number of levels and sub-models at each level based on each dataset. To simplify the model training process, we normalize both point coordinates and block IDs to fit within a unit range.

We use HC for ordering in RSMI, as it consistently outperforms the ZC in the query performance.

3.5 Experiments

We report on the experimental results of our implementation of ZM and RSMI, comparing them with traditional spatial indices in terms of index build time, query processing, and update efficiency.

3.5.1 Experimental Setup

The experiments are done on a computer running 64-bit Ubuntu 20.04 with a 3.60 GHz Intel i9 CPU, 64 GB RAM, and a 1 TB hard disk. We use *PyTorch* 1.4 [83] and its C++ APIs to implement the learned indices based on CPU. The traditional indices are implemented using C++ (except for the RR*, which was implemented in C [10]) based on CPU.

Index structures. We compare the following indices:

- Z-order model [103] (**ZM**): This is our implementation of the ZM index. Our implementation is a recursive version with three levels that use 1, $\sqrt{n/B^2}$, and n/B^2 sub-models, respectively.
- Recursive Spatial Model Index [84] (**RSMI**): This is our implementation of the RSMI index. Our implementation uses $N = 10,000$ as the partition threshold.
- Grid File [72] (**Grid**): This index partitions the data space with a regular grid, assigns data points to cells based on their coordinates, and stores the data points by their cells. We use a $\sqrt{n/B} \times \sqrt{n/B}$ grid, i.e., B points (one block) per cell under a uniform distribution.
- K-D-B-tree [87] (**KDB**): This index implements a kd-tree [13] with a B-tree structure to support block storage.
- Rank space-based R-tree [85] (**HRR**): This is an R-tree bulk-loaded using the rank space technique and an HC for the ordering. This R-tree offers state-of-the-art range query performance.
- Revised R*-tree [10] (**RR***): This is an improvement of the R*-tree, which has shown strong query performance.

KDB, HRR, and RR* have up to five levels in the experiments (on datasets with up to 128 million points).

Implementation details. We use the original implementation of HRR and RR*. For Grid, KDB, ZM, and RSMI, no source code is available. We use the static component of a Grid File for moving objects [46] for Grid. We implement KDB following the original paper. We implement ZM and RSMI as described in Section 3.4. We run all indices and algorithms in the main memory for ease of comparison (it is straightforward to place the data blocks in external memory).

For all structures, we use data blocks with a capacity of 100, i.e., $B = 100$. The R-tree and K-D-B-tree leaf nodes (blocks) and the Grid File blocks can store up to 100 points each, while the internal nodes of the tree structures store up to 100 MBRs. No buffering is assumed.

Datasets. We use two real datasets: **Tiger** and **OSM**. Tiger contains over 17 million rectangles (950 MB in size) representing geographical features in 18 Eastern states of

TABLE 3.1: Parameters and their values

Parameter	Setting
Distribution	Uniform, Normal, **Skewed**
n (million)	1, 2, 4, 8, **16**, 32, 64, 128
Query range size (%)	0.0006, 0.0025, **0.01**, 0.04, 0.16
Query range aspect ratio	0.25, 0.5, **1**, 2, 4
k	1, 5, **25**, 125, 625
Inserted points (%)	10, 20, 30, 40, 50
Deleted points (%)	10, 20, 30, 40, 50

the USA [97]. We use the centers of the rectangles as our data points. OSM contains over 100 million points (2.2 GB in size) in the USA extracted from OpenStreetMap [76].

We generate three groups of synthetic datasets, **Uniform**, **Normal**, and **Skewed**, with up to 128 million points (2.5 GB in size). The synthetic data falls into the unit square. Uniform and Normal datasets follow uniform and normal distributions, respectively. Skewed datasets are generated from uniform data by raising the y-coordinates to their powers y^α ($\alpha = 4$ by default), following HRR [85].

As summarized in Table 6.5, we vary the dataset size n, the query range size and aspect ratio, the query parameter k, and the percentages of points inserted or deleted, respectively. Default settings are shown in boldface.

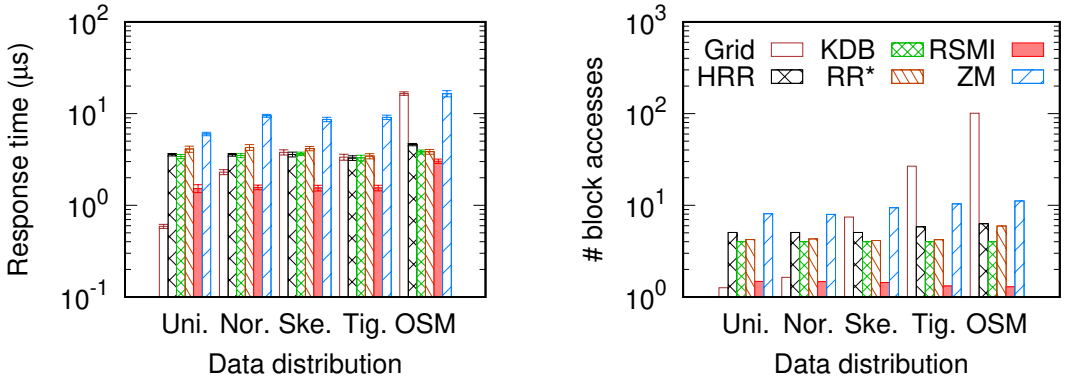
We generate queries that follow the data distribution for each set of query experiments and report the average **response time** and **number of block accesses** per query. For each type of query, we execute the queries with 10 different sequences and report the average time and the variance in histogram-based figures (i.e., Figure 3.3(b), Figure 3.7(a), and Figure 3.11(a)). The block accesses serve as a performance indicator for an external memory-based implementation of the algorithms.

3.5.2 Results

We report query efficiency through point, range, and kNN query processing, followed by scalability via the impact of N, and update handling.

3.5.2.1 Point Queries

We use all data points in each dataset as the query points and report the average performance per point query.



(A) Query time (B) # block accesses

FIGURE 3.3: Point query vs. data distribution

Varying the data distribution. As Figure 3.3 shows, RSMI offers the best query performance on both real (Tiger and OSM) and synthetic (Normal and Skewed) data. It improves the query time by at least 1.3 times and up to 5.5 times compared with the competing techniques, i.e., 3.0 μs vs. 3.8 μs (KDB) and 16.5 μs (Grid) on OSM. It also improves the number of block accesses by at least a factor of 5.3 (1.4 vs. 7.4 for RSMI vs. Grid on Skewed) and up to 77.5 times (1.3 vs. 100.8 for RSMI vs. Grid on OSM). Grid works better on Uniform data, as such data can be partitioned relatively evenly across the cells and take full advantage of Grid. Note that Grid has much higher numbers of block accesses than the other techniques while its running time may not seem as high, because it features a simple checking procedure per block, and the blocks are stored in memory.

TABLE 3.2: Prediction error bounds (min_{err}, max_{err})

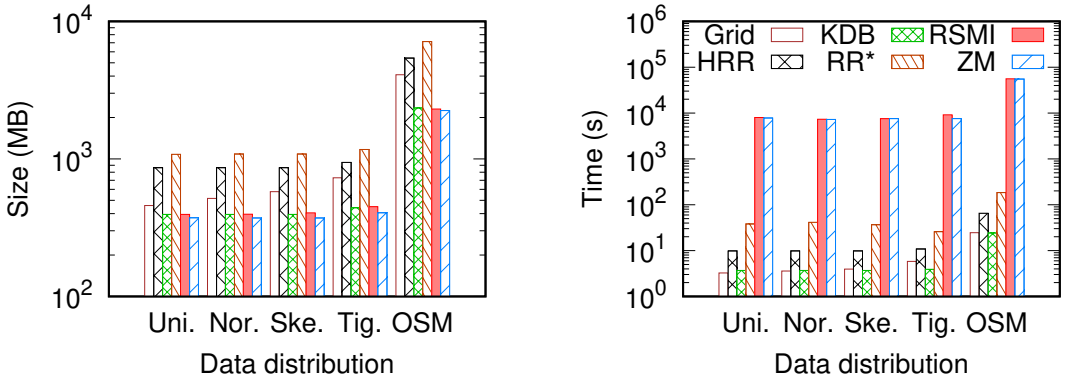
	Uni.	Nor.	Ske.	Tig.	OSM
ZM ($\times 10^4$)	(1.9, 1.9)	(1.2, 5.5)	(0.9, 3.7)	(0.7, 0.7)	(7.4, 11)
RSMI	(43, 82)	(37, 91)	(55, 78)	(70, 69)	(89, 74)

The strength of RSMI comes from its fast and accurate predictions. Table 3.2 summarizes the maximum prediction errors (min_{err}, max_{err}) of ZM and RSMI. For example, min_{err} and max_{err} of RSMI on Skewed are 55 and 78 blocks, respectively. The average number of block accesses is much lower than these bounds, e.g., 1.4 for RSMI on Skewed.

ZM offers less accurate predictions due to its design. On Skewed, its prediction error can be as large as 3.7×10^4 blocks, and its average number of block accesses is 8.1 (binary search on the Z-values is used to reduce the number of block accesses). KDB, HRR, and RR* incur fewer block accesses than ZM does. However, they still need to access inner nodes. Also, due to overlapping node MBRs, the R-trees may need to access multiple nodes at each tree level.

We further report the *average depth* of RSMI, i.e., the average number of sub-models invoked to reach a data block, which are 3.11, 3.26, 3.30, 3.04, and 4.01 on Uniform, Normal, Skewed, Tiger, and OSM, respectively. RSMI only needs 3 or 4 function invocations to locate a data block. KDB, HRR, and RR* have a depth of 3 on the first four datasets and 4 on OSM (excluding the data block level). They need to scan 3 or 4 nodes (maybe more for R-trees due to overlapping MBRs) to locate a data block, which is slower. Grid and ZM have fixed depths of 1 and 3. They suffer from the number of block accesses as discussed above.

Another observation concerning ZM vs. RSMI is that ZM suffers more in terms of block accesses and less in terms of response time. This is because ZM can quickly skip a data block accessed by testing whether the Z-value of the query point belongs to the Z-value range of the block. Its processing time per block is smaller than that of RSMI.



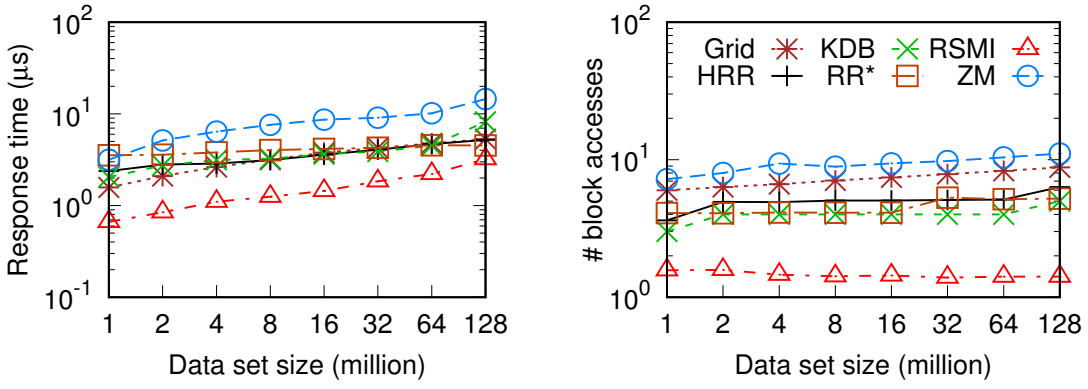
(A) Index size (B) Index build time

FIGURE 3.4: Index size and build time vs. data distribution

Figure 3.4(a) shows the index size. The learned indices are the smallest because they only store the data blocks and the (small) sub-models. In contrast, Grid stores the data blocks and a cell table that maps grid cells to the corresponding data blocks; KDB, HRR, and RR* store the data blocks (leaf nodes) and the internal nodes. RSMI has a

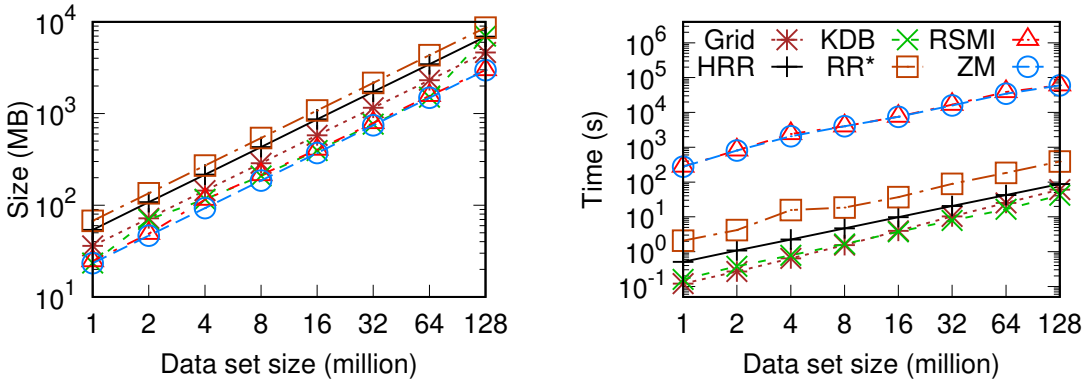
slightly larger index size than ZM due to its slightly larger number of sub-models. This is because RSMI is built adaptively based on the data distribution, while the number of sub-models in ZM is determined by the number of data points (cf. Section 3.5.1). For example, on OSM, RSMI has 10,445 sub-models, while ZM has 10,101 (2,303 MB vs. 2,244 MB, i.e., 2.6% larger). RR* is the largest because its nodes are less compact. HRR is also larger than RSMI because it uses two extra B-trees for its rank space mapping [85].

The advantages of RSMI in query performance and index size come with a higher build time (Figure 3.4(b)), which is a common characteristic of learned indices. RSMI can be trained within 16 hours on OSM (over 100 million points), which is justified given its query performance. This training time assumes CPUs. If GPUs are used, we can reduce the training time by over 74%. For example, training RSMI on a Skewed dataset with 128 million points takes 60,514 seconds on a CPU. This can be reduced to 15,698 seconds on an RTX 2080 Ti GPU. *This time is still about an order of magnitude higher than those required to build the traditional indices, which is our core motivation to study efficient algorithms to build learned spatial indices in the later chapters.* ZM is faster than RSMI at training, because RSMI needs to first sort and partition the data points *each* time it learns a sub-model. ZM only sorts the data points *once* for learning *all* sub-models. Among the traditional indices, HRR is bulk-loaded, which only takes a few rounds of sorting and data scans. This is faster than RR*, which is created by means of top-down insertions. Grid and KDB are the fastest in index-building due to their simple sorting-based construction.



(A) Query time (B) # block accesses

FIGURE 3.5: Point query vs. dataset size



(A) Index size (B) Index build time

FIGURE 3.6: Index size and build time vs. dataset size

Varying the dataset size. In Figure 3.5, we vary the dataset size using Skewed data. Results on the other datasets show similar patterns and are omitted for succinctness. The same applies in the remaining experiments.

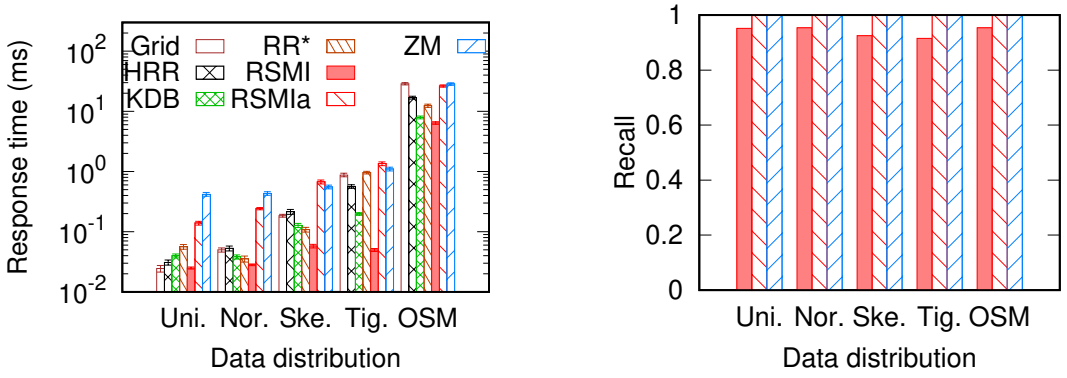
As expected, the query costs increase with the dataset size. RSMI offers the lowest query costs, and the advantage is up to 5.8 times (1.3 μs vs. 7.6 μs for RSMI vs. ZM with 8 million points). We observe a slight drop in the number of block accesses for RSMI. When there are more points, RSMI may learn a structure with more levels, where the leaf models are more compact, yielding more accurate predictions and fewer block accesses. The query time still increases as there are more levels – the average depth increases from 2.49 to 4.46 for 1 to 128 million points, and the maximum depth is 10. These findings indicate that RSMI is scalable.

The index sizes and build times also increase with the dataset size, as shown in Figure 3.6. RSMI is consistently small in size, while its build time does not grow drastically.

3.5.2.2 Window Queries

We generate 1,000 range queries under each setting and report the average cost per query. Since the number of block accesses aligns with the query response time, we omit coverage of the number of block accesses. Also, index size and build time are independent of the query type and are omitted hereafter. As learned indices may offer approximate range query answers (without false positives, cf. Section 3.3.3), we report their **recall** — the number of points returned over the cardinality of the ground truth answer.

We add one more technique called **RSMIa** to the comparison. It offers accurate query answers by performing an R-tree-like traversal by utilizing MBRs associated with the sub-models in RSMI, as described in Section 3.3.3.

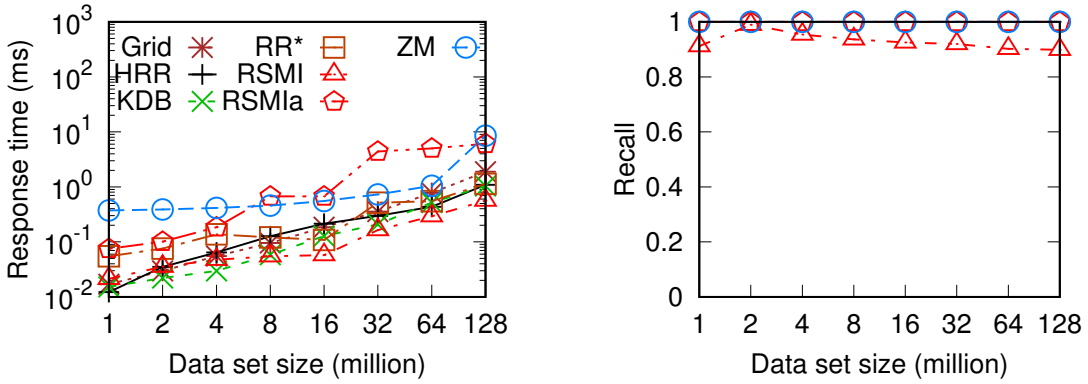


(A) Query time (B) Recall

FIGURE 3.7: Range query vs. data distribution

Varying the data distribution. Figure 3.7 reports the range query costs across different datasets. As for point queries, RSMI is the fastest for range queries except for on Uniform data where Grid is slightly faster (0.025 ms vs. 0.024 ms). This is because Grid can easily locate the cells and hence the data blocks that overlap with the query range. On non-uniform data, Grid may have cells that only partially overlap with the query range while containing many blocks (and false positives) to be filtered for the query. On such datasets, RSMI outperforms the traditional indices by at least a factor of 1.33 (6.4 ms vs. 8.0 ms for RSMI vs. KDB on OSM) and up to 17 times (0.05 ms vs. 0.85 ms for RSMI vs. Grid on Tiger). RSMI also outperforms ZM by 4.4 times on OSM (6.4 ms vs. 28.5 ms) and by over an order of magnitude on the other four datasets. Meanwhile, RSMIa outperforms ZM on Uniform, Normal, and OSM, and its query performance is comparable to those of the R-tree indices on the real datasets Tiger and OSM. These findings show the applicability of RSMIa when accurate query answers are needed.

Next, we consider the recall of the learned indices. RSMIa has 100% recall as it uses MBRs for query processing. ZM is more accurate than RSMI. It uses ZCs, where the bottom left and top right corners of a query range bound the search region better than all four corners of the query range in RSMI, which uses HCs. However, RSMI has much lower query costs. It also has a consistently high recall of over 91.4% and up to 95.4% on Normal data (Figure 3.7(b)).

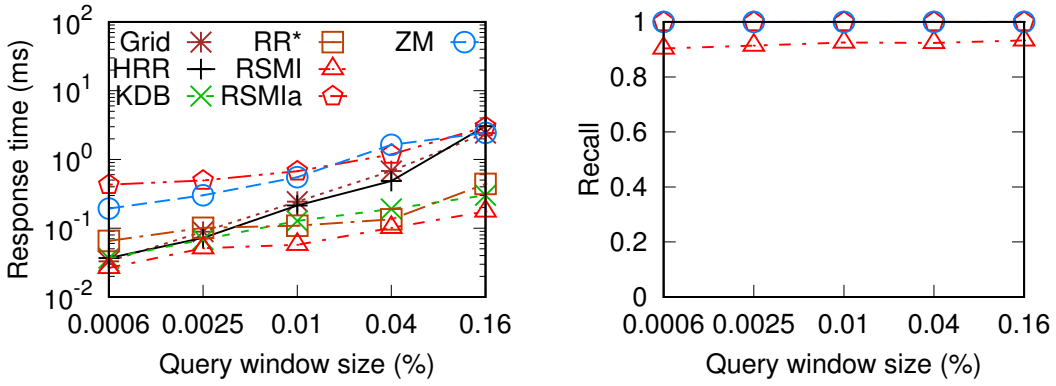


(A) Query time (B) Recall

FIGURE 3.8: Range query vs. dataset size

Varying the dataset size. When the dataset size varies (Figure 3.8(a)), RSMI is the fastest except for on datasets with fewer than 4 million points, where KDB is slightly faster. KDB creates non-overlapping data partitions, which benefits query performance. However, as the dataset grows, the partitions become long and thin due to the block size limit (especially on Skewed). This leads to many tree nodes overlapping with queries and high query costs. RSMIa is faster than ZM when the dataset size is below 8 million or exceeds 128 million. It queries 128 million points in just 6.1 ms.

Figure 3.8(b) shows the recall of RSMI, which is consistently high and is over 89.8% for 128 million points. The recall drops slightly as the dataset size increases, since it is more difficult to train an accurate prediction model on more points.



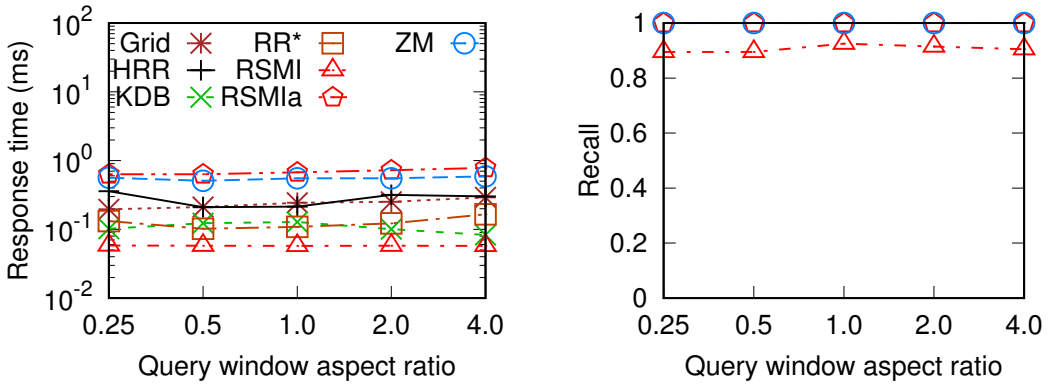
(A) Query time (B) Recall

FIGURE 3.9: Range query vs. query range size

Varying the query range size. We vary the query range size from 0.0006% to 0.16% of the data space size. As Figure 3.9 shows, the query times grow with the query range

size since more points are queried. The relative performance in both the query time and the recall among the indices is similar to that observed above. RSMI offers highly accurate answers (over 90.3%) and the lowest query times, thus showing robustness for range queries in varying settings.

Varying the query range aspect ratio. We further vary the query range aspect ratio from 0.25 to 4.0. Figure 3.10 shows that the aspect ratio a is less impactful than the query range size. We conjecture that this is because the query costs are averaged over 1,000 queries that are positioned following the data distribution. Every set of 1,000 queries of a given aspect ratio may cover a similar set of data points overall, and hence has a similar average query cost. RSMI again outperforms all competitors and is at least 1.4 times faster (0.058 ms vs. 0.083 ms for RSMI vs. KDB when the aspect ratio is 4) than the other structures, and its recall exceeds 89.4%.



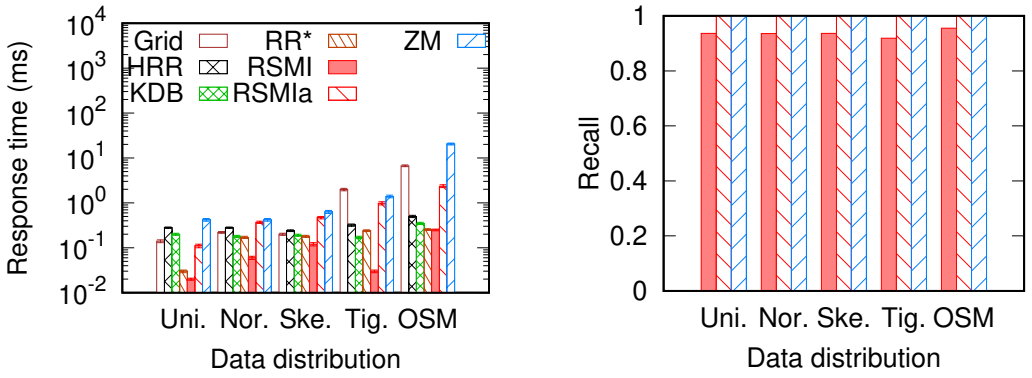
(A) Query time (B) Recall

FIGURE 3.10: Range query vs. query range aspect ratio

3.5.2.3 KNN Queries

We generate 1,000 kNN queries under each setting and report the average query time and recall. Here, the recall refers to the number of true kNN points returned divided by k. This is the same as the precision. ZM does not come with a kNN algorithm, and we use the kNN algorithm of RSMI for it. For the other indices, we use the best-first algorithm [88] to execute kNN queries.

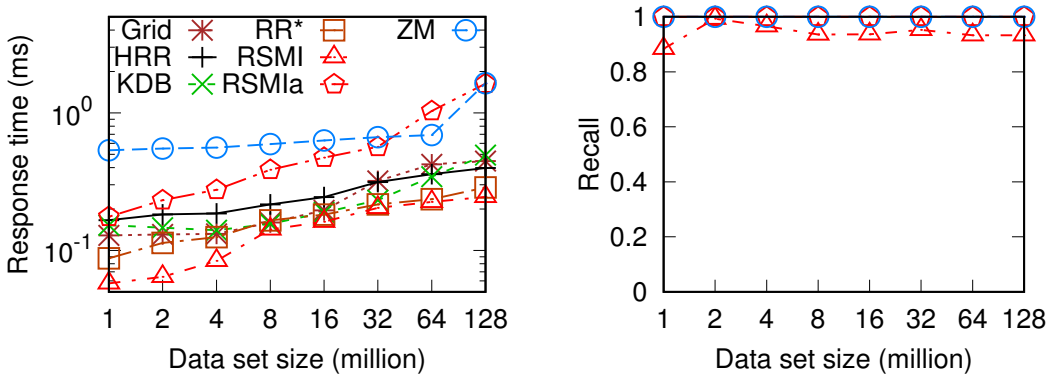
Varying the data distribution. Figure 3.11 shows that RSMI is also the fastest for kNN queries. It outperforms ZM by up to 46 times (0.03 ms vs. 1.38 ms on Tiger). This is



(A) Query time (B) Recall

FIGURE 3.11: KNN query vs. data distribution

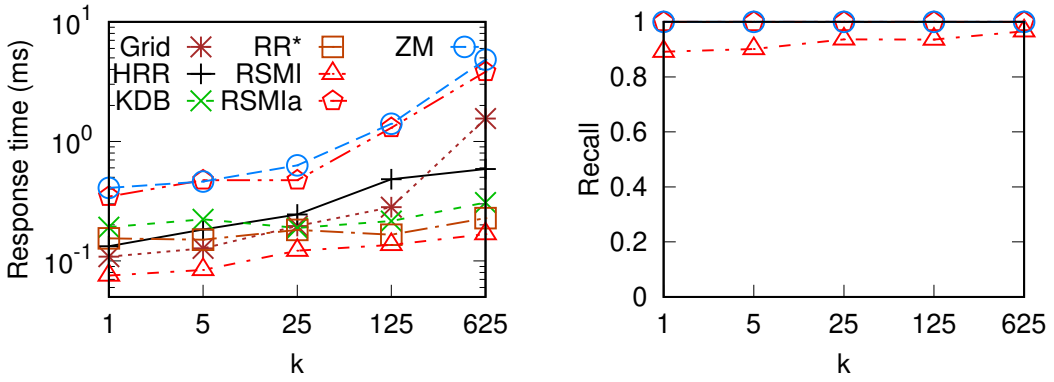
because both techniques use range queries for kNN queries, where RSMI is much faster. RSMI also outperforms the other indices. For Grid, the kNNs may spread in multiple cells which makes it uncompetitive. The other traditional indices require tree traversals and accesses to possibly many tree nodes. In terms of recall, RSMI is again very close to ZM and is over 91.8%. This is able to show the applicability of RSMI to kNN queries.



(A) Query time (B) Recall

FIGURE 3.12: KNN query vs. dataset size

Varying the dataset size. In Figure 3.12, we vary the dataset size. RSMI again yields high recall and the fastest query time, while RSMIa produces accurate answers and is faster than ZM (except when $n = 64$ million). The query times grow with the dataset size. When $n = 128$ million, RSMI is over an order of magnitude faster than ZM. The recall of RSMI also decreases slightly with the dataset size but it stays above 88.3%. This is in line with the observations for range queries where the dataset size is varied (cf. Figure 3.8).



(A) Query time (B) Recall

FIGURE 3.13: KNN query vs. k

Varying k. In Figure 3.13, we vary the query parameter k from 1 to 625. We see that the query costs increase with k, which is expected as more blocks and data points are examined. The high efficiency and recall (89.1% to 96.5%) of RSMI indicate that it scales to large k values.

3.5.2.4 Impact of RSMI Partition Threshold N

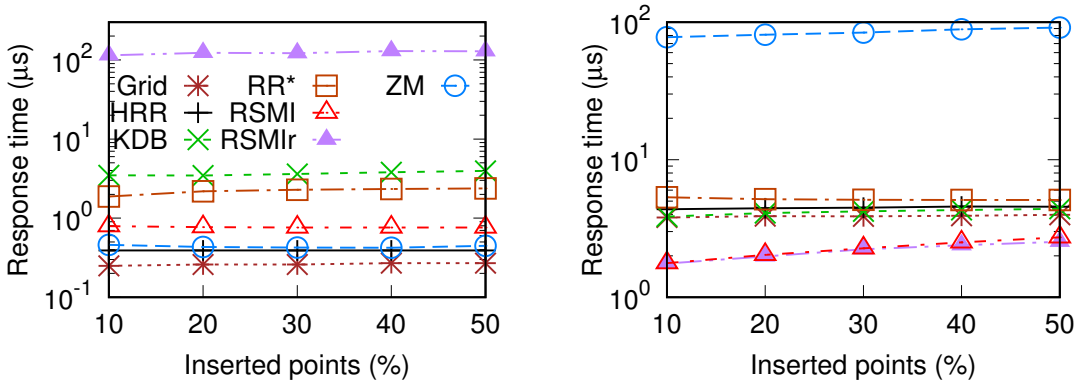
TABLE 3.3: Impact of N

N	2,500	5,000	10,000	20,000	40,000
Index build time (s)	10,997	8,215	7,553	7,602	7,161
Height	9	5	4	4	3
Index size (MB)	488.5	425.5	405.5	398.9	391.3
Query # block accesses	1.28	1.35	1.44	1.47	1.52
Query time (μs)	1.79	1.59	1.44	1.47	1.49

We first study the impact of N to optimize RSMI. As Table 3.3 shows, when N increases (from 2,500 to 40,000), the index build time, height, and index size all decrease overall. This is because a larger N means more points in each partition and fewer partitions, leading to fewer levels and sub-models, and hence shorter training times. Meanwhile, the average number of block accesses per point query increases, because the leaf models become less accurate. The point query time, however, first decreases and then increases again. This results from a combined effect of fewer sub-models to compute while more data blocks to examine as N increases. The query time is the shortest when $N = 10,000$. We use this N value in the rest of the experiments.

3.5.2.5 Update Handling

We further examine the impact of data updates. ZM does not come with update algorithms, so we adapt those of RSMI for it. We initialize the indices with the default dataset, insert (or delete) 10%n to 50%n data points, and query the updated indices with 1,000 queries. We report the average response time per insertion and the average response time per query. We also studied the impact of deletions but omit those for succinctness. We note however, that they replicate the performance figures of insertions.



(A) Insertion time (B) Point query time

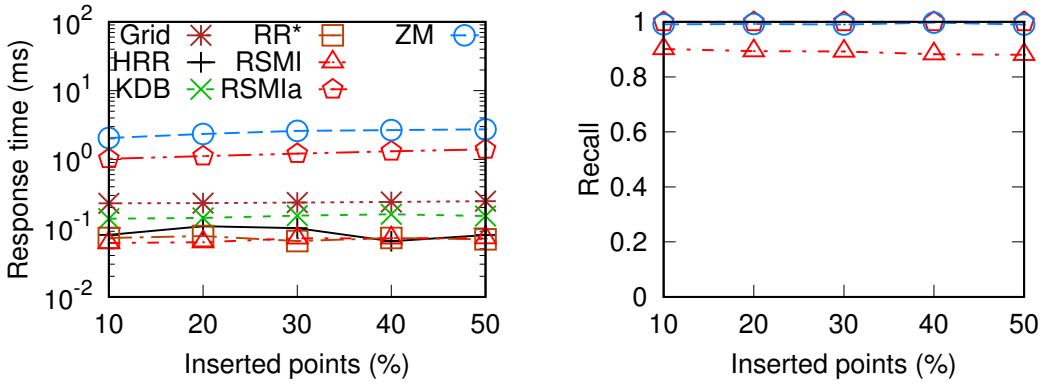
FIGURE 3.14: Insertion and point queries after insertions

Insertion. Figure 3.14(a) shows slowly increasing insertion times with more insertions, as the tree index height has not increased till 50%n insertions, while the learned indices keep appending new points to the end of the predicted blocks. Grid adds a new point p to the last block in the cell enclosing p, which is the fastest. RSMI insertions cost more than those of ZM, since it has more sub-models than ZM, and it takes more time to reach a data block for the insertion.

Figure 3.14(b) shows that insertions cause the point query times to increase. There are more points to query, and the indices become less optimal. ZM and RSMI are impacted the most as they have more blocks to scan with insertions. RSMI still yields the best query performance after 50%n insertions, i.e., 2.7 μs (RSMI) vs. 3.9 μs (Grid).

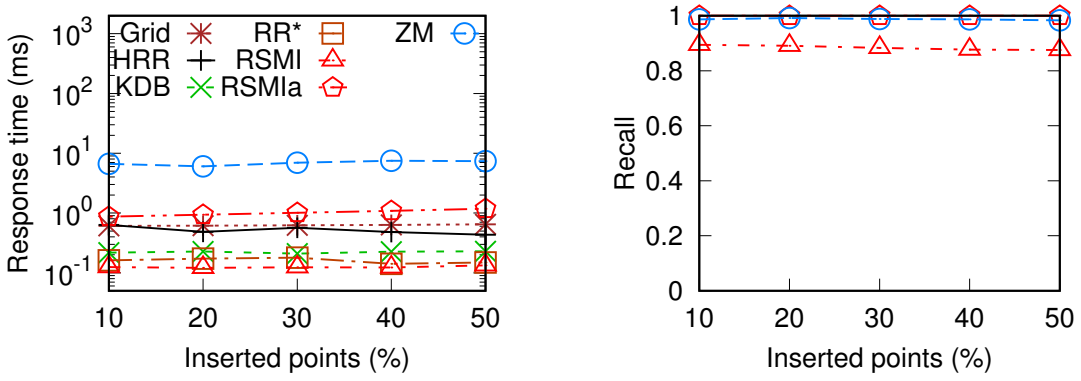
We further compare with an RSMI variant named **RSMIr**, which rebuilds the sub-models that exceeded the partition threshold after every 10%n insertions. RSMIr has an amortized insertion time of some 130 μs, which is the highest among the index structures tested, which is another motivation for us to study for more efficient algorithms to rebuild learned spatial indices. It also improves the point query performance, especially when

there are more insertions. A similar pattern is observed on range and kNN queries. We omit its curve in those figures for succinctness.



(A) Query time (B) Recall

FIGURE 3.15: Range queries after insertions



(A) Query time (B) Recall

FIGURE 3.16: KNN queries after insertions

For range queries (Figure 3.15(a)), RR* keeps adjusting MBRs to reduce overlaps, while HRR checks the newly created blocks with tree traversals. They now perform similarly to RSMI. For RSMI, the created blocks are linked after blocks with the same predicted location. When the first block does not contain a query range corner point, more blocks need to be checked, which increases the query time. For kNN queries (Figure 3.16(a)), as there are more points, the size of the initial search region of RSMI decreases. This helps RSMI retain the fastest query time. The recall of RSMI for both range and kNN queries consistently exceeds 87.5% (Figure 3.15(b) and Figure 3.16(b)). These results indicate the effectiveness of the update algorithm in maintaining the query performance of RSMI.

3.6 Summary

We implemented and studied the empirical performance of two early learned spatial indices ZM and RSMI. We showed that a straightforward extension of the learned index technique to spatial data (e.g., ZM) does not always lead to enhanced query performance, especially over large datasets which challenge the learning capacity of individual models. On the other hand, advanced learned spatial indices that learn not only index models but also data partitions (e.g., RSMI) have shown a strong advantage in query performance, outperforming traditional indices by over an order of magnitude.

However, the index building process of both ZM and RSMI is much slower than those of the traditional indices, e.g., over an order of magnitude slower on large datasets with over 100 million points. This impinges to practical applicability of learned spatial indices. The rest of the thesis aims to address this efficiency issue in index building.

Chapter 4

Efficiently Learning Spatial Indices

In Chapter 3, we have shown that learned spatial indices achieved highly efficient query processing. However, efficient building and rebuilding of such indices is a challenge. As model training for the learn spatial indices is expensive, building and rebuilding of learned spatial indices on large datasets can be time-consuming if performed by means of direct model training and retraining.

In this chapter, we propose a system named ELSI that enables the efficient building and rebuilding of learned spatial indices (as long as they follow two simple design principles). The core idea is to reduce the model (re-)building times by engineering reduced training sets that preserve key data distribution patterns. ELSI encompasses a suite of methods for constructing small and distribution-preserving training sets from input datasets. Further, given an input dataset, ELSI can adaptively select a method that produces a learned index with high query efficiency. Experiments on real datasets of 100+ million points show that ELSI can reduce the build times of four different learned spatial indices consistently (by up to two orders of magnitude) without jeopardizing query efficiency.

The work reported in this chapter has been published in the following paper: Guanli Liu, Lars Kulik, Christian S. Jensen, James Bailey, Jianzhong Qi. *Efficiently Learning Spatial Indices*, IEEE International Conference on Data Engineering (**ICDE**) 2023.

4.1 Overview

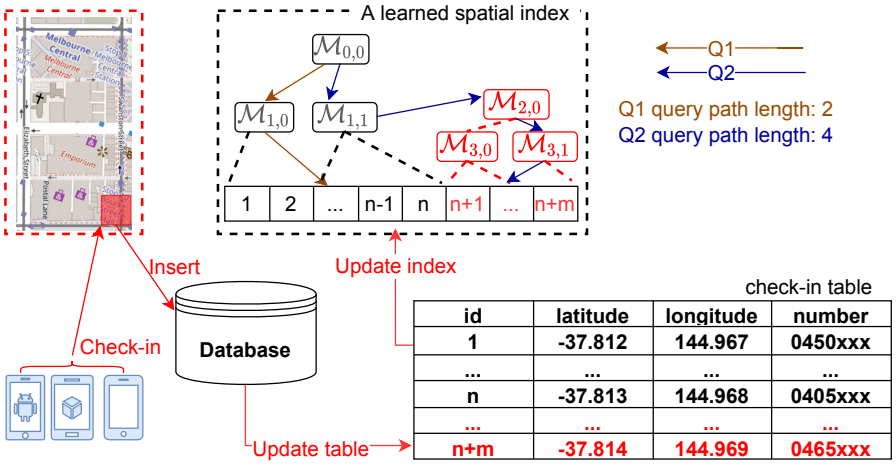


FIGURE 4.1: An unbalanced learned spatial index created by skewed insertions (red objects, best viewed in color).

While learned spatial indices target high query efficiency, their performance can degrade quickly with data updates, especially when the updates cause the data distribution to change. A straightforward solution is to frequently relearn the index function $\mathcal{M}$ but frequent relearning is prohibitively expensive. Learned spatial indices such as RSMI [84] and LISA [58] use additional models and data pages, respectively, to support data insertions without relearning the initially learned function $\mathcal{M}$, leading to increasingly sub-optimal query processing the more points are inserted.

For example, Figure 4.1 shows an RSMI index with three models $\mathcal{M}_{0,0}$, $\mathcal{M}_{1,0}$, and $\mathcal{M}_{1,1}$ learned initially on n check-in where $\mathcal{M}_{i,j}$ denotes the jth model at the ith index layer. After m skewed insertions (e.g., check-ins from a small region), three more local models $\mathcal{M}_{2,0}$, $\mathcal{M}_{3,0}$, and $\mathcal{M}_{3,1}$ (in red) are built by RSMI, causing an unbalanced structure and more function invocations for some queries. For example, while query Q1 only needs two model invocations (brown arrows), query Q2 needs four (blue arrows).

To avoid degrading performance, index rebuilds are required, which is also done for one-dimensional indices in database systems such as Oracle and SQL Server [3, 71]. However, (re)building a learned spatial index on a large dataset is challenging due to the high cost of learning function $\mathcal{M}$. LISA and RSMI report hours to learn indices on 50 to 100 million points [58, 84].

To enable efficient index (re)building, we propose a system for Efficient Learning of Spatial Indices, named ELSI. As Figure 4.2 shows, our core idea is that, given a spatial

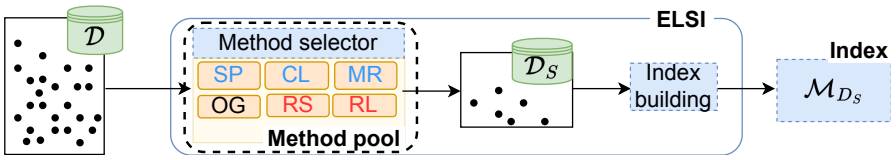


FIGURE 4.2: An overview of ELSI.

dataset $\mathcal{D}$, we compute a much smaller set $\mathcal{D}_S$ (e.g., $|\mathcal{D}_S|$ is at the million scale, while $|\mathcal{D}|$ is at billion scale) through ELSI, which preserves the data distribution of $\mathcal{D}$ in $\mathcal{D}_S$. We then learn an index model $\mathcal{M}_{\mathcal{D}_S}$ on $\mathcal{D}_S$ and use it for querying $\mathcal{D}$. Comparing with learning a model $\mathcal{M}_{\mathcal{D}}$ on $\mathcal{D}$, learning $\mathcal{M}_{\mathcal{D}_S}$ is much more efficient, since $|\mathcal{D}_S| \ll |\mathcal{D}|$, and the learned model $\mathcal{M}_{\mathcal{D}_S}$ is expected to retain high query efficiency on $\mathcal{D}$.

To ensure both build and query efficiency, ELSI offers a method pool of six *index building methods*. Five are for generating (or fetching a pre-generated) $\mathcal{D}_S$ and the other just uses the OriGinal (OG) dataset $\mathcal{D}$, which serves as a backup option. ELSI can learn to choose the most suitable method for $\mathcal{D}$ using a method selector, based on predicted index building and query costs. To improve the overall rebuilding efficiency, ELSI also predicts the times when rebuilding is needed.

Three of the six index-building methods are adapted from the literature: Sampling (SP) [59], Clustering (CL) [23], and MR (detailed in Chapter 5). SP generates $\mathcal{D}_S$ by *systematic sampling* [19] over $\mathcal{D}$. This is simple and efficient, but queries in sparse regions, where no data points have been sampled for model training, may experience poor performance. CL uses cluster centroids from $\mathcal{D}$ as $\mathcal{D}_S$. Its limitation is the time needed to build an index, since clustering a large dataset is expensive compared to sampling. MR generates synthetic datasets and pre-trains index models on them. Then $\mathcal{D}$ is indexed using a pre-trained model corresponding to the synthetic dataset that is the most similar to $\mathcal{D}$ according to their CDFs. Technically, MR does not generate $\mathcal{D}_S$ after receiving $\mathcal{D}$. Its query efficiency suffers if none of the synthetic datasets is sufficiently similar to $\mathcal{D}$.

Two of the six index-building methods are proposed: Representative Set (RS) and RL, to obtain better approximations $\mathcal{D}_S$ of $\mathcal{D}$ at reduced costs. RS recursively partitions the data space and selects a point from each final partition to represent the partitions and form $\mathcal{D}_S$. Each space partition, i.e., a cell, is divided into 2^d (d is the data dimensionality) cells until each cell has at most a preset number of points. For example, when $d = 2$,

RS forms a quadtree [31] partitioning. A point from each non-empty cell is selected to form $\mathcal{D}_\mathcal{S}$. This process is highly efficient, and every data point in $\mathcal{D}$ is approximated by a close-by point (i.e., sharing the same cell). RL, on the other hand, partitions the data space through a grid. Instead of selecting from points in each grid cell, it assumes empty grid cells at the start and learns to add points to a subset of the cells to form a $\mathcal{D}_\mathcal{S}$ that best approximates (the CDF of) $\mathcal{D}$. We formulate the search process of $\mathcal{D}_\mathcal{S}$ (i.e., a sequence of point adding/removal operations) as a Markov Decision Process (MDP) and apply reinforcement learning for the search. RL offers a high-quality approximation of $\mathcal{D}$ while the size of $\mathcal{D}_\mathcal{S}$ can be controlled by the grid resolution.

To summarize, this chapter makes the following contributions:

1. We propose ELSI – a system for efficient building and update (in terms of rebuild) of learned spatial indices. ELSI is the first such system that can support any learned spatial indices that follow the map-and-sort index paradigm and the predict-and-scan query paradigm (detailed in Section 4.2).

2. To index a dataset $\mathcal{D}$, ELSI learns to choose an index-building method that generates a small dataset $\mathcal{D}_\mathcal{S}$ resembling $\mathcal{D}$ and builds an index with $\mathcal{D}_\mathcal{S}$ for $\mathcal{D}$, based on a preference factor that balances index building cost and the subsequent query processing efficiency. We propose two index-building methods RS and RL based on space partitioning and reinforcement learning, both of which build learned indices of high query performance efficiently.

3. We integrate ELSI into four different learned spatial indices and report on extensive experiments on both synthetic and real data. ELSI improves the index build times by a factor of 70 on average. The resulting indices retain high query efficiency for both point and range queries, only the kNN query times differ by just 3%.

4.2 Problem Statement

Given a set $\mathcal{D}$ of n points $p_1, p_2, \ldots, p_n$ in d-dimensional Euclidean space ($d \in \mathbb{N}^+$ and $d \geq 2$), a learned spatial index learns an *index model* $\mathcal{M}$ that predicts the storage address of a point p_i given its coordinates. Our aim is a system that enables efficient

(re)building of learned spatial indices. The problem that the system aims to solve is defined as follows.

Definition 4.1 (Efficient Index Learning)**.** Given a dataset $\mathcal{D}$, we aim to compute a set $\mathcal{D}_\mathcal{S}$ where $|\mathcal{D}_\mathcal{S}| \ll |\mathcal{D}|$, and $\mathcal{D}_\mathcal{S}$ preserves the distribution of $\mathcal{D}$, such that an index model $\mathcal{M}_{\mathcal{D}_S}$ can be learned on $\mathcal{D}_\mathcal{S}$ efficiently, and $\mathcal{M}_{\mathcal{D}_S}$ can be used as model $\mathcal{M}$ to query $\mathcal{D}$ with little sacrifice in query efficiency.

System applicability conditions. We do *not* propose a new learned spatial index but instead propose a versatile system named ELSI that can be integrated into existing learned spatial indices and thus enable efficient (re)building for these. To use with ELSI, a learned spatial index must satisfy:

(1) Map-and-sort index paradigm: Points in $\mathcal{D}$ are mapped to a one-dimensional space and stored based on the sorted order in the mapped space. The index model $\mathcal{M}$ learns this storage order. This condition enables sampling in the mapped space to form $\mathcal{D}_\mathcal{S}$, as well as point ordering in the mapped space after $\mathcal{D}_\mathcal{S}$ is obtained.

(2) Predict-and-scan query paradigm: A point query q (which is the basis for more complex queries such as range queries) on $\mathcal{D}$ is processed by an index model invocation $\mathcal{M}(q)$ to predict the storage address of q, followed by a scan over the addresses in $[\mathcal{M}(q) - min_{err}, \mathcal{M}(q) + max_{err}]$. This condition guarantees query correctness (for point queries) when there are prediction errors.

Dataset similarity measurement. A core capability of ELSI is to quantify the similarity between $\mathcal{D}_\mathcal{S}$ and $\mathcal{D}$. For this purpose, we use the CDF of the search keys of the datasets. This is because the data points are sorted and stored in a certain order. *An index model effectively learns a mapping from the key values used for sorting to the sorted ranks of the points. This mapping corresponds to the CDF of the key values.* For example, ZM [103] sorts the points by Z-values of the points.

Let $\mathcal{K}(\mathcal{D}_\mathcal{S})$ and $\mathcal{K}(\mathcal{D})$ be the sets of key values of $\mathcal{D}_\mathcal{S}$ and $\mathcal{D}$, and let $cdf_{\mathcal{K}(\mathcal{D}_\mathcal{S})}(\cdot)$ and $cdf_{\mathcal{K}(\mathcal{D})}(\cdot)$ be their CDFs. The similarity between the key value distributions of $\mathcal{D}_\mathcal{S}$ and $\mathcal{D}$ is defined as follows.

Definition 4.2 (Similarity between the key value distributions of two datasets)**.** Given $\mathcal{D}_\mathcal{S}$ and $\mathcal{D}$, the similarity between the distributions of their key values is 1 minus the

maximum distance between the CDFs of $\mathcal{K}(\mathcal{D}_\mathcal{S})$ and $\mathcal{K}(\mathcal{D})$:

$$sim(\mathcal{D}_\mathcal{S}, \mathcal{D}) = 1 - \sup_{x \in \mathbb{R}} |cdf_{\mathcal{K}(\mathcal{D}_\mathcal{S})}(x) - cdf_{\mathcal{K}(\mathcal{D})}(x)| \tag{4.1}$$

Here, $\sup_{x \in \mathbb{R}} |cdf_{\mathcal{K}(\mathcal{D}_\mathcal{S})}(x) - cdf_{\mathcal{K}(\mathcal{D})}(x)|$ is the maximum distance between the two CDFs.

For succinctness, we call $sim(\mathcal{D}_\mathcal{S}, \mathcal{D})$ the similarity and $dist(\mathcal{D}_\mathcal{S}, \mathcal{D}) = 1 - sim(\mathcal{D}_\mathcal{S}, \mathcal{D})$ the dissimilarity between $\mathcal{D}_\mathcal{S}$ and $\mathcal{D}$ hereafter as long as the context is clear. This similarity metric is based on the *Kolmogorov–Smirnov* (KS) test [51], which is a non-parametric test that returns the maximum distance between the empirical CDFs of two datasets.

Computing $dist(\mathcal{D}_\mathcal{S}, \mathcal{D})$ (or $sim(\mathcal{D}_\mathcal{S}, \mathcal{D})$) can be done by a scan over $\mathcal{D}_\mathcal{S}$ and $\mathcal{D}$ to compute $cdf_{\mathcal{D}_\mathcal{S}}(x)$ and $cdf_{\mathcal{D}}(x)$ for every $x \in \mathcal{D}_\mathcal{S} \cup \mathcal{D}$. This takes $O(n_\mathcal{S} + n)$ time, where $n_\mathcal{S} = |\mathcal{D}_\mathcal{S}|$ and $n = |\mathcal{D}|$, assuming sorted sets. We use a more efficient algorithm that only scans $\mathcal{D}_\mathcal{S}$. For the i-th value $\mathcal{D}_\mathcal{S}[i]$ in $\mathcal{D}_\mathcal{S}$ (in key-value order), we run a binary search to find its rank j in $\mathcal{D}$ (i.e., $\mathcal{D}[j]$ is the first element in $\mathcal{D}$ no smaller than $\mathcal{D}_\mathcal{S}[i]$). We compute the absolute gap $|i/n_\mathcal{S} - j/n|$ and report the maximum gap for all $i \in [1, n_\mathcal{S}]$ as $dist(\mathcal{D}_\mathcal{S}, \mathcal{D})$. This reduces the time complexity to $O(n_\mathcal{S} \log n)$. Since $n_\mathcal{S} \ll n$, $O(n_\mathcal{S} \log n)$ is better than $O(n_\mathcal{S} + n)$ in practice.

The Earth Mover's Distance (EMD) [27] is another similarity measure. Computing EMD on $\mathcal{D}$ and $\mathcal{D}_\mathcal{S}$ takes $O(n^3 \log n)$ time (even the state-of-the-art approximation takes $O(dn)$ time [69]), which is too expensive for our system.

4.3 Proposed System

This section starts with an overview of ELSI and illustrates how to build an index with it, in Section 4.3.1. We then explain the two core ELSI components, build processor and update processor, in Section 4.3.2.

4.3.1 ELSI Overview

As shown in Figure 4.3, ELSI has a build processor and an update processor that facilitate the building and updates of a learned spatial index, which we call the *base index*.

Different base indices have different building procedures. Some may train a single index model on $\mathcal{D}$, while others may partition $\mathcal{D}$ and train an index model for each partition.

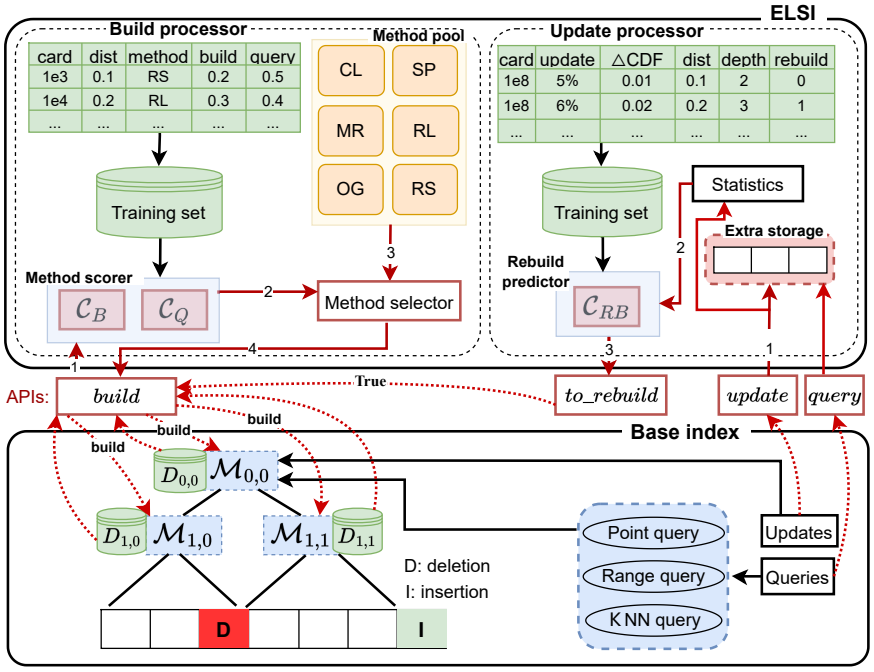


FIGURE 4.3: The ELSI system.

ELSI improves the build time of each index model while it does not interfere with the partitioning (cf. Figure 4.3 where ELSI helps build three models $\mathcal{M}_{0,0}$, $\mathcal{M}_{1,0}$ and $\mathcal{M}_{1,1}$).

Algorithm 1 summarizes the structure of ELSI and how ELSI builds a single index model for $\mathcal{D}$ (or a partition of it), i.e., the *build* API in Figure 4.3. Here, we use `typewriter` font to denote functions that come with a base index. The algorithm starts by mapping $\mathcal{D}$ to a one-dimensional space (line 1) and sorting it (line 2). The mapping function `map`$(\cdot)$ is supplied by the base index (e.g., the ZC mapping of ZM). Then, the sorted dataset is passed to our build processor (lines 3 and 4, detailed shortly), where we use a small dataset $\mathcal{D}_\mathcal{S}$ to approximate $\mathcal{D}$. We train an index model $\mathcal{M}_{\mathcal{D}_\mathcal{S}}$ on $\mathcal{D}_\mathcal{S}$ using the `train`$(\cdot)$ function of the base index (e.g., FFN training, line 5). Finally, we predict the address of every point in $\mathcal{D}$ using $\mathcal{M}_{\mathcal{D}_\mathcal{S}}$ and a model prediction function `predict`$(\cdot)$ from the base index (e.g., FFN prediction), and we record the maximum prediction errors min_{err} and max_{err} (line 6). Model $\mathcal{M}_{\mathcal{D}_\mathcal{S}}$ is returned as the index model $\mathcal{M}$ for $\mathcal{D}$ together with the error bounds (line 7).

After an index is built, queries are processed using the procedures that come with the base index. Updates go through our update processor (the *update* API in Figure 4.3, detailed shortly) that predicts the next time to rebuild (the *to_rebuild* API). When a rebuild is triggered, we use the *build* API to rebuild the base index.

Algorithm 1: build_index

Input: $\mathcal{D}$

Output: Model $\mathcal{M}$, error bounds min_{err} and max_{err}

```
1 D ← map(D);
2 D ← sort(D);
3 P ← get_build_method(λ, w_Q, D);
4 D_S ← compute_set(D, P);
5 M_{D_S} ← train(D_S);
6 min_err, max_err ← get_error_bound(M_{D_S}, D, predict(·));
7 return M_{D_S}, min_err, max_err;
```

Query error bounds. Existing learned spatial indices only offer empirical query error bounds but not theoretical bounds. ELSI enables efficient building of such indices and hence also offers empirical query error bounds. A few learned indices for one-dimensional data, e.g., PGM [6], use piece-wise linear approximation to approximate the CDF of one-dimensional data, which allows a theoretical bound on the query error based on the approximation error. Extending this idea to learned spatial indices is interesting but beyond the scope of our study.

4.3.2 ELSI Modules

ELSI has two main modules: the *build processor* and the *update processor*.

Build Processor. The build processor uses an *index building method selector* to select a method P from a *method pool* $\mathcal{P}$ for learning an index. The aim is to achieve both efficient index building and query processing for $\mathcal{D}$. A *method scorer* is employed, and the method with the maximum score is selected. The key element of method scorer is two FFNs (Component 2 in Figure 4.4), one that estimates the index building cost of a method P, denoted by $\mathcal{C}_B(\cdot)$, and one that estimates the query cost of the index built by P, denoted by $\mathcal{C}_Q(\cdot)$. We consider point query costs since point queries are building blocks for more complex queries. Costs of other query types, e.g., range queries, can also be considered.

The combined score $\mathcal{C}(P, \mathcal{D})$ of method P for $\mathcal{D}$ is then a weighted sum of the two costs, with a balancing parameter $\lambda \in [0, 1]$ and a query frequency parameter $w_Q \in [1, \infty)$ allowing user-defined trade-offs between the costs:

$$\mathcal{C}(P, \mathcal{D}) = \lambda \cdot \mathcal{C}_B(P, \mathcal{D}) + (1 - \lambda) \cdot w_Q \cdot \mathcal{C}_Q(P, \mathcal{D}) \tag{4.2}$$

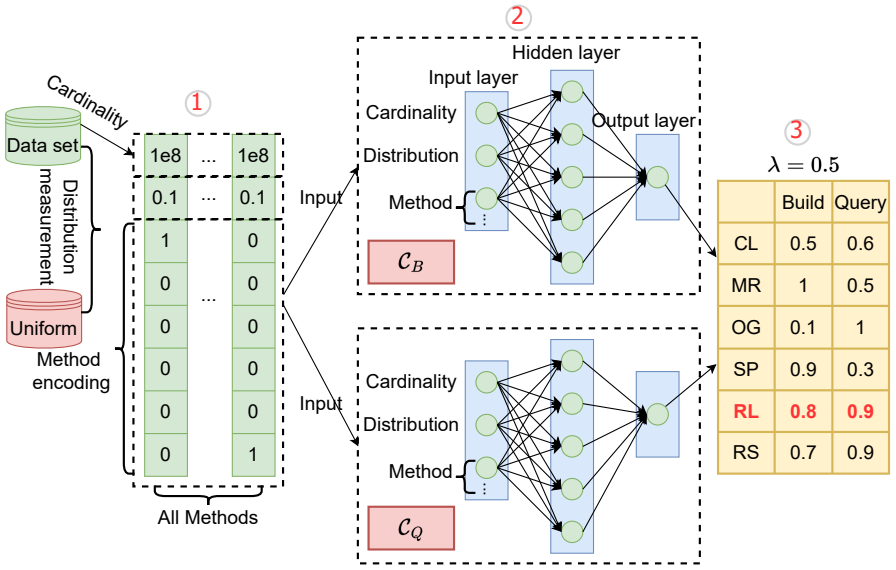


FIGURE 4.4: The ELSI index building method scorer.

Each FFN of the method scorer takes as input the ID (a one-hot embedding) of method P and the cardinality and distribution of $\mathcal{D}$ (Component 1 in Figure 4.4). We use $dist(\mathcal{D}_\mathcal{U}, \mathcal{D})$ to represent the distribution of $\mathcal{D}$ for fast online computation, where $\mathcal{D}_\mathcal{U}$ is a uniform dataset of the same size as $\mathcal{D}$. The method scorer is trained based on every method in the *method pool*, which contains the methods for shrinking the training set. Currently, we have six methods. The output of the two FFNs is a pair of predicted index building and query cost scores of P (Component 3 in Figure 4.4). They represent the predicted speedups that P can yield, compared with the original index building method of the base index.

Update Processor. The update processor provides default update procedures (a base index can also use its built-in update procedures). It uses a separate list to store newly inserted points and deleted existing points (or just marks the points as deleted if allowed by the base index). This list is scanned when processing a query, and the results are combined with, or are used to filter, those from the index to form the query result. A binary tree on the IDs of the updated points can be employed to reduce the query time.

As more and more updates are performed on the full dataset $\mathcal{D}$, the index models learned for $\mathcal{D}$ may become sub-optimal for queries. We then perform a rebuild, by triggering a full index build with the base index and the build processor. ELSI takes a learning-based approach to predict the time to rebuild, unlike traditional database systems such as Oracle [71], which uses empirical rules. We use a *rebuild predictor* (an FFN, the

to_rebuild API) $\mathcal{C}_{RB}(\cdot)$ that has a similar structure to that of the FFN in the method scorer (Figure 4.4) but outputs a binary value, indicating whether or not to rebuild. The rebuild predictor takes as input the cardinality and distribution of $\mathcal{D}$ (i.e., $dist(\mathcal{D}_{\mathcal{U}}, \mathcal{D})$ as above), the index depth, the update ratio, i.e., $|\mathcal{D}'|/|\mathcal{D}| - 1$ ($\mathcal{D}'$ is the updated dataset), and the *changes* to $\mathcal{D}$ caused by updates. We quantify the changes to $\mathcal{D}$ by the difference between the CDFs of $\mathcal{D}'$ and $\mathcal{D}$, i.e., $sim(\mathcal{D}', \mathcal{D})$, since a learned index learns the CDF of a dataset. When a full index is built (or rebuilt) on $\mathcal{D}$, we compute and store its CDF (an $O(n)$-sized vector). We maintain a copy of this CDF as the CDF of $\mathcal{D}'$ and compute $sim(\mathcal{D}, \mathcal{D}')$ as updates are processed. We run the rebuild predictor after every f_u, with f_u being a system parameter. Compared with the build processor, the model input has no information about the build method because the build processor concerns the build methods, while the rebuild predictor concerns the index itself.

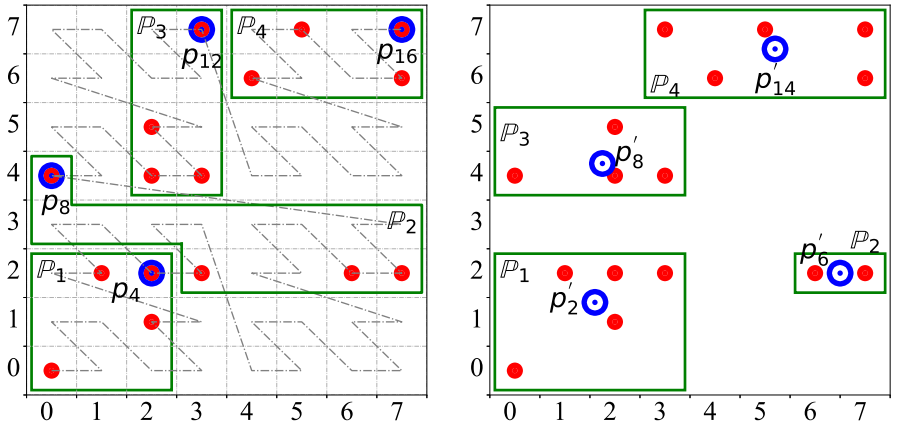
4.4 Index Building Methods

This section details the index-building methods in ELSI. As mentioned earlier, these methods do not build new types of indices but rather construct (or find) small datasets $\mathcal{D}_{\mathcal{S}}$ that resembles the input dataset $\mathcal{D}$. We present three methods adapted from the literature in Section 4.4.1 and propose two new methods in Section 4.4.2. We cover implementation details in Section 4.6.2.

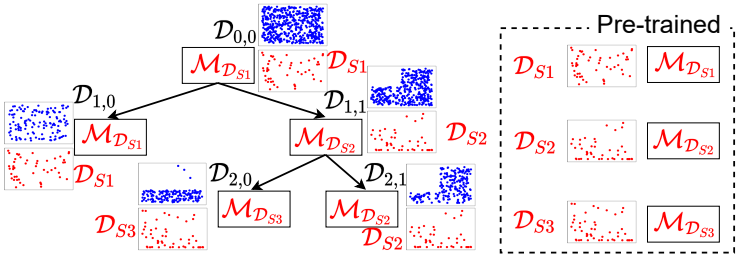
4.4.1 Adapted Methods

The first two adapted methods use sampling and clustering to construct $\mathcal{D}_{\mathcal{S}}$ from $\mathcal{D}$, while the third identifies a pre-generated dataset $\mathcal{D}_{\mathcal{S}}$ that matches $\mathcal{D}$.

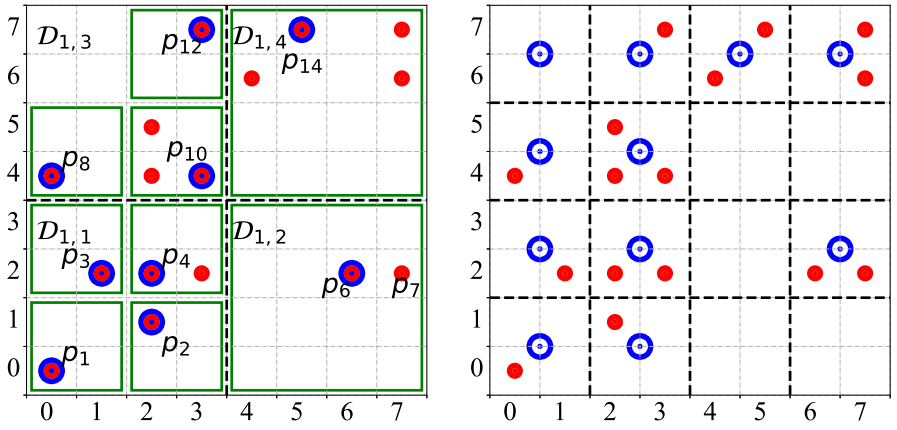
Sampling. Random sampling was used in a recent learned index [59] to reduce the training set size. The ELSI *sampling* method (SP) uses *systematic sampling* [19] instead. Given a sorted set $\mathcal{D}$, SP constructs $\mathcal{D}_{\mathcal{S}}$ by selecting a point after every $\lfloor 1/\rho \rfloor - 1$ points, where ρ is the sampling rate. $\mathcal{D}_{\mathcal{S}}$ then has $\rho \cdot n$ points, where $n = |\mathcal{D}|$. Figure 4.5((a)) shows an example with 16 points (in red) mapped and sorted with a ZC (SP also works with other mapped one-dimensional spaces). Given $\rho = 0.25$, $\mathcal{D}_{\mathcal{S}} = \{p_4, p_8, p_{12}, p_{16}\}$ (points in $\mathcal{D}_{\mathcal{S}}$ are in blue, also in the figures for subsequent methods).



(A) Sampling ($\rho = 0.25$) (B) Clustering ($C = 4$)



(C) Model reuse



(D) Representative set ($\beta = 4$) (E) Reinforcement learning ($\eta = 4$)

FIGURE 4.5: Examples of ELSI index building methods.

Let the i-th point in $\mathcal{D}$ be $\mathcal{D}[i]$, and let its nearest sampled point in $\mathcal{D}_\mathcal{S}$ be $\mathcal{D}[j]$ (i.e., the j-th point in $\mathcal{D}$). With systematic sampling, we bound the gap between i and j by $\lfloor 1/\rho \rfloor - 1$, i.e., $|i - j| \leq \lfloor 1/\rho \rfloor - 1$. According to the pigeonhole principle, no other sampling strategy (including random sampling) can achieve a smaller bound on this gap. We thus expect the $\mathcal{D}_\mathcal{S}$ produced by SP to be a strong baseline approximation of $\mathcal{D}$.

While SP is efficient, it only considers the point order in the mapped space, not the data point locations. The sampled points may then be far away from the points they represent, especially in sparse regions, cf. p_8 in Figure 4.5((a)).

Clustering. To preserve data distribution patterns, the *clustering* method (CL) clusters $\mathcal{D}$ in the original space into C (a system parameter) clusters and uses the set of cluster centroids as $\mathcal{D}_\mathcal{S}$. In this work, we use the k-means algorithm due to its simplicity, although other algorithms may also apply. Figure 4.5((b)) shows an example. After clustering the 16 points into $C = 4$ clusters, we compute four centroids p'_2, p'_6, p'_8, and p'_{14} to obtain $\mathcal{D}_\mathcal{S}$. Note that these centroids are closer to the points that they represent than those chosen by SP.

We note that the cluster centroids may not be part of $\mathcal{D}$. However, this does not impact the mapping from a data point's coordinates to its search key, which is either independent of the dataset (e.g., ZM [103] using ZCs), or is computed using $\mathcal{D}$ (e.g., the ML-Index [23] using iDistance [44]). MR and RL, to be presented shortly, also use non-subsets of $\mathcal{D}$.

However, clustering with k-means is expensive when C is large, taking $O(C \cdot |\mathcal{D}| \cdot d \cdot i)$ time for a straightforward implementation with i iterations. Further, it may create unbalanced clusters. For example, in Figure 4.5((b)), cluster $\mathbb{P}_2$ has only two points, while the other clusters have four or five points.

Model Reuse. *Model reuse* (MR) was originally proposed for one-dimensional data, which will be detailed in Chapter 5. It generates synthetic datasets with different distributions (e.g., Gaussian) and pre-trains index models on them. The dataset with the CDF that is the most similar to that of $\mathcal{D}$ (by $dist(\mathcal{D}_\mathcal{S}, \mathcal{D})$) is used as $\mathcal{D}_\mathcal{S}$, and its pre-trained model is used to index $\mathcal{D}$. MR first generates CDFs that together (heuristically) cover the CDF space, such that the distance between the CDF of any input dataset and that of at least one generated CDF is approximately bounded by a pre-defined threshold $\epsilon \in (0, 1]$. MR then generates synthetic datasets following the CDFs. By following this idea, we first generate CDFs and then corresponding synthetic datasets in the mapped space of a base index.

Figure 4.5((c)) shows an example of a multi-level index (RSMI) built by MR. There are three synthetic datasets $\mathcal{D}_{\mathcal{S}1}$, $\mathcal{D}_{\mathcal{S}2}$, and $\mathcal{D}_{\mathcal{S}3}$ (plotted in the original space for ease of

Algorithm 2: get_RS

Input: $\mathcal{D}$, β, $\{(l_1, u_1), (l_2, u_2), \ldots, (l_d, u_d)\}$
Output: Representative set $\mathcal{D}_S$

```
if |D| ≤ β then
    D_S ← {median point in D};
else
    for i = 0; i < 2^d; i++ do
        {(l_1^i, u_1^i), ..., (l_d^i, u_d^i)} ← compute bounds for partition i;
        D_i ← get points in the i-th partition;
        D_S ← D_S ∪ get_RS(D_i, β, {(l_1^i, u_1^i), ..., (l_d^i, u_d^i)});
return D_S;
```

observation), with pre-trained models $\mathcal{M}_{\mathcal{D}_{S1}}$, $\mathcal{M}_{\mathcal{D}_{S2}}$, and $\mathcal{M}_{\mathcal{D}_{S3}}$, respectively. Given a dataset $\mathcal{D}_{0,0}$ for indexing, we compare it with all three synthetic datasets and find $\mathcal{D}_{\mathcal{S}1}$ to be the most similar. We thus use $\mathcal{M}_{\mathcal{D}_{S1}}$ to index $\mathcal{D}_{0,0}$. This model predicts $\mathcal{D}_{0,0}$ into two partitions $\mathcal{D}_{1,0}$ and $\mathcal{D}_{1,1}$, which are the most similar to $\mathcal{D}_{\mathcal{S}1}$ and $\mathcal{D}_{\mathcal{S}2}$ and hence are indexed by $\mathcal{M}_{\mathcal{D}_{S1}}$ and $\mathcal{M}_{\mathcal{D}_{S2}}$, respectively. The process continues until no more partitioning is required by the base index.

4.4.2 Proposed Methods

To address the limitations related to index building efficiency (CL) or query efficiency (SP and MR) of the adapted methods, we propose two additional methods.

Representative Set. The first method, *representative set* (RS), recursively partitions the d-dimensional original data space into 2^d equi-sized partitions (e.g., four quadrants in a 2-dimensional space, as in quadtree partitioning). The process continues until every partition has no more than β points, where β is a system parameter. For each partition, the median point in the mapped one-dimensional space is added to $\mathcal{D}_{\mathcal{S}}$.

We summarize RS in Algorithm 2, where the lower and the upper bounds of the data space are denoted by $\{(l_1, u_1), (l_2, u_2), \ldots, (l_d, u_d)\}$. The bounds of the 2^d partitions in the recursive function calls are combinations of $l_1, l_2, \ldots, l_d$, $u_1, u_2, \ldots, u_d$, and $\frac{l_1+u_1}{2}, \frac{l_2+u_2}{2}, \ldots, \frac{l_d+u_d}{2}$. Points in each $\mathcal{D}_i$ can be computed with a scan over $\mathcal{D}$.

Figure 4.5((d)) gives an example, where $d = 2$ and $\beta = 4$, and $\mathcal{D}_{i,j}$ denotes partition j at partition level i. At level 1, $\mathcal{D}_{1,2}$ and $\mathcal{D}_{1,4}$ do not exceed 4 points each and do not require further partitioning. Partitions $\mathcal{D}_{1,1}$ and $\mathcal{D}_{1,3}$ are partitioned for one more level.

Finally, we have 9 non-empty partitions (solid green rectangles). Their median points form $\mathcal{D}_S = \{p_1, p_2, p_3, p_4, p_6, p_8, p_{10}, p_{12}, p_{14}\}$.

RS uses partitions of the original space and ranks in the mapped space for sampling in order to better approximate the distribution patterns of $\mathcal{D}$ in both the original and mapped spaces as it carefully selects middle points from equal-sized partitions. Therefore, it preserves the pattern of distribution of the original dataset such that enhances the efficiency of queries and decreases the cost of index construction. RS is expected to achieve high query performance, to be confirmed experimentally.

Reinforcement Learning. The *reinforcement learning* (RL)-based method aims to learn a set $\mathcal{D}_\mathcal{S}$ of up to η^d points that best approximate $\mathcal{D}$, using reinforcement learning. Here, η is a system parameter. RL partitions the data space with an η^d grid, fills every cell with a point initially, and continues to remove points from (or add points back into) the cells (Figure 4.5((e))). It monitors $dist(\mathcal{D}_\mathcal{S}, \mathcal{D})$ and terminates when $dist(\mathcal{D}_\mathcal{S}, \mathcal{D})$ stops improving the similarity between $\mathcal{D}_\mathcal{S}$ and $\mathcal{D}$.

Unlike traditional methods that employ static rules for data sampling, RL starts dynamically adding or removing points, thereby changing the CDF of $\mathcal{D}_\mathcal{S}$ and converging to an optimal. This adaptability also enhances query performance, as the RL model focuses on minimizing the distance between the sampled subset and the original data, leading to a more accurate and efficient querying process.

Reinforcement learning formulation. There are 2^{η^d} point combinations for forming different $\mathcal{D}_\mathcal{S}$. To approach the optimal $\mathcal{D}_\mathcal{S}$, we reduce the search cost via reinforcement learning and formulate the search as an MDP:

1. State space $\mathcal{S}$, where a state $s_t \in \mathcal{S}$ at time step t is a vector $(s_t[1], s_t[2], \ldots, s_t[\eta^d])$, where $s_t[i]$ is a binary variable that indicates whether there is a point in cell i. The cells are ordered by their ranks in the mapped space of the base index. The initial state s_0 has value 1 in every cell, i.e., $\mathcal{D}_\mathcal{S}$ starts with a uniform distribution.

2. Action space $\mathcal{A}$, where an action $a \in \mathcal{A}$ is to add (or remove) a point to (from) a cell of the partitioned grid.

3. Reward function $\mathcal{R}$: $\mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow \mathcal{R}$, which is calculated as the reduction in $dist(\mathcal{D}_{\mathcal{S}}, \mathcal{D})$ caused by an action $a_t \in \mathcal{A}$ on state $s_t \in \mathcal{S}$, i.e., $\mathcal{R}(s_t, a_t, s_{t+1}) = dist(\mathcal{D}_{\mathcal{S}t}, \mathcal{D}) - dist(\mathcal{D}_{\mathcal{S}t+1}, \mathcal{D})$.

4. State transition function $\mathcal{P}$: $\mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow [0, 1]$, which describes a probability distribution of the next state s_{t+1} given state s_t at time step t and action a_t.

5. Discount factor $\gamma \in [0, 1]$, which discounts the reward accumulated further into the future ($\gamma = 0.9$ in experiments).

We use a Deep Q-network (DQN) [67] to learn the optimal policy π: $\mathcal{A} \times \mathcal{S} \rightarrow [0, 1]$ that maximizes the reward. At step t, we choose cell i with DQN, update $s_t[i]$ to $s_{t+1}[i] = 1 - s_t[i]$ with probability ζ ($\zeta = 0.8$ in experiments), and compute the reward. The DQN is trained by recent state transition and reward records in memory after every five steps. When the learning stops, the final state yields the $\mathcal{D}_{\mathcal{S}}$ returned by RL.

4.5 Cost Analysis

We analyze the build, query, and update costs when using ELSI and benchmark these against those when not using ELSI.

4.5.1 Cost Formulation

For simplicity, we consider building and querying a single index model $\mathcal{M}$ on a dataset $\mathcal{D}$. We decompose the build and query costs to facilitate the performance comparison.

Build cost decomposition. The build cost $cost_b$ of a map-and-sort-based index can be decomposed into three terms: (i) data preparation cost $cost_{dp}$, (ii) model training cost $cost_{tr}$, and (iii) extra costs $cost_{ex}$ introduced by ELSI where $cost_b = cost_{dp} + cost_{tr} + cost_{ex}$. The build costs are:

1. $cost_{dp} = O(nd + n \log n)$: All n points in $\mathcal{D}$ are mapped to a one-dimensional space, which typically takes $O(nd)$ time to scan all points and dimensions. The points are then sorted in the mapped space, taking $O(n \log n)$ time.

2. $cost_{tr} = \mathbb{T}(n) + \mathbb{M}(n)$: All n points in $\mathcal{D}$ are used to train an index model. The costs depend on the dataset size, the model structure (e.g., the numbers of layers), and the number of training iterations. For ease of comparison, we focus on the impact of the dataset size and denote the two costs by $\mathbb{T}(n)$ for training and $\mathbb{M}(n)$ for model invocation.

Query cost decomposition. We focus on point queries and consider a predict-and-scan process. The query cost $cost_q$ has two terms: (i) prediction cost $cost_{pr}$ and (ii) scan cost $cost_{sc}$ where $cost_q = cost_{pr} + cost_{sc}$. The query costs are:

1. $cost_{pr} = \mathbb{M}(1)$: The index model is invoked once with the (mapped) query point. We use $\mathbb{M}(1)$ to denote this cost.

2. $cost_{sc} = O(min_{err} + max_{err})$: A scan is run over addresses adjacent to the predicted one, bounded by min_{err} and max_{err}.

4.5.2 Build Cost

To build an index with ELSI, the data preparation cost remains $cost_{dp} = O(nd+n \log n)$, while the model training cost now is $cost_{tr} = \mathbb{T}(|\mathcal{D}_{\mathcal{S}}|) + \mathbb{M}(n)$, since training over $\mathcal{D}_{\mathcal{S}}$. The extra costs $cost_{ex}$ incurred by ELSI include $\mathbb{M}(1)+O(n)$ time to invoke the method scorer to select build method and additional method-specific costs. Next, we consider $\mathbb{T}(|\mathcal{D}_{\mathcal{S}}|)$ and the extra costs of each method. The overall costs and empirical results of the methods are summarized in Table 4.1.

1. **SP** samples $\rho \cdot n$ points. Its model training cost is $cost_{tr} = \mathbb{T}(\rho \cdot n) + \mathbb{M}(n)$, and it incurs $O(\rho \cdot n)$ extra time to sample the points.

2. **CL** uses C cluster centroids for model training. Its model training cost is $cost_{tr} = \mathbb{T}(C) + \mathbb{M}(n)$. Clustering with k-means adds $O(C \cdot n \cdot d \cdot i)$ time to $cost_{ex}$, using a straightforward implementation with i iterations. The centroids also need to be mapped and sorted, but they do not impact $cost_{dp}$ in big-O terms, as $C \ll n$.

3. **MR** uses pre-trained models and does not run online training, i.e., $cost_{tr} = \mathbb{M}(n)$. It computes the similarity between $\mathcal{D}$ and n_{mr} synthetic sets, each of size $n_{\mathcal{S}}$, taking $O(n_{mr} n_{\mathcal{S}} \log n)$ extra time.

4. **RS** stops partitioning when a partition reaches β points and uses one point per partition for training. Its training cost is $cost_{tr} = \mathbb{T}(n/\beta) + \mathbb{M}(n)$. The partitioning tree has depth $O(\log_{2^d}(n/\beta))$ on average and takes $O(n \log_{2^d}(n/\beta))$ time to compute all the points.

5. **RL** approximates the original dataset with a synthetic set of size η^d, i.e., $cost_{tr} = \mathbb{T}(\eta^d) + \mathbb{M}(n)$. Generating this set with reinforcement learning adds extra costs for state and action training, reward calculation, and cell selection via DQN. Each step invokes the DQN network, which takes $\mathbb{M}(1)$ cost, and calculates the reward (i.e., $dist(\mathcal{D}_S, \mathcal{D})$), which takes $O(\eta^d \log n)$ time. The DQN is trained once in every five steps, performing learning on α past state transition records and corresponding rewards. With e steps, these combine to contribute $\mathbb{M}(e) + O(e\eta^d \log n) + \mathbb{T}(\alpha)$ extra time.

4.5.3 Query Cost

ELSI does not change the structures of index models. Its prediction cost is still $cost_{pr} = \mathbb{M}(1)$. It may incur extra scan costs, since model $\mathcal{M}_{\mathcal{D}_S}$ that is built on $\mathcal{D}_S$ may have different error bounds from those of model $\mathcal{M}$ that is built on $\mathcal{D}$. Let min^S_{err} and max^S_{err} be the error bounds of $\mathcal{M}_{\mathcal{D}_S}$, and let $\Delta err = min^S_{err} + max^S_{err} - min_{err} - max_{err}$. Then, the scan cost of ELSI is $cost_{sc} = O(min_{err} + max_{err} + \Delta err)$. Just like there are no non-trivial bounds for min_{err} and max_{err}, there is no non-trivial bound for Δerr. Intuitively, Δerr is expected to be inversely proportional to $\rho \cdot n$ for SP, C for CL, $1 - \epsilon$ for MR, $1/\beta$ for RS, and η for RL. For future work, we plan to derive a theoretical bound for Δerr.

4.5.4 Update Cost

Update processing with our default procedures takes $O(\log n_u)$ time assuming the list of inserted and deleted points is indexed by the point IDs in a binary tree given n_u points in the tree. The rebuild predictor is triggered to make a prediction after every f_u updates adding $\mathbb{M}(1)/f_u$ time.

4.6 Experiments

We run experiments on a computer running 64-bit Ubuntu 20.04 with a 3.60 GHz Intel i9 CPU, an RTX 2080 Ti GPU, 64 GB RAM, and a 500 GB solid-state drive. We use *PyTorch* 1.4 [83] and its C++ APIs to implement the learned indices based on the GPU. The traditional indices are implemented using C/C++ based on the CPU.

4.6.1 Experimental Setting Details

We apply ELSI to four learned spatial indices: ZM [103], ML-Index [23], RSMI [84], and LISA [58]. We report the performance of the last three and denote them as "**ML-F**", "**RSMI-F**", and "**LISA-F**", respectively ('-F' indicates that the ELSI framework is used). Since ZM has been shown in Chapter 3 to be outperformed by RSMI, we only consider it in Section 4.6.4 to investigate the performance of ELSI.

We note that not all index building methods are applicable to all learned spatial indices. For example, CL and RL do not apply to LISA that relies on $\mathcal{D}$ to construct a grid, while CL and RL may generate new points not in $\mathcal{D}$. ELSI offers an API to configure the index building methods used.

Competitors. We benchmark the learned spatial indices and compare them using four traditional indices:

1. **Grid** [72] partitions the data space with a regular grid, assigns data points to the cells they fall into and stores the data points in cell-wise fashion. We use a $\sqrt{n/B} \times \sqrt{n/B}$ grid, i.e., each cell (block) has an average of B points.
2. **KDB** [87] implements a kd-tree [13] with a B-tree structure to support block storage.
3. **HRR** [85] is an R-tree bulk-loaded using a *rank space* technique and a HC for the ordering. This index offers the state-of-the-art range query performance;
4. **RR*** [10] is a version of the R*-tree (i.e., the *revised R*-tree*) with improved query performance.

Datasets. We use four real datasets, **OSM1**, **OSM2**, **TPC-H**, and **NYC**, and two synthetic datasets, **Uniform** and **Skewed**. **OSM1** has about 100 million points (2.2 GB) in North America. **OSM2** has over 180 million points (4.4 GB) in South America. Both sets are extracted from OpenStreetMap [76]. **TPC-H** [99] contains the `quantity` and `shipdate` columns of 120 million records (5.0 GB) from the `lineitem` table of the TPC-H benchmark. **NYC** [98] contains the pickup points of 143 million yellow taxi transactions (5.6 GB) from the New York City.Each synthetic dataset has 128 million points (2.5 GB) in a unit square. **Uniform** has a uniform distribution, while **Skewed** is obtained by replacing the y-coordinates of the points in **Uniform** by y^s ($s = 4$), following **HRR** [85].

4.6.2 Implementation Details

Implementation of the Spatial Indices. For Grid, HRR, KDB, RR*, and RSMI, we use the implementation from Chapter 3. We implement ML and LISA to enable a consistent comparison with the other indices. We use an FFN for all the prediction models, ReLU activation function for the hidden layer and minimize the L_2 loss, and a learning rate of 0.01 and 500 epochs with the Adam optimizer, respectively. For the method scorer, we set $w_Q = 1.0$ for simplicity, and we study the impact of λ. Note that the use of FFNs instead of the piecewise linear functions used in LISA breaks the monotonicity of its shard prediction functions, which impacts the accuracy of range queries. We did this for consistency and simplicity of implementation.

We use a block size of $B = 100$ for data storage. We perform all experiments in memory for ease of comparison (it is straightforward to place the blocks in external memory).

Implementation of ELSI. To train the method scorer, we generate datasets with cardinalities ranging from 10^l to 10^u, where $l = 4$ and $u = 8$. We vary $dist(\mathcal{D}_\mathcal{U}, \mathcal{D})$ from 0.0 to 0.9 with a step size of 0.1 to generate datasets with different distributions. When integrated with a base index, we use every applicable method in the method pool to build an index for each generated dataset and run point queries. We record the speedups of index building and querying relative to those of the original methods of the base index. These form the ground truth for training the method scorer.

To train the rebuild predictor, we build learned indices with datasets of size 10^u and different distributions, as done above. We insert random points (or delete existing points) and run point queries every time $2^i\% \cdot n$ ($i \in \mathbb{N}$) point updates (inserts or deletes), on indices with and without rebuilds. We record the statistics to compose model training samples, and we set the corresponding model output heuristically: 1 (to rebuild) when the query time without rebuilds exceeds that with rebuilds by 10%, and 0, otherwise.

System preparation costs. ELSI preparation includes the training of method scorer and rebuild predictor, which require the generation of training sets and the training of new models from scratch. ELSI preparation is an off-line and one-off task, and once learned, the ELSI method selector and rebuild predictor can be reused for different datasets. As u ($u \in \mathbb{N}^+$) decreases from 8 to 4, the ELSI preparation times will drop from 10.5 to 1.9, 0.2, 0.03, and 0.003 hours, respectively.

4.6.3 Effectiveness of the Method Selector

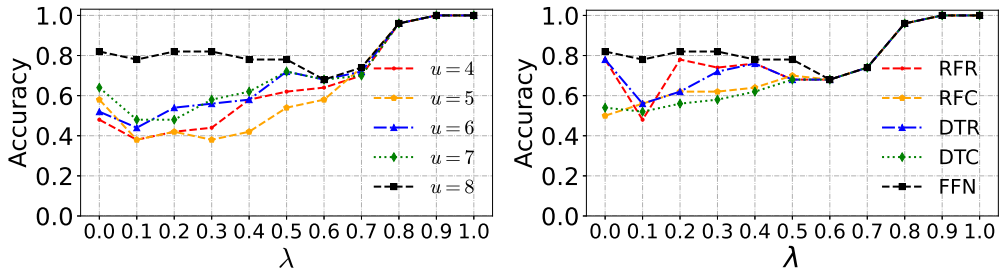
As mentioned in method scorer training (Section 4.6.2), there are 300 generated synthetic datasets, which are composed of the combinations of five different cardinalities, six build methods, and ten different similarities with uniform distribution. We then use accuracy to measure the method selector where the accuracy represents the ratio that the method selector selects the same method as expected.

To test the accuracy of the method selector under different values of u, we vary u from 4 to 8 and report the accuracy in Figure 4.6((a)). As expected, the method selector has the highest accuracy when $u = 8$, which means that the long preparation time (10.5 hours) has paid off. Meanwhile, we see that, if index build time is of priority, even using a small value of $u = 4$ can offer a high accuracy, as it is easier to predict which method to use in this case (e.g., MR).

To further highlight the effectiveness our FFN-based method selector, we compare it with method selectors using Random Forests (RF) and Decision Trees (DT), including two variants each, i.e., regression-based (R) and classification-based (C). We compare with four models: RFR, RFC, DTR, and DTC.

Figure 4.6((b)) shows that using an FFN as the method selector has a consistently higher (or the same) accuracy than that using a random forest or a decision tree, especially

when $\lambda < 0.6$. This highlights the effectiveness of our FFN-based method selector. When $\lambda \leq 0.6$ (prioritising query times), the selector accuracy is in general lower than that when $\lambda \geq 0.8$ (prioritizing index build times). This is because optimizing the query times of the learned indices is more difficult than optimizing the index build times, since the query times of the indices built by different methods are much closer than the build times of the different methods. FFN manages to achieve an accuracy around 0.8, while the other methods have accuracy at as low as 0.5. Further, when $\lambda \approx 0.6$, the accuracy is the lowest for FFN, because the build times and the query times have similar weights, which makes it difficult to learn and select the optimal index building method. When $\lambda = 0.1$, there is a drop in the accuracy of both RFR and DTR. They have been confused by the high query efficiency of the indices built by OG, and have often selected OG as the building method.



(A) Accuracy vs. u. (B) Accuracy vs. selector model.

FIGURE 4.6: Accuracy of method selector vs. λ

4.6.4 Effectiveness of Index Building Methods

We study the relative performance of the index building methods SP, CL, MR, RS, and RL (cf. Section 4.4). We also show the performance of the base indices without ELSI, i.e., *original* (OG), and a random sampling method RSP [59].

First, we show the point query times and build times of the methods when varying method-specific parameters following SOSD [50]. Figure 4.7 presents the results on OSM1 for all four base indices. We make four observations:

1. The build times increase while the query times decrease for the different methods, with the increase of ρ in SP and RSP, C in CL, $1-\epsilon$ in MR, $1/\beta$ in RS, and η in RL ($\alpha = 10,000, e = 50,000$). RSP has higher query times than SP and few benefits

in build times, because RSP has larger CDF distances between $\mathcal{D}_S$ and $\mathcal{D}$ than SP does, impacting query efficiency on $\mathcal{D}$. In what follows, we only consider SP.

2. The proposed RS and RL methods have low query times, because of their high-quality training sets. RS achieves the lowest query time for ZM, while it is very close to CL for RSMI and SP for LISA. Meanwhile, its build times are much lower than those of CL that employs an expensive clustering process. RL performs close to RS but is inapplicable for LISA.

3. RS and RL have larger build times than SP and MR, due to their larger training set size and more complex learning process, respectively. SP simply samples ρn points for training, while MR just reuses a pre-trained model. Both methods are thus favored in build-cost sensitive settings.

   RS and RL have the advantage that tuning their parameters β and η leads to query times close to, or even below, those of OG, which is more difficult to achieve with SP and MR. The two methods are favored in query-cost sensitive settings.

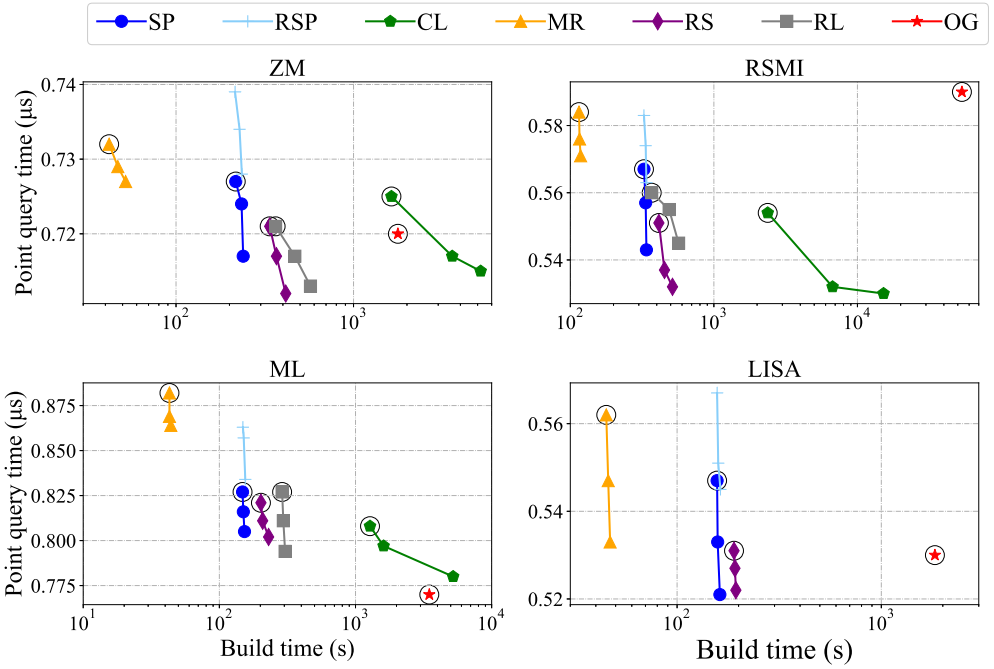


FIGURE 4.7: Comparison of different build methods on OSM1. In each sub-figure, as the query time decreases, ρ in SP and RSP increases from 0.0001 to 0.01, C in CL increases from 100 to 10,000, ϵ in MR decreases from 0.5 to 0.1 (if ϵ is too small, no pre-trained models may be reused), β in RS decreases from 10,000 to 100, and η in RL increases from 8 to 32.

4. The query times of the indices built by ELSI may be lower than those of OG. This is counterintuitive but can be explained as follows. OG uses the full datasets,

which may be too large and may contain noisy patterns (e.g., outliers) that confuse the index model. In comparison, the methods in ELSI learn from representative data samples, which are smaller and contain more consistent patterns, leading to indices with better query performance on average.

Results on the other datasets exhibit similar patterns and are omitted. Since the build times change much more rapidly (note the scales of the axes) and ELSI is proposed to reduce the index build times, we use the values that yield the best build times as default values, i.e., $\rho = 0.0001$, $C = 100$, $\epsilon = 0.5$, $\beta = 10,000$, and $\eta = 8$ (corresponding the points denoted by '$\odot$' in Figure 4.7).

Cost decomposition. Table 4.1 decomposes the index building and query costs on OSM1 with ZM. For index building, as all methods share the same map-and-sort data preparation process and have the same cost (i.e., $O(nd + n \log n)$ cost and 14 s in experiments), we omit it in Table 4.1. SP, CL, RS, and RL train index models using subsets of similar sizes and have similar training times. OG has the largest training time, while MR has the smallest training time, as it simply reuses pre-trained models. After adding in the extra costs, all methods except CL outperform OG by at least an order of magnitude, confirming the efficiency of ELSI in index building. Meanwhile, the model prediction errors $min_{err} + max_{err}$ (denoted by "|Error|") of the different methods are all at the same magnitude as OG, confirming the query efficiency of ELSI.

Table 4.1: Cost decomposition on OSM1

	Building cost (seconds)		|Error|
	Training	Extra ($+\mathbb{M}(1) + O(n)$)	($\times 10^5$)
SP	$\mathbb{T}(\rho n) + \mathbb{M}(n)$	$O(\rho n)$	6.4
	1	1	
CL	$\mathbb{T}(C) + \mathbb{M}(n)$	$O(Cndi)$	5.8
	1	213	
MR	$\mathbb{M}(n)$	$O(n_{mr} n_{\mathcal{S}} \log n)$	6.8
	1	0.1	
RS	$\mathbb{T}(n/\beta) + \mathbb{M}(n)$	$O(n \log_{2^d}(n/\beta))$	6.0
	1	20	
RL	$\mathbb{T}(\eta^d) + \mathbb{M}(n)$	$\mathbb{M}(e) + O(e\eta^d \log n + \mathbb{T}(\alpha))$	5.2
	1	18	
OG	$\mathbb{T}(n) + \mathbb{M}(n)$	n/a	6.3
	360	0	

4.6.5 Ablation Study

We conduct an ablation study comparing ELSI with a variant of ELSI that uses an index building method selector, denoted by "**Rand**", that selects each index building method with an equal probability.

Performance comparison. Table 4.2 shows a comparison of the build and query times when building an index using ELSI, Rand, and each index building method, on OSM1 using $\lambda = 0.8$. As expected, Rand has worse build times than ELSI, as Rand risks selecting a slow method for some index models. Meanwhile, Rand does not offer better query times. This confirms the effectiveness of the ELSI design with a method pool and a learned method selector.

Table 4.2: Comparison of ELSI with a random method selector on OSM1

	Index	ELSI	Rand	SP	CL	MR	RS	RL	OG
Build time (s)	ZM	57	217	277	1,635	42	414	364	1,778
	RSMI	233	557	326	2,372	115	337	369	53,435
	ML	60	212	148	1,278	43	202	289	3,483
	LISA	42	175	157	NA	37	190	NA	1,830
Point query time (μs)	ZM	0.73	0.73	0.73	0.73	0.73	0.72	0.72	0.72
	RSMI	0.57	0.56	0.60	0.55	0.57	0.55	0.56	0.59
	ML	0.88	0.85	0.83	0.81	0.88	0.82	0.83	0.77
	LISA	0.56	0.55	0.55	NA	0.56	0.53	NA	0.53

4.6.6 Index Building Performance

We now compare the build times of the learned indices with and without using ELSI and the traditional indices.

Varying data distribution. Figure 4.8 shows that the traditional indices (i.e., Grid, KDB, HRR, and RR*) are faster to build than the learned indices (i.e., ML, LISA, and RSMI) without ELSI. Using ELSI, the build times of the learned indices (i.e., ML-F, LISA-F, and RSMI-F) are reduced to the level of the traditional indices. The performance gain is at least 25 times (217 s vs. 5,481 s for ML-F and ML on OSM2) and up to 229 times (233 s vs. 53,435 s for RSMI-F and RSMI on OSM1). LISA-F even outperforms all the traditional indices on OSM2, i.e., 74 s vs. 126 s (Grid), on TPC-H, i.e., 43 s vs. 66 s (KDB), and on NYC, i.e., 55 s vs 75 s (KDB), respectively.

On average, ELSI speeds up the building of the learned indices by 70 times. We notice that Grid is worse on NYC than on the other datasets. This is because Grid uses a two-level structure where every cell contains an array of MBRs each corresponding to a data block (to help query processing later on). Inserting points into this structure while minimizing the MBRs is more expensive when the data points are skewed (which is the case for NYC), since the data blocks in the denser cells require frequent splits.

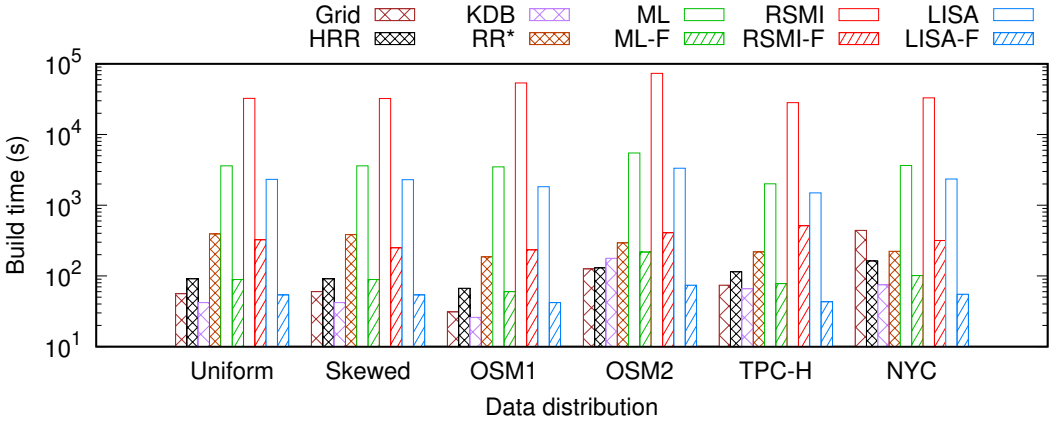


FIGURE 4.8: Build time vs. data distribution.

Varying λ. Figure 4.9 shows how the build times of the learned indices using ELSI change when varying λ. The traditional indices and the learned indices without ELSI are not impacted by λ. Their performance remain as reported in Figure 4.8. For ease of comparison, we include the results of the traditional index RR* and the learned index RSMI without using ELSI in Figure 4.9. As the relative performance of the different indices are similar across the datasets, we only show results on Skewed and OSM1. This also applies to the subsequent studies.

We see that the build times of the ELSI-based indices decrease as λ increases. Guided by Equation 4.2, the ELSI index building method selector selects more build-time efficient methods given a larger λ. MR is the most frequent choice when $\lambda \geq 0.8$ (as it simply reuses pre-trained models), which brings down the index build times to below those of RR* (except for RSMI-F on OSM1). When λ is small, the query-optimized methods RS, RL and OG are selected. While these incur higher build times, the resultant build times are still much closer to RR* than RSMI without ELSI. This again confirms the effectiveness of ELSI in reducing the index build times.

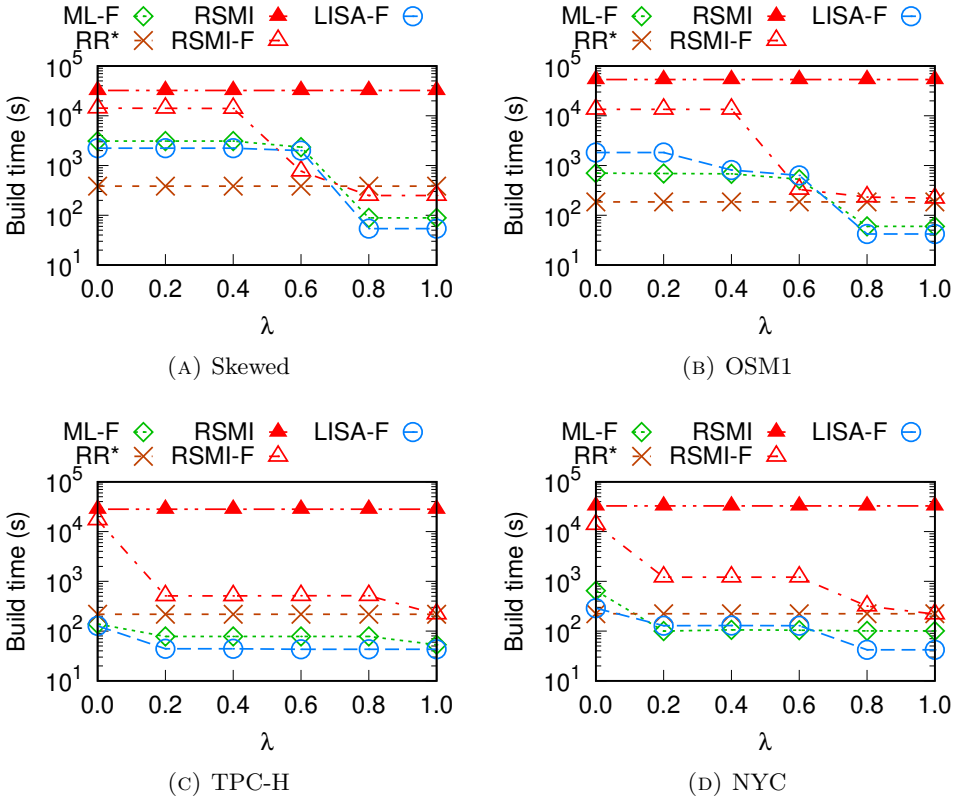


FIGURE 4.9: Build time vs. λ.

4.6.7 Query Performance

Next, we report point, range, and k nearest neighbor (kNN) query times of the different indices. Following the experimental setting in Chapter 3, we also execute the queries with 10 different random sequences in Figure 4.10, Figure 4.12, and Figure 4.14.

4.6.7.1 Point Queries

We execute point queries over every point in a dataset after indices are built and report the average query time per query.

Varying the data distribution. As Figure 4.10 shows, like what has been shown in Chapter 3, the learned indices outperform the traditional ones except on Uniform where Grid is the fastest. This observation is also consistent with previous results on learn spatial indices [58]. Using ELSI yields very similar query times to those of the learned indices when not using ELSI. The point query time increases by at most 14%, i.e., 0.882 μs (ML-F) vs. 0.773 μs (ML) on the OSM1 dataset. Recall that our default λ value

of 0.8 optimizes towards index build times. MR, which has high query times, is more likely to be chosen by ELSI. RSMI-F and LISA-F still obtain faster point query times than RSMI (on all real datasets) and LISA (on OSM2 and NYC), since real data points are skewed and noisy, and learning an index model on a large set of such data may be impacted by noisy patterns, as discussed earlier. On average, the point query times of the learned indices are not increased by ELSI.

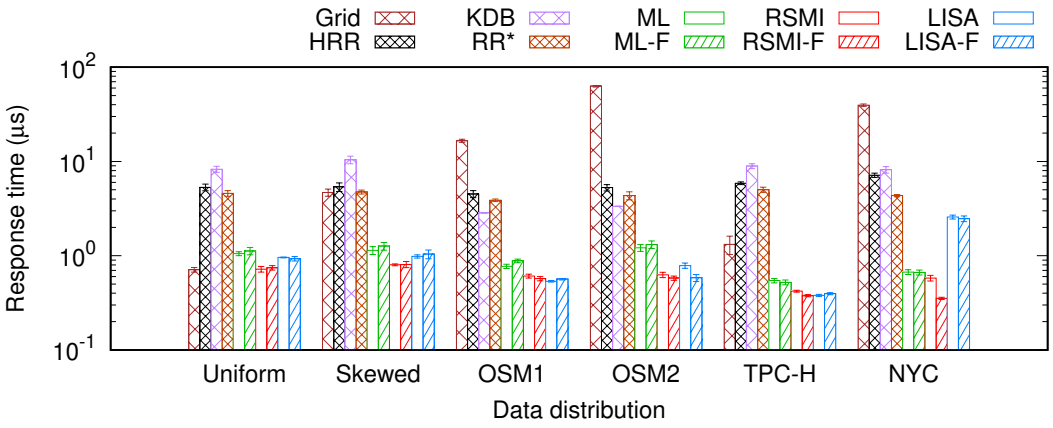


FIGURE 4.10: Point query time vs. data distribution.

Varying λ. Figure 4.11 shows that the point query times of the learned indices using ELSI increase slowly with λ. The maximum increase is observed on ML-F, i.e., from 0.741 μs to 0.883 μs on OSM1 for λ equal to 0 vs. 1. RSMI-F and LISA-F outperform both RSMI and RR* on OSM1. RSMI-F, LISA-F, and ML-F all outperform RSMI and RR* on TPC-H. This again confirms the effectiveness of ELSI in reducing index build times while retaining query efficiency, and it justifies using a relatively large λ value to achieve fast index building.

4.6.7.2 Window Queries

We report the average range query performance over 1,000 queries that are generated following the data distribution.

Varying data distribution. The learned indices with ELSI have very similar query times to those of the learned indices without ELSI when the query range size is 0.01% of the data space (Figure 4.12((a))). The worst case is observed on Skewed, where LISA-F is 1.35 times slower than LISA (2,976 μs vs. 2,197 μs), while LISA-F is 1.37 times faster than LISA on TPC-H (6,048 μs vs. 8,329 μs).

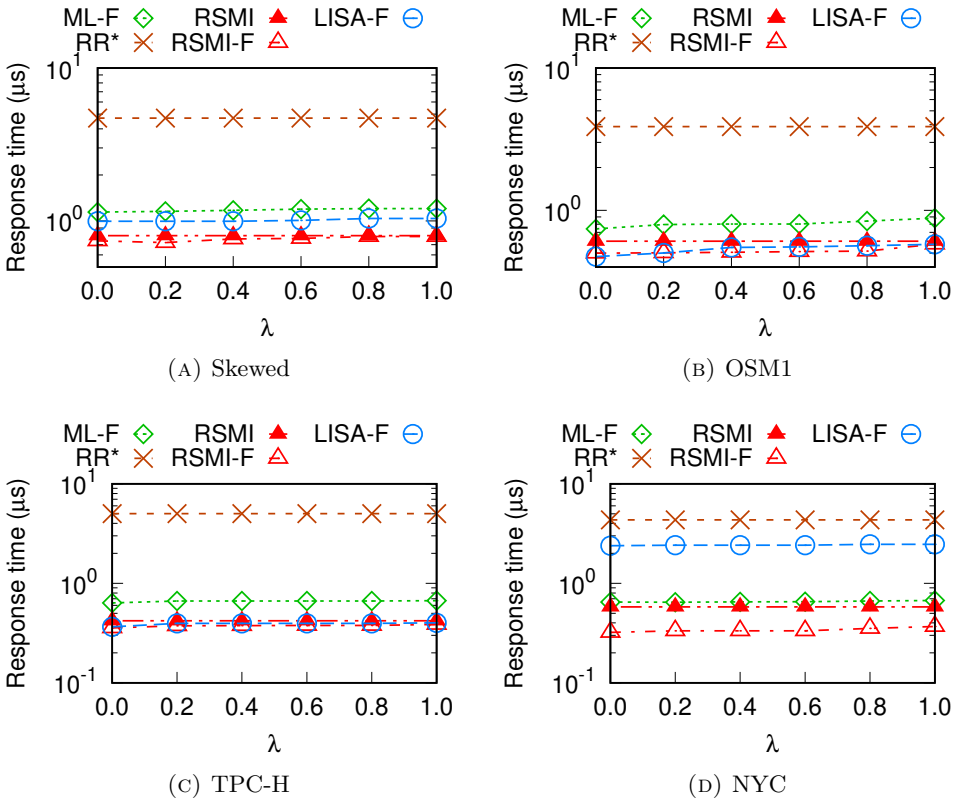
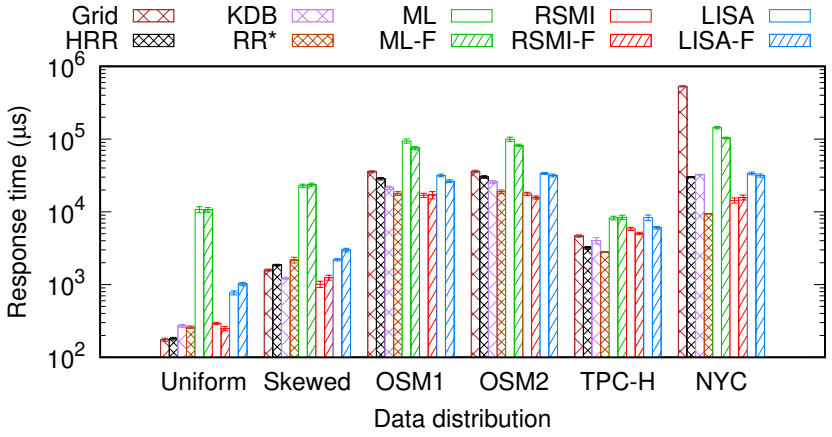


FIGURE 4.11: Point query time vs. λ.

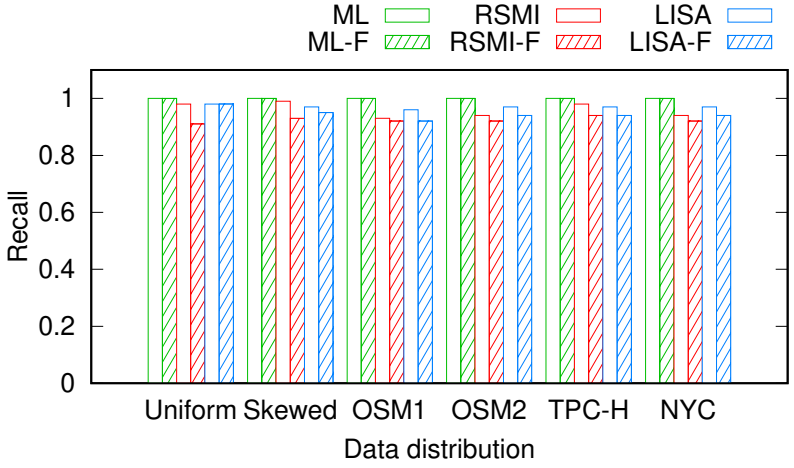
RSMI (by original design) and LISA (when using FFNs) return approximate range query results. Figure 4.12((b)) shows the impact of ELSI on the query *recall* (i.e., the ratio of ground truth points in the returned results). The recall of both RSMI and LISA drops with ELSI, which is expected. The recall of RSMI-F and LISA-F stays above 91% and 92%, respectively, again confirming the effectiveness of ELSI in retaining the query performance. By design, ML offers accurate results, which are not impacted by ELSI. For brevity, we omit the recall figures for the range query experiments below, since the recall gaps remain similar, and the recall of RSMI-F and LISA-F stays over 90%.

Varying λ. Figure 4.13((a)) shows the range query times when varying λ on OSM1. As observed for point queries, the range query times of the ELSI-based indices increase slowly with λ, confirming the robustness of ELSI to variations in λ for range queries. Results on the other datasets exhibit a similar pattern, and their figures are omitted.

Varying query range size. Figure 4.13((b)) further shows the impact of the query range size. The query times increase with the query range size (from 0.0006% to 0.16% of the data space size) for all indices as expected. Notably, the query times of the



(A) Query time



(B) Recall

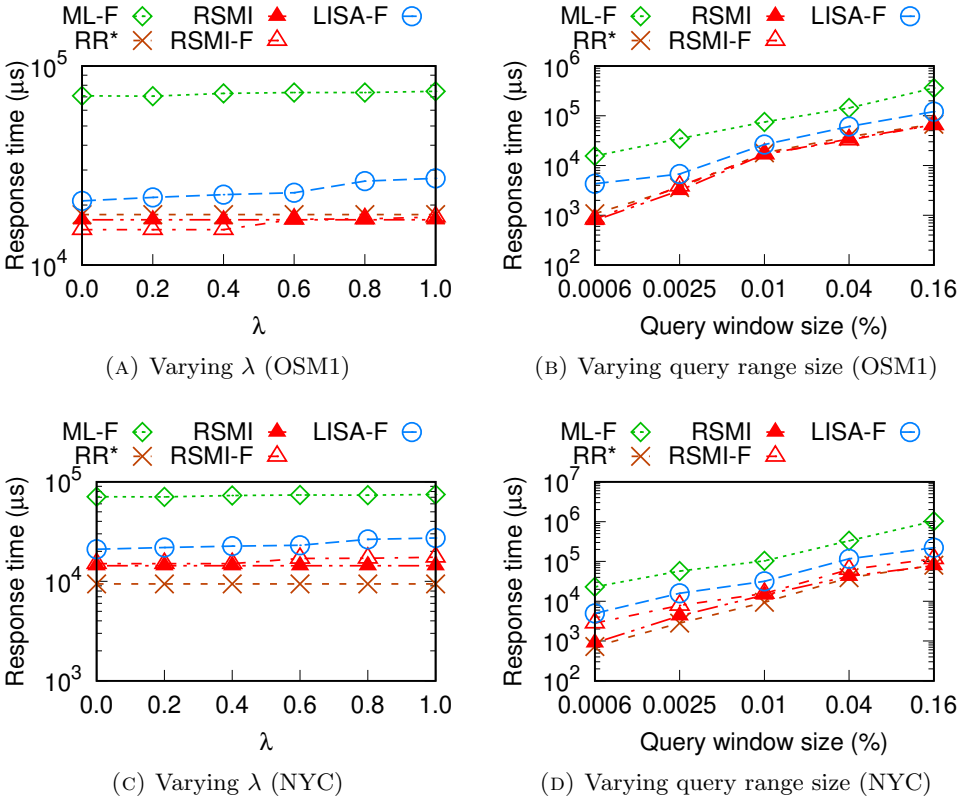
FIGURE 4.12: Range query time vs. data distribution.

ELSI-based indices do not grow faster than those of RR* and RSMI without ELSI. This indicates the robustness of the ELSI-based indices to variations in the query range size.

4.6.7.3 KNN Queries

We report average kNN query performance over 1,000 kNN queries that follow the data distribution of the datasets with $k = 25$. We omit results when varying λ and k because they resemble those for range queries.

Varying data distribution. Figure 4.14((a)) shows the kNN query times when $k = 25$. Like the observations on range queries, RSMI and RSMI-F are the fastest, except on Uniform and TPC-H. This is expected because the learned indices use range queries as the basis for kNN queries. Using ELSI retains the query performance, yielding an



(A) Varying λ (OSM1)

(B) Varying query range size (OSM1)

(C) Varying λ (NYC)

(D) Varying query range size (NYC)

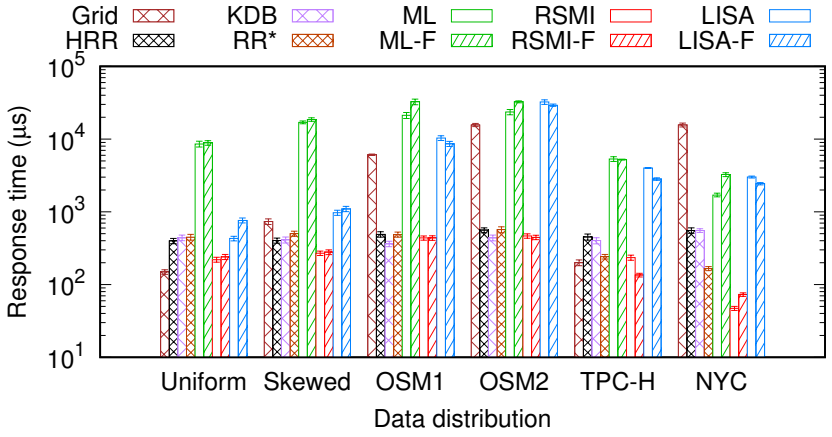
FIGURE 4.13: Range query time vs. λ and range size.

average increase in the query time of only 3%. In the worst case, ELSI increases the query time from 430 μs (LISA) to 736 μs (LISA-F) on Uniform, which is considered reasonable given the large dataset of 128 million points.

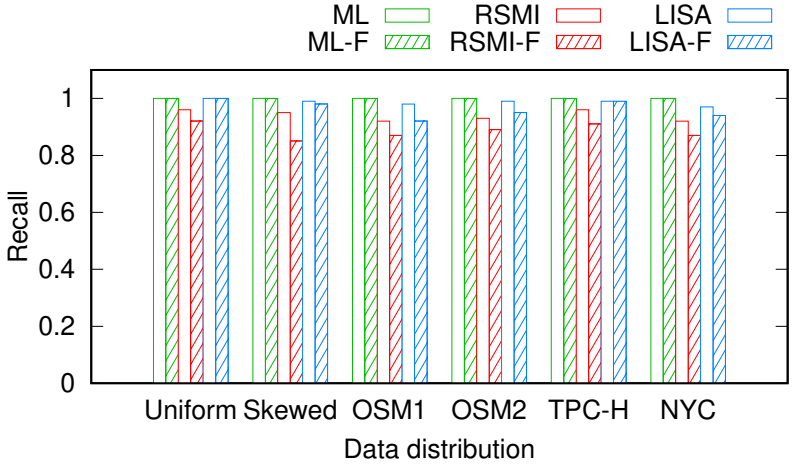
In terms of recall, Figure 4.14((b)) shows that ELSI causes a maximum drop of 10% (95% vs. 85% for RSMI and RSMI-F on Skewed). LISA-F drops at most 6% compared with LISA (98% vs. 92% on OSM1), while ML-F stays at 100%.

4.6.8 Update Performance

We examine the impact of updates and focus on insertions for succinctness. We use 10% of the points from OSM1 as the initial set to build an index and use data from Skewed for insertions. We run point queries for all points indexed and 1,000 range queries after the insertions (results of kNN queries are omitted for brevity). We report the average time for each query type.



(A) Query time

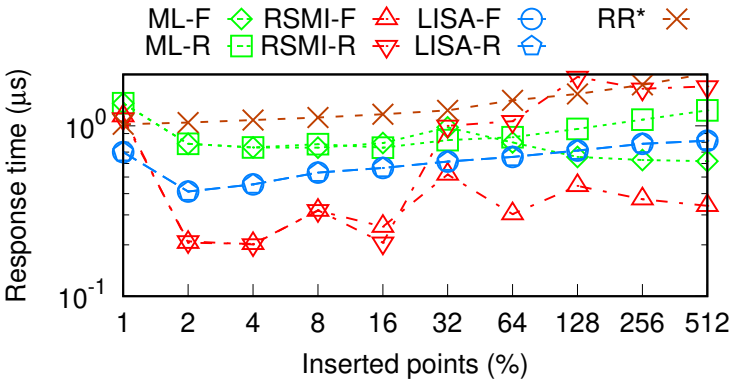


(B) Recall

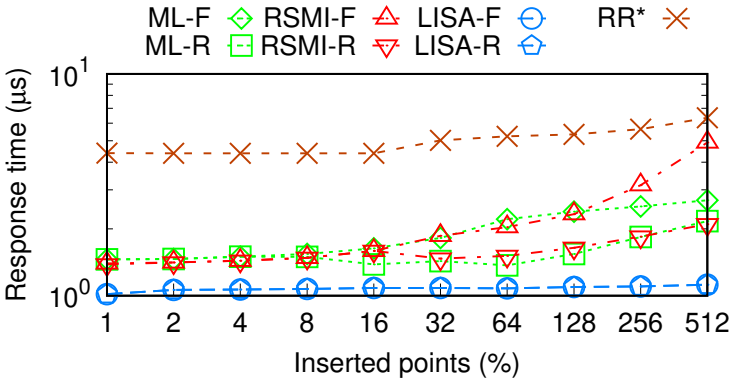
FIGURE 4.14: KNN query time vs. data distribution.

We show the results of the ELSI-based indices with and without global rebuilding with the update processor. The indices without rebuilds are denoted by ML-F, LISA-F, and RSMI-F; and those with rebuilds are denoted by ML-R, LISA-R, and RSMI-R. We include RR* as it is the traditional index that shows the overall best query performance.

Insertions. Figure 4.15((a)) shows the average insertion time. For RR*, the insertion time grows gradually as more points are inserted due to its self-balancing insertion procedure without rebuilds. For the learned indices, the average insertion times are relatively high for the first 1% of n insertions because most data pages are full after the learned indices are built, meaning that insertions cause the creation of relatively many pages (LISA and RSMI use built-in insertion procedures, and ML uses extra data pages to store points inserted into each index model). Subsequently, LISA-F and LISA-R have stable insertion times that increase gradually as for RR*. The rebuild predictor guides



(A) Insertion time vs. insertion ratio



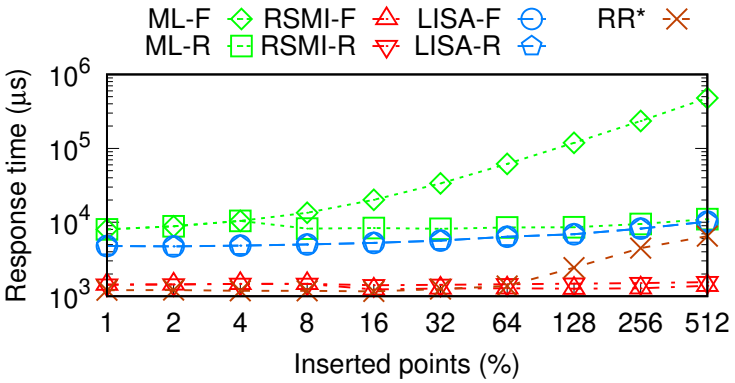
(B) Point query time vs. insertion ratio

FIGURE 4.15: Skewed data insertion.

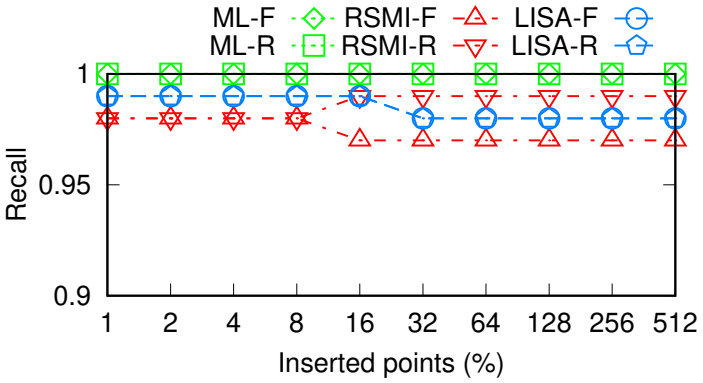
LISA-R not to rebuild as its point query times do not deteriorate (cf. Figure 4.15((b))). For ML-R and RSMI-R, global rebuilds are triggered after 8% of n insertions (and a rebuild takes up to 133 and 60 seconds, respectively), causing higher insertion times. We will see next that these rebuilds pay off by keeping the query times stable.

Point queries. Figure 4.15((b)) shows the point query times after insertions, which mostly increase as expected. ML-R and RSMI-R are exceptions. Their global rebuilds bring down the query times substantially, e.g., after 512% of n insertions, the query times of ML-R and RSMI-R are 19% and 47% lower than ML-F and RSMI-F, respectively.

Range queries. For range queries after insertions, the query times again increase, as shown in Figure 4.16((a)). Now RR* and RSMI have the lowest query times, which is consistent with observations from the earlier range query experiments. The global rebuilds keep the query times of ML-R lower than those of ML-F. However, RSMI-R is not always faster than RSMI-F. This is because the local index rebuilds of RSMI-F may lead to less accurate structures with fewer points to be scanned for a range query.



(A) Query time vs. insertion ratio



(B) Query recall vs. insertion ratio

FIGURE 4.16: Range queries with skewed data insertion.

Figure 4.16((b)) shows the RSMI-R global rebuilds help retain the recall above 97%, while the local rebuilds (RSMI-F) only keep the recall above 90%, further justifying global rebuilds.

4.6.9 Result Summary

We conducted experiments to show (1) the effectiveness of ELSI to select the most appropriate index building method and to determine the right timing for index rebuilds, (2) the efficiency of index building with ELSI, and (3) the efficiency of query processing using the indices built by ELSI. The results show that: (1) ELSI can select the most appropriate index building method with an accuracy over 80%, and it also predicts the right time for index rebuilds, which reduces the query times after data insertions by more than 47%. (2) For index building, using ELSI (i.e., ML-F, RSMI-F, and LISA-F) is one to two orders of magnitude faster than building the learned indices directly (i.e., ML, RSMI, and LISA). This brings down the index building time to the level of those

of the traditional indices (Grid, KDB, HRR, and FR*). (3) For query processing, the learned indices built with ELSI share similar times with the respective learned indices built without ELSI, e.g., the kNN query times differ by just 3% on average.

4.7 Summary

We proposed a system named ELSI that enables efficient building and rebuilds of learned spatial indices while retaining their high query efficiency. ELSI is versatile, and it can support a learned spatial indices as long as it follows a map-and-sort based data indexing paradigm and a predict-and-scan based query paradigm. ELSI has a suite of methods to construct small and representative training datasets for the index learning and rebuilds. It can adaptively choose the method that produces a learned index with a high query efficiency for a given dataset. Experiments on real datasets with over 100 million points show that ELSI reduces the build time of four different learned spatial indices consistently and by up to 229 times, while it also predicts the right time to rebuild the indices, which helps reduce more than 47% of the query time.

Chapter 5

Efficiently Learning Indices via Model Reuse and Fine-tuning

Chapter 4 described a framework to speed up learned spatial index building. In the framework, there are six candidate methods that can be adaptively chosen for index building, including a method named MR. MR replaces an online index learning process with offline index model pre-training plus online matching between a target dataset and the pre-trained index models. This method avoids online index learning from scratch, which can significantly reduce the index build times. In this chapter, we will detail this method and further enable *fine-tuning* after MR to optimize query performance. The MR method is generic to learned indices and not limited to spatial data. Without loss of generality, we will present the idea of the method assuming one-dimensional data, and we will detail its extension to spatial data in Section 5.3.

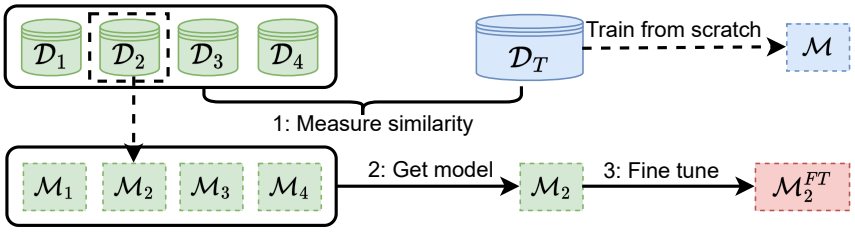
5.1 Overview

While learned indices have efficient query procedures, they can be prone to slow building, since machine learning models are expensive to train. The bottleneck of index building is that every data record within the training set will be full-scanned in each epoch and

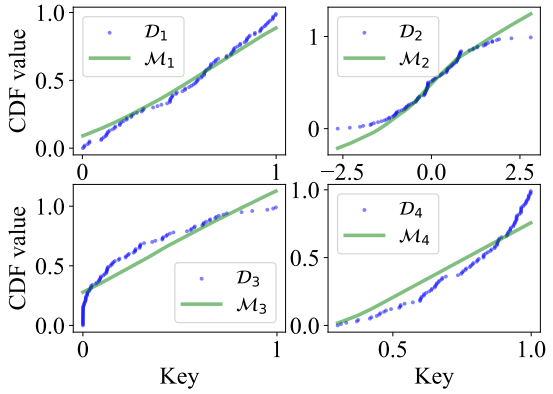
The work reported in this chapter has been published in the following paper: Guanli Liu, Jianzhong Qi, Lars Kulik, Kazuya Soga, Renata Borovica-Gajic, Benjamin I. P. Rubinstein. *Efficient Index Learning via Model Reuse and Fine-tuning*, International Workshop on Databases and Machine Learning (**DBML**) 2023, held together with ICDE 2023.

there is at least one epoch during learning. Even with simple models such as linear regression, a learned index such as the RMI [53] is more than an order of magnitude slower to build than a B-tree [61]. Meanwhile, techniques that learned indices in a single pass such as RS [49] and PGM-index [30] still require a nested loop to build, e.g., to check the validity of given error bounds for every key. They also tend to produce sub-optimal indices of large sizes and lower query efficiency than RMI[1].

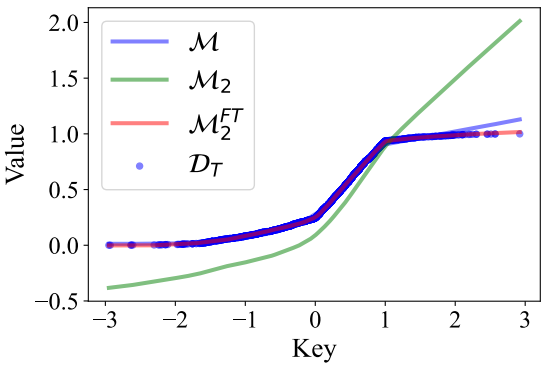
To address these issues, we aim to save index build times by leveraging pre-trained models to avoid training from scratch. This solution is inspired by *domain adaptation* [11]. Given a model $\mathcal{M}_S$ trained on a known (source) dataset $\mathcal{D}_S$, domain adaptation reuses $\mathcal{M}_S$ for a new (target) dataset $\mathcal{D}_T$ by fine-tuning $\mathcal{M}_S$ over $\mathcal{D}_T$ (see Figure 5.1(a)). This avoids the expensive training of a new model on $\mathcal{D}_T$ from scratch.



(A) Pre-training: Model $\mathcal{M}_1$ to $\mathcal{M}_4$ are pre-trained on known datasets $\mathcal{D}_1$ to $\mathcal{D}_4$, respectively. Existing solutions train a new model $\mathcal{M}$ on $\mathcal{D}_T$ from scratch. We fine-tune a pre-trained model $\mathcal{M}_2$ as $\mathcal{M}_2^{FT}$ to index $\mathcal{D}_T$.



(B) The CDFs of four known datasets and their trained models.



(C) The CDF of $\mathcal{D}_T$ and models $\mathcal{M}$, $\mathcal{M}_2$, and $\mathcal{M}_2^{FT}$.

FIGURE 5.1: Our pre-training and fine-tuning based approach.

For example, in Figure 5.1(a), there are four known (e.g., synthetic or historical) datasets $\mathcal{D}_1$ to $\mathcal{D}_4$, and we train a model over each one. We call the resultant models (i.e., $\mathcal{M}_1$ to $\mathcal{M}_4$) pre-trained models. If we train model $\mathcal{M}$ over $\mathcal{D}_T$ from scratch, the training cost

[1] https://learnedsystems.github.io/SOSDLeaderboard/leaderboard/

is high. Instead, to save training time, we select $\mathcal{M}_2$ as the model to index $\mathcal{D}_T$ because $\mathcal{D}_2$ is the most similar to $\mathcal{D}_T$ among the four datasets.

Subsequently, Figure 5.1(b) shows the CDFs of the four known datasets and their corresponding models. Figure 5.1(c) shows that $\mathcal{M}$ fits $\mathcal{D}_T$ better than $\mathcal{M}_2$ (which is the best fit among the four pre-trained models), while after fine-tuning, the fine-tuned model $\mathcal{M}_2^{FT}$ almost perfectly fits $\mathcal{D}_T$. In addition, the cost of selecting and fine-tuning model $\mathcal{M}_2$ is substantially cheaper than training $\mathcal{M}$ from scratch.

In terms of CDF fitness, the reason that fine-tuning $\mathcal{M}_2$ to create $\mathcal{M}_2^{FT}$ yields better results than directly training on $\mathcal{D}_T$ is twofold. First, $\mathcal{M}_2$ is initially trained on a smaller but representative dataset, $\mathcal{D}_2$, capturing the essential distribution of $\mathcal{D}$ without overfitting to noise or outliers. Second, fine-tuning $\mathcal{M}_2$ on $\mathcal{D}_T$ adapts the model to the features of the whole dataset, making it more efficient for querying.

A key requirement for successful adaptation of $\mathcal{M}_S$ to $\mathcal{D}_T$ is that $\mathcal{D}_S$ and $\mathcal{D}_T$ should have similar distributions [21, 60]. Otherwise, $\mathcal{M}_S$ may yield large errors on $\mathcal{D}_T$. This motivates us to generate synthetic datasets to cover a wide range of different distributions and pre-train reusable models on such datasets. The next question is then how to select a pre-trained model for $\mathcal{D}_T$, i.e., how to measure the dataset similarity.

Our dataset generation process aims to simulate as many different CDFs as possible. We propose an efficient dataset generation method that takes a CDF distance threshold ϵ and a dataset cardinality n as the input, and it outputs a set of synthetic datasets. Then, we train a model $\mathcal{M}_S$ over every synthetic dataset $\mathcal{D}_S$. To measure the similarity between two datasets, we use a fast implementation of the EMD [27]. When given a new dataset $\mathcal{D}_T$, to reuse a pre-trained model, we measure the EMD between $\mathcal{D}_T$ and all the synthetic datasets. We select a model $\mathcal{M}_S$ where $\mathcal{D}_S$ and $\mathcal{D}_T$ have the highest similarity (smallest EMD). Then, we fine-tune $\mathcal{M}_S$ over $\mathcal{D}_T$ to better fit the data distribution.

To showcase the applicability of our model reuse technique, we integrate it into RMI – a learned index that displays state-of-the-art performance in key lookup speeds [50]. We show that model reuse and fine-tuning can significantly reduce the training time of the sub-models in RMI. We also integrate model reuse and fine-tuning with learned spatial indices to reduce their index building costs.

In summary, our key contributions are:

1. We propose a model reuse and fine-tuning technique to accelerate index building and key lookup. The reused models are trained over synthetic datasets, which are generated based on a heuristic method.

2. To reuse the pre-trained models, we use the earth mover's distance to measure the similarity between the target dataset $\mathcal{D}_T$ and all the synthetic datasets.

3. Extensive experiments on synthetic and real datasets show that model reuse and fine-tuning can accelerate the building of learned one-dimensional indices by 30.4% and improve lookup efficiency by up to 24.4% on real datasets and 22.5% on skewed synthetic datasets. When applied on learned spatial indices, model reuse and fine-tuning reduce index build times by 24 to 125 times, while improving the point query time by up to 13%.

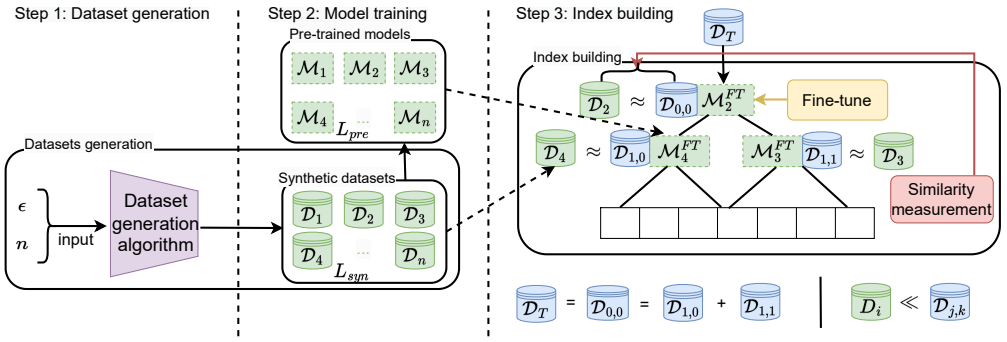


FIGURE 5.2: An overview of dataset generation, model pre-training, and index building. For model reuse and fine-tuning, we use the similarity comparison between $\mathcal{D}_2$ and $\mathcal{D}_{0,0}$ as an example. Then, we fine-tune model $\mathcal{M}_2$ after adaption, which derives $\mathcal{M}_2^{FT}$. Here, $\mathcal{D}_T = \mathcal{D}_{0,0} = \mathcal{D}_{1,0} \cup \mathcal{D}_{1,1}$, i.e., $\mathcal{D}_T$ is $\mathcal{D}_{0,0}$, while $\mathcal{D}_{0,0}$ is separated into $\mathcal{D}_{1,0}$ and $\mathcal{D}_{1,1}$. Any generated synthetic dataset is much smaller than the input datasets in the index building procedure, i.e., $\mathcal{D}_i \ll \mathcal{D}_{j,k}$.

5.2 Model Reuse and Fine-Tuning

We first present an overview of our model reuse technique in Section 5.2.1. We then detail its key components, including dataset similarity measurement in Section 5.2.2, synthetic dataset generation in Section 5.2.3, model adaptation in Section 5.2.4, and fine-tuning in Section 5.2.5.

5.2.1 Solution Overview

Figure 5.2 depicts the overall workflow of our proposed solution. First, we propose a heuristic method to generate synthetic datasets (Step 1). After acquiring a set of synthetic datasets L_{syn}, we prepare the pre-trained models L_{pre}. Each pre-trained model $L_{pre}[i]$ is trained over a synthetic dataset $L_{syn}[i]$ (Step 2). The preparation process is a one-off work and L_{syn} and L_{pre} can be efficiently loaded for index building.

For index (e.g., RMI [53]) building (Step 3), Algorithm 3 summarizes our procedure. The algorithm scans all the synthetic datasets (line 2) and computes the distance (dissimilarity) between $\mathcal{D}_T$ and each synthetic dataset $L_{syn}[i]$ (line 3), to find the dataset $\mathcal{D}_S$ that has the smallest distance to $\mathcal{D}_T$ (lines 4 and 5). After finding the optimal dataset $\mathcal{D}_S$ and model $\mathcal{M}_S$, we adapt (i.e., to fit the data domain) and fine-tune $\mathcal{M}_S$ based on $\mathcal{D}_S$ and $\mathcal{D}_T$ to obtain $\mathcal{M}_S^{FT}$ (lines 6 and 7). Finally, we compute the error range of $\mathcal{M}_S^{FT}$ on $\mathcal{D}_T$ to bound the index lookup range (line 8) and return $\mathcal{M}_S^{FT}$ afterwards (line 9).

Algorithm 3: Model Reuse and Fine-Tuning

Input: $\mathcal{D}_T$, L_{syn}, L_{pre}
Output: $\mathcal{M}_S^{FT}$

1 $dist_{min} \leftarrow MAX\_INT$;
2 **for** $i \in [1, L_{syn}.size()]$ **do**
3 $\quad dist \leftarrow$ `cal_distance`$(L_{syn}[i], \mathcal{D}_T)$;
4 $\quad$ **if** $dist < dist_{min}$ **then**
5 $\quad\quad \mathcal{D}_S \leftarrow L_{syn}[i]$, $\mathcal{M}_S \leftarrow L_{pre}[i]$, $dist_{min} \leftarrow dist$;
6 $\mathcal{M}_S^{FT} \leftarrow$ `adapt_model`$(\mathcal{M}_S, \mathcal{D}_S, \mathcal{D}_T)$;
7 $\mathcal{M}_S^{FT} \leftarrow$ `fine_tune`$(\mathcal{M}_S^{FT}, \mathcal{D}_T)$;
8 $\mathcal{M}_S^{FT}.errors \leftarrow \mathcal{M}_S^{FT}$.`calc_err`$(\mathcal{D}_T)$;
9 **return** $\mathcal{M}_S^{FT}$;

5.2.2 Dataset Similarity Measurement

A model $\mathcal{M}_S$ learns a CDF of dataset $\mathcal{D}_S$. To reuse $\mathcal{M}_S$ on $\mathcal{D}_T$, it is important that the CDFs of $\mathcal{D}_S$ and $\mathcal{D}_T$ are similar. We thus define the distance by the EMD based on the CDFs.

Definition 5.1 (Similarity between two datasets)**.** Given two datasets $\mathcal{D}_S$ and $\mathcal{D}_T$, their *similarity* is defined by the area between their empirical CDFs:

$$\texttt{dist}(\mathcal{D}_S, \mathcal{D}_T) = \int_{-\infty}^{\infty} |cdf_S(x) - cdf_T(x)|dx \tag{5.1}$$

However, directly using EMD in our work has two practical issues: 1) The data ranges of $\mathcal{D}_S$ and $\mathcal{D}_T$ can be different, which will impact the accuracy of EMD; 2) The computation cost of EMD is linear to the dataset size n, i.e., $O(n)$, which can affect index build times for larger datasets.

To address these issues, we observe that the similarity measurement is used to choose a pre-trained model, without stringent requirements for high accuracy. Thus, we propose a fast approximation of the similarity metric using *relative frequency histograms* ("histograms" for short), which discretize the data domain into m bins and record relative data frequencies (i.e., percentages) of each bin. A histogram is a discrete approximation of the Probability Density Function (PDF) of a dataset. We use it to compute approximations of the CDFs and their distance, denoted by $\texttt{dist}(\mathcal{D}_S, \mathcal{D}_T)$.

To generate the histogram, all the data keys are normalized by min-max normalization, such that bin i contains keys in the range $(\frac{i-1}{m}, \frac{i}{m}]$. To compute $\texttt{dist}(\mathcal{D}_S, \mathcal{D}_T)$, we first compute the histograms of $\mathcal{D}_S$ and $\mathcal{D}_T$, denoted by H_S and H_T. We then go through each bin of H_S and H_T, add up the probabilities at the bins (to approximate cumulative probabilities of the CDFs), and record the maximum difference in the accumulated probabilities of $\mathcal{D}_S$ and $\mathcal{D}_T$.

Algorithm 4 summarizes the computation, where the input histograms H_S and H_T each has m (a system parameter) bins. We use $H_S[i]$ and $H_T[i]$ to denote the i-th bins and their relative frequencies. The sum of the probabilities of the first i bins of H_S and H_T are denoted by P_S and P_T, i.e., $P_S = \sum_{j=1}^{i} H_S[j]$ and $P_T = \sum_{j=1}^{i} H_T[j]$.

The algorithm computes *dist*, i.e., an approximation of $\texttt{dist}(\mathcal{D}_S, \mathcal{D}_T)$, by looping through the bins (lines 2 to 4). In the i-th iteration ($i \in [0, m-1]$), it computes $H_S[i] + P_S$. This is the approximate $cdf_S(x)$ for any $x \in (\frac{i-1}{m}, \frac{i}{m}]$ (in our synthetic datasets, $x \in [0, 1]$), because P_S has accumulated the probabilities for $x \leq \frac{i-1}{m}$ while $H_S[i]$ further adds the probability for $x \in (\frac{i-1}{m}, \frac{i}{m}]$. Meanwhile, P_T is the approximate $cdf_T(x)$ for any

Algorithm 4: Approximate EMD

Input: H_S, H_T

Output: *dist*

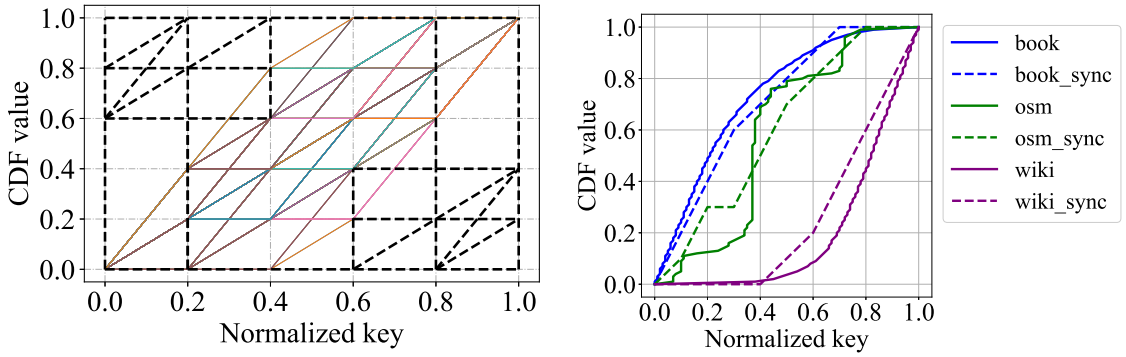
```
dist ← 0, P_S ← 0, P_T ← 0;
for i ∈ [1, m] do
    P_S ← P_S + H_S[i], P_T ← P_T + H_T[i];
    dist ← dist + |P_S − P_T| · 1/m;
return dist;
```

$x \in (\frac{i-1}{m}, \frac{i}{m}]$. Thus, we use $|P_S - P_T|$ and $|P_S - P_T| \cdot \frac{1}{m}$ to approximate the CDF distance and the difference in the area of bin i, respectively.

Using histograms reduces the similarity computation time to $O(\log |\mathcal{D}_T| + m)$, i.e., $O(\log |\mathcal{D}_T|)$ time for H_T computation and $O(m)$ time for Algorithm 4. Histogram H_S is pre-computed since $\mathcal{D}_S$ is known. Its cost is omitted here.

5.2.3 Synthetic Dataset Generation

We aim to generate a set of datasets that can represent any given real dataset with a high similarity. Since it is difficult to determine the position for every single data record in a dataset with large cardinality n, we again approximate the probability density of a dataset by a histogram such that the CDF can be seen as an accumulation of the bins. Then, we can control CDF generation by controlling the number of bins.



(A) The enumeration of all possible CDFs of synthetic datasets ($\epsilon = 0.2$)

(B) CDFs of real and synthetic datasets.

FIGURE 5.3: Examples of the CDFs of synthetic dataset generation and how to approximate real datasets. In (b), the solid lines (e.g., book) are CDFs of real datasets and the dashed lines (e.g., book_sync) are those of synthetic datasets.

To limit the bin value combinations and hence the number of CDFs (synthetic datasets) generated, we use a boundary value ϵ which limits the maximal bin size and limit the probability value of each bin to be within $\{0, \epsilon/2, \epsilon\}$. We use $m = \lceil 2/\epsilon \rceil$ bins in the histogram heuristically. When a target dataset $\mathcal{D}_T$ is matched by a synthetic dataset, their CDF similarity may be within $\epsilon/2$ rather than ϵ, which improves the query performance. Our total number of histograms generated is: $\sum_{i=0}^{m}(C_m^i \cdot C_{m-i}^{\lfloor(1-i\epsilon)/(\epsilon/2)\rfloor})$, where the two combinatorial terms represent the numbers of bins with probability values ϵ and $\epsilon/2$, respectively. Once a histogram is created, we generate a synthetic dataset of n key values ($n = 100$ in our experiments) based on the histogram, where the data range is $[0, 1]$, and random key values are generated for each bin.

Algorithm 5: Synthetic Dataset Generation

Input: ϵ, n
Output: L_{syn}

$m \leftarrow \lceil 2/\epsilon \rceil$;
$bin_h \leftarrow \{0, \epsilon/2, \epsilon\}$;
$Set_H \leftarrow$ histograms with m bins and heights in bin_h;
for $H \in Set_H$ **do**
  $\mathcal{D} \leftarrow \{\}$;
  for $i \in [0, m-1]$ **do**
    Randomly generate $H[i] \cdot n$ records in range $(\frac{i}{m}, \frac{i+1}{m}]$ for $\mathcal{D}$;
  $L_{syn}.add(\mathcal{D})$;
return L_{syn};

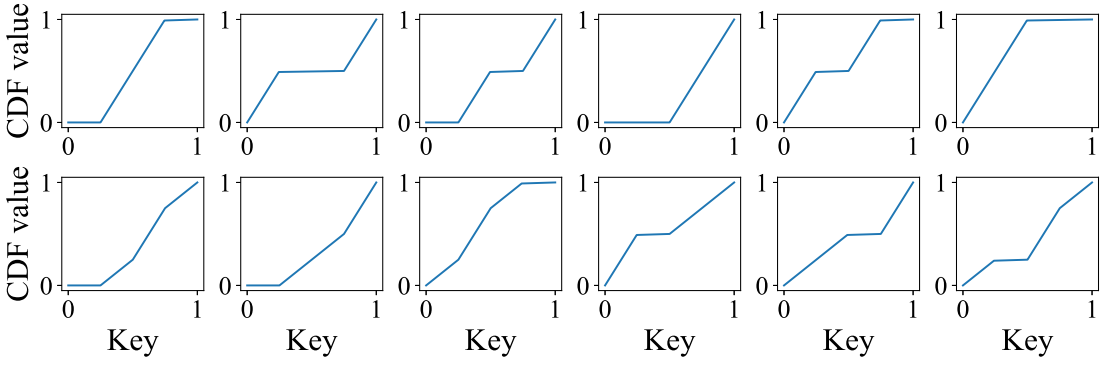


FIGURE 5.4: CDFs of generated synthetic datasets

As shown in Figure 5.3(a), after the generation of L_{syn}, all CDFs lie in a $[0, 1] \times [0, 1]$ space. Any CDF can be seen as a curve that starts at $(0, 0)$ and travels to $(1, 1)$ in a non-decreasing manner (in the CDF value dimension). We discretize this space with a grid, where each row has a height of ϵ ($\epsilon = 0.2$ in the figure), and each column has a width of $\lceil 1/\epsilon \rceil$. Consider the set $\mathcal{L}$ of polylines each starting from $(0, 0)$ and traveling

to $(1, 1)$ via the grid vertices in a non-decreasing manner (in the CDF value dimension, e.g., the colored lines). In Figure 5.3(b) for all the three CDFs of real datasets, there is a polyline $l \in \mathcal{L}$ such that the distance between l and the CDF is bounded by $\epsilon = 0.2$. However, our synthetic datasets will not cover extremely skewed CDFs (e.g., the black polylines in Figure 5.3(a)).

This procedure is shown to be effective and efficient empirically. We summarize the generation procedure in Algorithm 5. In Figure 5.4, we present 12 generated CDFs when $\epsilon = 0.5$.

5.2.4 Model Adaptation

When model $\mathcal{M}_S$ pre-trained on $\mathcal{D}_S$ is selected, we need to adapt $\mathcal{M}_S$ based on the data domains of $\mathcal{D}_S$ and $\mathcal{D}_T$. This is because $\mathcal{M}_S$ will not work properly on a domain over which it was not trained, even if the CDFs of $\mathcal{D}_S$ and $\mathcal{D}_T$ share a similar shape. Let the data ranges of $\mathcal{D}_S$ and $\mathcal{D}_T$ be $[x_S^s, x_S^e]$ and $[x_T^s, x_T^e]$, and their data storage position ranges be $[y_S^s, y_S^e]$ and $[y_T^s, y_T^e]$, respectively. Model $\mathcal{M}_S$ is trained to take a search key in $[x_S^s, x_S^e]$ as the input and predict a storage position in $[y_S^s, y_S^e]$. Here, we assume that $\mathcal{M}_S$ predicts the storage position of record p directly rather than its rank (or percentile), i.e., $p.addr \approx \mathcal{M}_S(p.key)$ (instead of $\mathcal{M}_S(p.key) \cdot |\mathcal{D}_S|$ as shown in Section 5.1). This simplifies the discussion but does not impact our key findings. To adapt $\mathcal{M}_S$ for $\mathcal{D}_T$, we take a search key in $[x_T^s, x_T^e]$, map it into $[x_S^s, x_S^e]$, and feed the mapped value into $\mathcal{M}_S$ for prediction. The predicted output is mapped back into $[y_T^s, y_T^e]$ for $\mathcal{D}_T$.

Let $S_{\Delta x} = \frac{x_S^e - x_S^s}{x_T^e - x_T^s}$ and $S_{\Delta y} = \frac{y_T^e - y_T^s}{y_S^e - y_S^s}$. The input mapping is performed by a linear transformation $\mathcal{T}_{in}(x) = a_1 \cdot x + b_1$ where $a_1 = S_{\Delta x}$ and $b_1 = x_S^s - x_T^s \cdot S_{\Delta x}$. This is an affine transformation that maps the data range (i.e., $\mathcal{T}_{in}(x_T^s) = x_S^s$ and $\mathcal{T}_{in}(x_T^e) = x_S^e$) without changing the distribution. Similarly, the output mapping is done by $\mathcal{T}_{out}(y) = a_2 \cdot y + b_2$ where $a_2 = S_{\Delta y}$ and $b_2 = y_T^s - y_S^s \cdot S_{\Delta y}$.

5.2.5 Fine-Tuning

After model reuse and model adaptation, $\mathcal{M}_S$ is able to index $\mathcal{D}_T$. However, without fine-tuning, the loss may still be high. This is because, while the data distributions are similar in shape, they are not identical. For fine-tuning, we consider both neural networks

and linear models. We use θ to represent the parameters of $\mathcal{M}_S$ and $\theta(t)$ denotes the parameters at epoch t during fine-tuning. We use $f(\mathcal{D}_T.key, \theta(t))$ to represent the prediction result of $\mathcal{M}_S$ on $\mathcal{D}_T$. Thus, the loss function $L(\theta)$, which is an L_2 loss, can be written as: $L(\theta) = ||f(\mathcal{D}_T.key, \theta) - \mathcal{D}_T.addr||_2$.

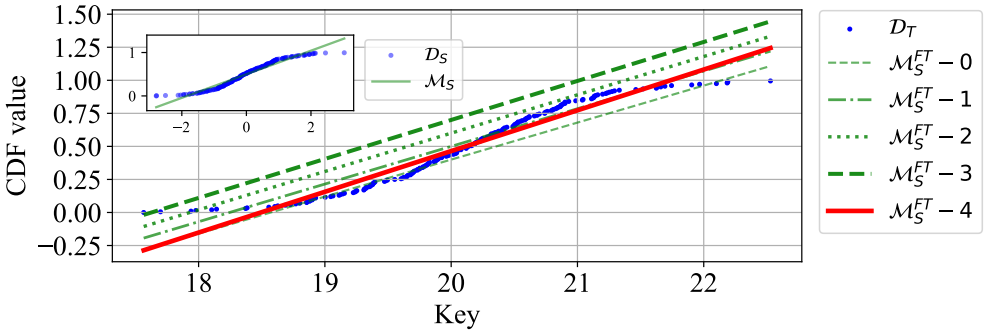


FIGURE 5.5: Model adaptation and fine-tuning, where $\mathcal{M}_S^{FT} - i$ is the ith epoch of fine-tuning ($\mathcal{M}_S^{FT} - 0$ equals to $\mathcal{M}_S$).

Before fine-tuning, $\mathcal{M}_S$ is trained over $\mathcal{D}_S$ such that we denote parameters of $\mathcal{M}_S$ as $\theta(0) = \theta_S$. When we fine-tune $\mathcal{M}_S$, we use gradient descent (GD) with learning rate η and we update the parameters as: $\theta(t+1) = \theta(t) - \eta\frac{\partial L(\theta(t))}{\partial \theta(t)}$, where t denotes the fine-tuning iteration number. In Figure 5.5, the inner figure shows a model $\mathcal{M}_S$ selected for $\mathcal{D}_T$. We first adapt the model $\mathcal{M}_S$ from the key range of $\mathcal{D}_S$ to $\mathcal{D}_T$ as $\mathcal{M}_S^{FT}$. Then, we fine-tune $\mathcal{M}_S^{FT}$ with 4 epochs to better fit $\mathcal{D}_T$.

5.3 Extending to Learned Spatial Indices

So far, we have shown how model reuse and fine-tuning can help streamline the building of learned one-dimensional indices. In this subsection, we will extend this technique to learned spatial indices. Such an extension is feasible for the following reasons.

1. Some learned spatial indices require an initial ordering of data points to derive the CDF of a dataset. This is a process analogous to the one used by learned one-dimensional indices. Thus, a pre-trained model, when applied to spatial data points, possesses the capability to produce results comparable to those of the models learned from scratch.

2. The key idea behind using pre-trained models hinges on generating synthetic datasets that closely mimic the distribution of real-world datasets. This same approach can be readily extended to accommodate spatial datasets.

To utilize pre-trained models for building learned spatial indices, it is essential to understand the types of models trained within such indices. Currently, there are two main categories of models:

1. One-dimensional models: These are models that are trained on one-dimensional data. For instance, learned spatial indices such as ZM-index [103], ML-index [23], and LISA [58] transform spatial data into one-dimensional values, and then conduct training on the mapped one-dimensional data.

2. Multi-dimensional models: These are models that are trained on multi-dimensional, specifically spatial data. For example, RSMI [84] directly processes spatial data with two dimensions, in line with its design to accommodate two-dimensional data.

For the one-dimensional model-based learned spatial indices, we can directly use the pre-trained models that have been trained on one-dimensional indices. However, for indices such as RSMI or others that are trained over multi-dimensional data, we need to develop pre-trained models specifically designed to handle multi-dimensional data.

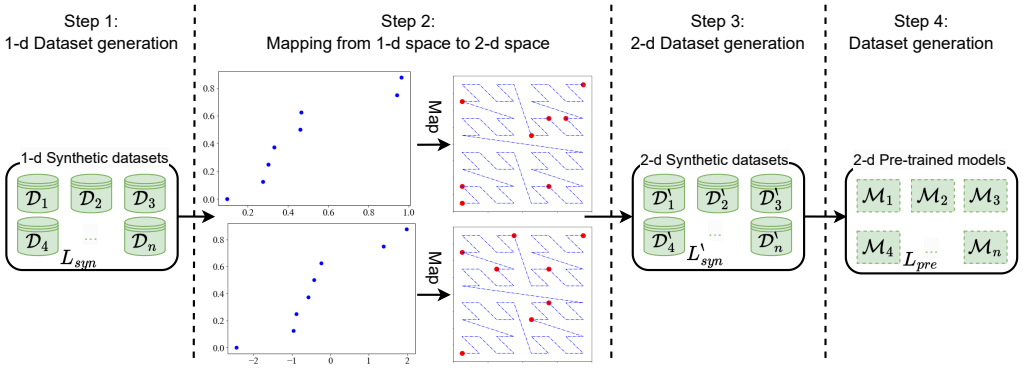


FIGURE 5.6: Generating two-dimensional pre-trained models for RSMI based on one-dimensional datasets.

To expand the applicability of model reuse to multi-dimensional model-based learned spatial indices, we adapt the dataset generation approach as described earlier for one-dimensional data. As depicted in Figure 5.6, there are four steps to generate two-dimensional pre-trained models using one-dimensional synthetic datasets, taking RSMI as an example.

- Step 1: We employ the dataset generation technique described in Section 5.2.3 to produce synthetic datasets. Note that we do not generate multi-dimensional synthetic datasets. This is because generating multi-dimensional datasets requires a similarity measurement between multi-dimensional datasets which can be unnecessarily difficult, and there is no efficient method for this purpose to the best of our knowledge.

- Step 2: We now have one-dimensional datasets, while RSMI models are trained on spatial datasets. We take a one-dimensional data to spatial data mapping strategy to generate spatial datasets from the one-dimensional datasets. This is feasible since the mapping function utilized by the SFCs such as ZCs is bijective. Given a synthetic dataset $\mathcal{D}_S$, we compute the Z-value for each data point with key value key by $CDF_{\mathcal{D}_S}(key) \cdot \ell^2$, where ℓ^2 represents the number of cells in the grid partitioning of a ZC to go through the data space.

- Step 3: Given the Z-values of a synthetic dataset $\mathcal{D}_S$, we map the Z-values to the grid cells and further to the coordinates of the center points of the cells, which form a synthetic spatial dataset corresponding to $\mathcal{D}_S$ (cf. red points in Figure 5.6).

- Step 4: We now pre-train an index model on the generated synthetic spatial dataset, using FFNs. Note that the size of the FFN (i.e., the number of neurons in the hidden layer) is determined by the size of the training set. To facilitate model reuse, it is important to maintain uniformity in the sizes of the FFNs, i.e., the number of neurons must be consistent across all FFNs.

5.4 Experiments

We report experimental results primarily on one-dimensional learned indices, in Sections 5.4.1 and 5.4.2. We show additional results on learned spatial indices in Section 5.4.3, to show the wider applicability of the proposed techniques.

5.4.1 Experimental Setup

The original implementation of RMI [53] is based on neural networks. However, linear models have lower build costs, which can accelerate both the index build time and

lookup time. Thus, we use linear models in the experiments, which are implemented using *Scikit-learn*. All experiments are performed on a 64-bit machine with a 3.60 GHz Intel i9 CPU, RTX 2080Ti GPU, 64 GB RAM, and a 1 TB hard disk drive.

Models tested. We compare our approach against both traditional and learned indices:

1. BTree [95] – a C++ based in-memory B^+-tree.
2. RMI [53] – the recursive model index using linear models.
3. PGM [30] – a piecewise geometric model index.
4. RS [49] – a single-pass learned index.
5. ALEX [25] – an updatable adaptive learned index.
6. LIPP [105] – an updatable learned index that yields precise positions of the search keys.

Proposed models. We evaluate the following adapted models equipped with our index building techniques:

1. RMI-MR[2] – RMI enhanced with model reuse (but no fine-tuning)
2. RMI-MR-FT[2] – RMI-MR with model fine-tuning.

Implementation details. The experimental setup follows the SOSD benchmark [50].

We summarize the number of synthetic datasets (each with $n = 100$) and the time to pre-train models on them in Table 5.1. When ϵ is smaller than 0.2, the number of bins (i.e., $\lceil 2/\epsilon \rceil$) is getting large such that the number of synthetic datasets explodes. Thus, $\epsilon = 0.2$ is the minimum value that we can use. To balance the computation cost and the fitness of model reuse, we use $\epsilon = 0.3$ as the default value in the later experiments. The pre-trained models and synthetic datasets can be loaded in memory within a second (less than 1 MB in size for linear models); and the total model comparison time to build an index in any of the experiments is also within a second.

For all the baselines, we use their published source code and default configurations. Following the SOSD benchmark, the BTree is built based on sampling data records

[2] https://anonymous.4open.science/r/MR_FT_on_SOSD-7C1B

from an input dataset, where the sampling rate varies from 1 to 2^{-16}. For RMI, we re-implement it using C++ and extend the implementation to integrate model reuse. For fine-tuning, the learning rate is 0.01 and we sample data records with a sampling rate of 0.02. For the calculation of EMD, we set the number of bins to be $m = 10$ for each histogram.

TABLE 5.1: Summary of synthetic datasets

ϵ	0.2	0.3	0.4	0.5
Number of bins (m)	10	7	5	4
Number of datasets	8,953	987	95	19
Model training time (s)	839.5	63.5	8.8	2.1

Datasets. Following SOSD, we use four real datasets: `amzn` (default) – an Amazon book popularity dataset, `face` – a Facebook user ID dataset, `osm` – an OpenStreetMap cell ID dataset, and `wiki` – a Wikipedia edit timestamp dataset. We further generate `skewed` datasets from uniform data by raising a key value x to its powers x^α ($\alpha = 3, 5, 7, 9$), following a study on data indexing [85]. Each dataset contains 200 million unsigned 64-bit integer keys (1.6 GB in size).

Performance metrics. For both real and synthetic datasets, we follow the pareto analysis in SOSD [61], which has ten configurations (i.e., different hyperparameter settings) ranging from minimum to maximum size for each index. For RMI, PGM, and RS, the hyperparameter is the number of models, the threshold of the error bound, and the number of radix bits, respectively. For BTree and ALEX, the hyperparameter is the number of sampled data records used to build the index. LIPP can only be built on the full datasets and hence cannot support the pareto analysis using sampled subsets. Thus, only one data point is recorded for it in each result figure. The SOSD benchmark queries over 10 million sampled keys to show the average lookup (i.e., a query key is in the indexed data) time as the index build time and index size change.

Here, we do not repeat the experiments with differ execution order of the queries, as the order may impact query time results because of caching, which has already been disabled in the SOSD framework.

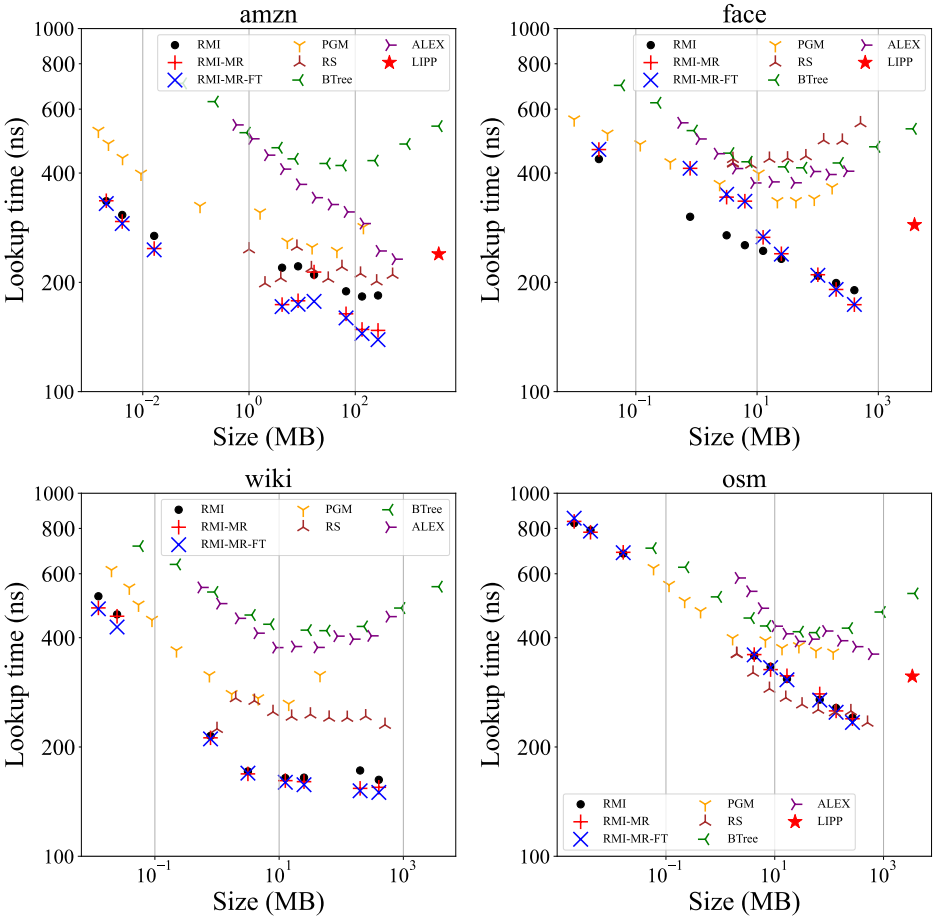


FIGURE 5.7: Index size vs. lookup time over real datasets (LIPP cannot be used on `wiki`).

5.4.2 Results

Index size and lookup time over real datasets. Figure 5.7 shows that the query efficiency of all learning-based indices is improved as the index size increases. This is because querying over learned indices uses model predictions and then a binary search. A larger number of models can partition the data range into smaller segments, which leads to a smaller search range and decreases the cost of binary search. For BTree, a larger index size means more data records are sampled to build the index, which leads to a trade-off between the tree height and the search range of a leaf node in the tree. Thus, as the index size increases, we first see a drop in lookup time, and then a rise when the index size is larger than about 100 MB.

Under similar index sizes, RMI shows better query performance than ALEX, PGM, and RS in most cases except for `osm` where RS overlaps with RMI in some cases, which is consistent with observations from the SOSD benchmark. RMI-MR and RMI-MR-FT

further show slightly better lookup performance than RMI. For example, on `amzn` the improvement by model reuse is up to 20.1% (147 ns vs. 184 ns) and 24.4% (139 ns vs. 184 ns) with further fine-tuning when the index size is 256 MB.

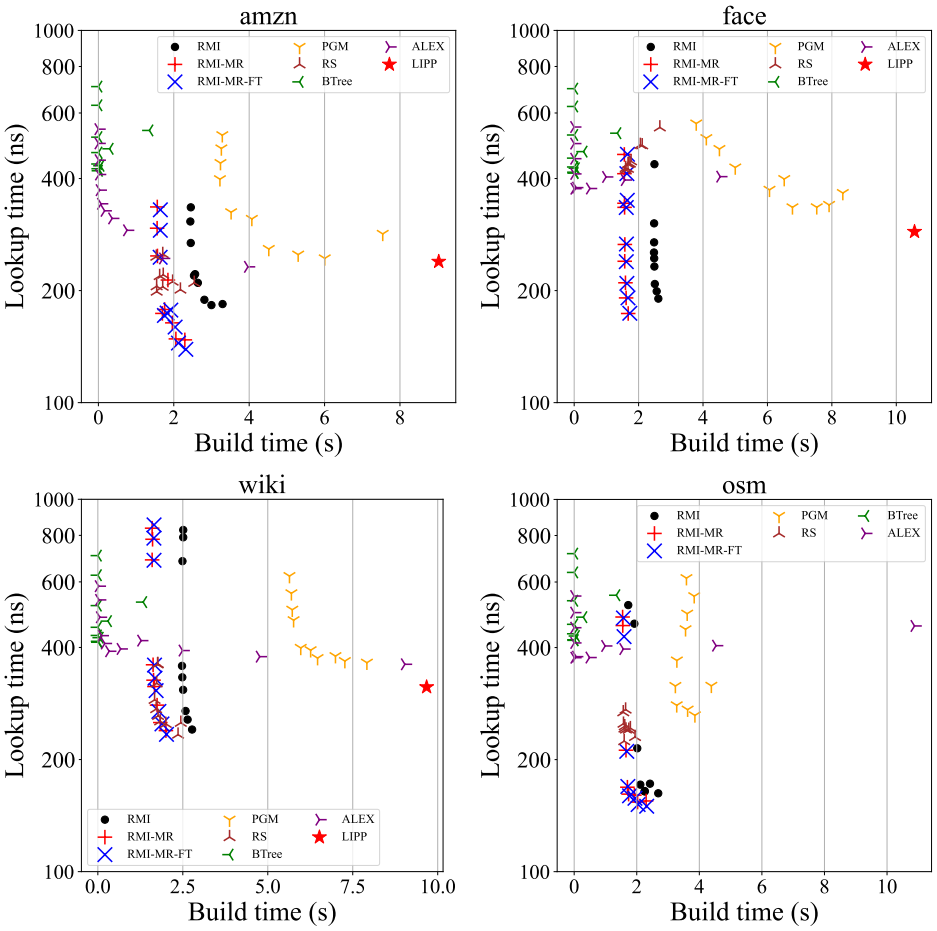


FIGURE 5.8: Build time vs. lookup time over real datasets.

Index build time and lookup time over real datasets. In Figure 5.8, we see that RMI-MR dominates RMI in index build times and query efficiency, especially on the `amzn` dataset, where there is a 30.3% (2.29 s vs. 3.29 s) reduction in the index build time. That means, even when RMI trains the linear models with a one pass strategy [50], we can still improve the build time through pre-trained models without jeopardising the query performance. In addition, the query performance can be further improved by fine-tuning (i.e., RMI-MR-FT) because we fine-tune the models with a small number of sampled data records, which brings just a marginal increment in the build time. PGM is slower in lookup under different index size settings. Smaller PGM structures have a large search range, while larger PGM structures have more layers. These lead to high

prediction costs. ALEX has an index structure similar to BTree such that they share similar lookup performance.

Index size and lookup time over skewed datasets. Figure 5.9 shows that RMI, RMI-MR, and RMI-MR-FT outperform the other methods in lookup time when the index size is large. This is because RMI has just two levels in practice, and the models in the last level (i.e., the leaf level) fit the skewed data records well given a large number of models. Like before, RMI-MR and RMI-MR-FT outperform RMI. For example, when the skewness parameter $\alpha = 3$, the gain in lookup time by model reuse is up to 20.6% (127 ns vs. 160 ns) using an index size of 256 MB. RMI-MR-FT further improves RMI by 22.5% (124 ns vs. 160 ns) in lookup time.

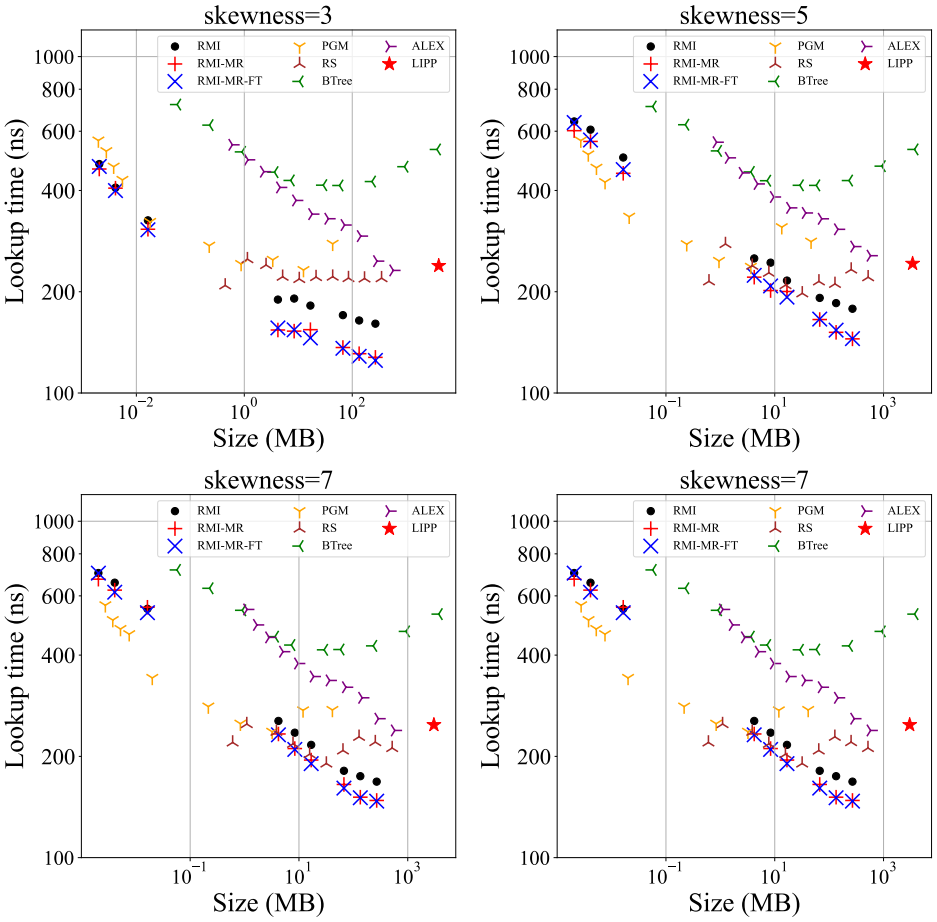


FIGURE 5.9: Index size vs. lookup time over skew datasets.

Index build time and lookup time over skewed datasets. From Figure 5.10, we see that RMI-MR significantly outperforms RMI in both lookup and build times for most cases. The results are similar to those on real datasets. Besides, adding fine-tuning (RMI-MR-FT) slightly improves query efficiency compared to RMI-MR, in the cost of

just around 0.1 seconds extra in build times when $\alpha \leq 7$. When $\alpha = 9$, the fine-tuning cost increases given larger index sizes, because there are more models to tune to fit a highly skewed data distribution.

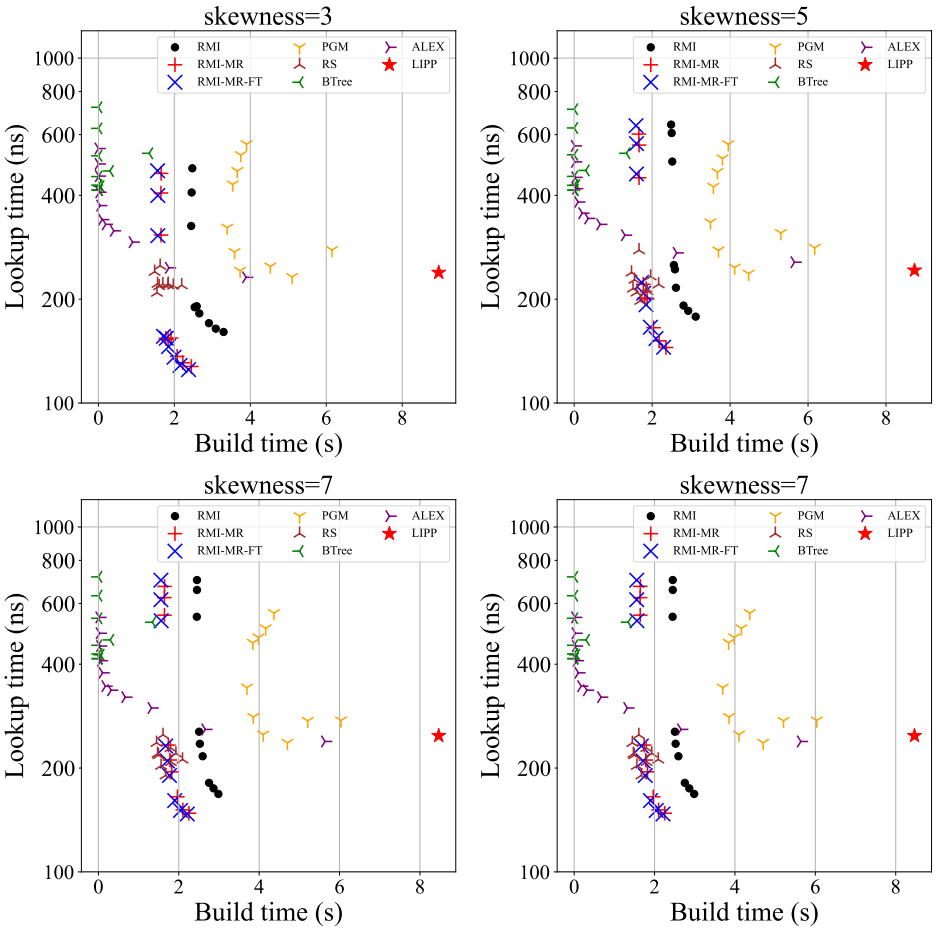


FIGURE 5.10: Build time vs. lookup time over skew datasets

5.4.3 Additional Results on Learned Spatial Indices

We have shown a reduction in index build times when utilizing pre-trained models for learned one-dimensional indices. To further show the applicability of pre-trained models for building learned spatial indices, we integrated our proposed techniques with three such indices, namely, ML [23], RSMI [84], and LISA [58], as described in Section 4.6.

We conducted experiments using four real spatial datasets and the default parameters of each learned spatial index, following the experimental setup described in Section 4.6. To evaluate the efficiency of index building, we measure the index build time and examined

the extent to which model reuse can help improve. Additionally, we show the point query time to observe if the query performance is impacted by model reuse.

In our experiments, the implementation of ML, RSMI, and LISA based on model reuse are denoted as "**ML-MR**", "**RSMI-MR**", and "**LISA-MR**", respectively. We compare these MR-based indices with the original versions which do not use MR, as well as four traditional spatial indices: **Grid** [72], **KDB** [87], **HRR** [85], and **RR*** [10].

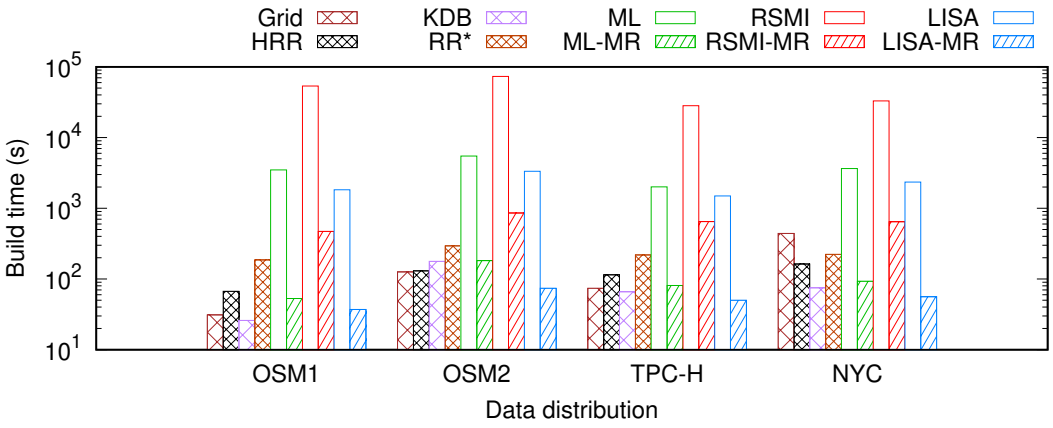


FIGURE 5.11: Build time vs. data distribution.

Index build time over real datasets. Figure 5.11 illustrates the impact of model reuse on the index build time of the learned spatial indices ML, LISA, and RSMI. Using model reuse, the index build times of the learned spatial indices are significantly reduced, reaching a level comparable to that of traditional indices. The performance gains range from at least 24 times (81 s vs. 2,011 s for ML-MR and ML on TPC-H) to up to 125 times (472 s vs. 53,435 s for RSMI-MR and RSMI on OSM1). Notably, LISA-MR performs on par with all the traditional indices across all real datasets.

Comparing these results to the experimental results earlier (e.g., Figure 5.8), it is evident that model reuse improves index build times for both learn one-dimensional indices and learned spatial indices. While the index build times on one-dimensional data is inherently shorter, those on spatial data are much longer such that the improvement achieved by model reuse becomes even more pronounced.

Point query time over real datasets. Figure 5.12 further shows that model reuse retains the high query time efficiency of the learned spatial indices. Using model reuse, the point query time for the learned spatial indices increases by up to 18% (0.497 μs for RSMI-MR vs. 0.421 μs for RSMI on the TPC-H dataset), while it can also decrease

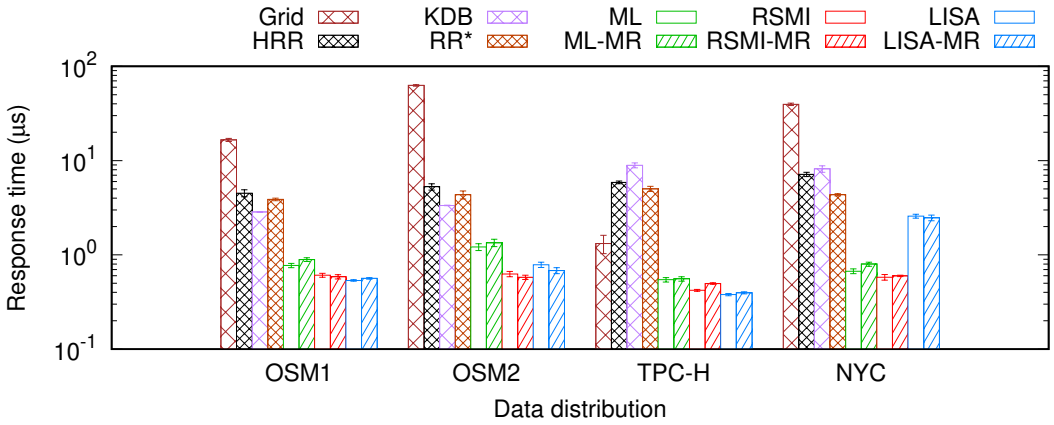


FIGURE 5.12: Point query time vs. data distribution.

by up to 13% (0.683 μs for LISA-MR vs. 0.786 μs for LISA on the OSM2 dataset). Such impacts are minor compared with the query performance gap between the learned spatial indices and the traditional ones. Overall, the results confirm that model reuse can also help learned spatial indices improve their index build time while retaining their low query time.

5.5 Summary

We proposed to use pre-trained models and fine-tuning for indexing new datasets to address the building overhead of learned indices. We proposed a CDF-based synthetic dataset generation method to generate a number of pre-trained models and use EMD as a similarity metric to select candidate pre-trained models for our learned index. We demonstrated the effectiveness of the proposed techniques by applying them to the RMI learned indices [53]. Experimental results on both synthetic and real data show that our model reuse and fine-tuning techniques can improve the building and the lookup performance of RMI, reducing the index build time by 30.4% on real datasets and 22.5% on skewed datasets. Furthermore, model reuse and fine-tuning significantly reduce index build times for learned spatial indices, i.e., by one to two orders of magnitude, to the level of those of the traditional indices, while retaining the high query efficiency of the learned spatial indices. These results show a strong potential of the model reuse techniques to be used in real systems to enhance the practical applicability of the learned indices.

Chapter 6

Efficiently Optimizing Traditional Spatial Indices via SFC Learning

In Chapter 4 and Chapter 5, we proposed new techniques to improve the build time efficiency of learned spatial indices. As discussed in Chapter 1, while learned spatial indices have shown strong query performance, their structure and query algorithms differ drastically from the traditional indices, which are well supported by off-the-shelf database systems.

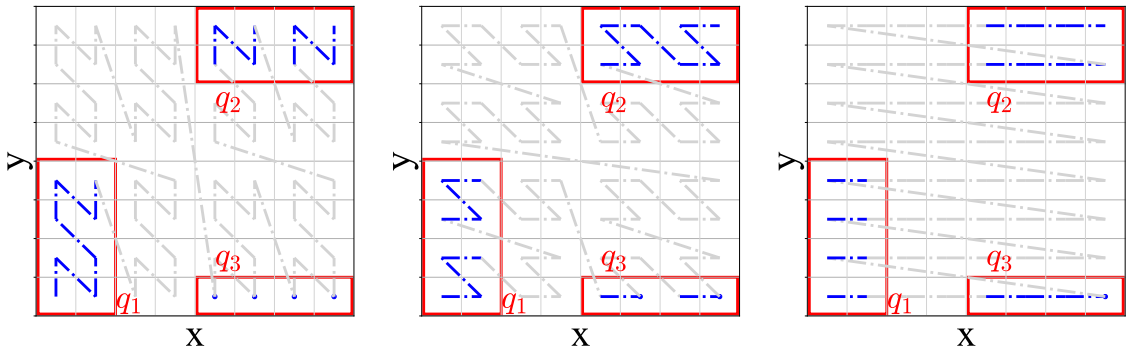
To reduce the additional overhead of replacing traditional indices with learned spatial indices, in this chapter, we take a different perspective and apply learning-based techniques to optimize the structure of traditional spatial indices. We focus on SFC-based spatial indices and propose RL-based methods to optimize such indices. We design techniques for efficient, constant time query cost estimation that enable *online* and *offline* cost estimation when using different SFC indices. First, given a query and a set of available SFC indices, our methods enable selecting the most efficient SFC index in a minimally intrusive manner, enabling practical application in database systems that support different SFC indices. Second, given a query workload, our methods leverage reinforcement learning to enable choosing an SFC index for processing the workload efficiently. Our experimental study offers evidence of the effectiveness and efficiency of the proposed methods. In nearly all settings considered, the chosen SFC indices outperform competing indices such as ones based on Z and Hilbert curves.

The work is in preparation for submission to *International Conference on Management of Data* (**SIGMOD**) 2024.

6.1 Overview

Indexing is essential to enable efficient query processing on increasingly massive data, including spatial and other low-dimensional data. In this setting, indices based on SFC are used widely. For example, ZC (see Figures 6.1(a) and 6.1(b)) [78] are used in Hudi [4], RedShift [2], and SparkSQL [22]; Lexicographic-order Curves (LC) (see Figure 6.1(c)) are used in PostgreSQL [82] and SQL Server [64]; and HC [29] are used in Google S2 [90]. Next, the arguably most important type of query in this setting is the range query that also serves as a foundation for other queries such as kNN queries.

The most efficient query processing occurs when the data needed for a query result is stored consecutively, or when the data is stored in a few blocks of needed data. Thus, the storage organization—the order in which the data is stored—affects the cost of processing a query profoundly. When using SFC indices to index data, the data is ordered according to its SFC ordering, and the choice of which SFC to use for the indexing is important.



(A) Curve 1 (ZC) (B) Curve 2 (ZC) (C) Curve 3 (LC)

FIGURE 6.1: Examples of SFCs (in grey) and queries (in red).

Different range queries benefit differently from different SFC indices. In Figure 6.1, three SFCs on the same data space are shown along with three queries. The fewer disconnected segments of an SFC that need to be accessed to compute a query, the better. To compute q_1, the SFC in Figure 6.1(a) is preferable because only a single segment needs to be accessed. Put differently the data needed may be in a single or in consecutive blocks. In contrast, the SFCs in Figures 6.1(b) and 6.1(c) map the data to two and four segments, respectively.

Next, we observe that no single SFC index is optimal for all queries. While the SFC in Figure 6.1(a) is good for q_1, it is sub-optimal for q_2 and q_3. It is thus critical to

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

Existing studies [68, 73, 107] provide an effective cost model based on counting the number of clusters (continuous curve segments) covered by a query. However, the cost model computation methods rely on a curve segment scan that requires $O(V)$ time, where V is proportional to the size of a query. Given a workload of n queries and m candidate SFCs, $O(n \cdot m \cdot V)$ time is needed to choose an SFC index, which is expensive given large n and m (e.g., a $k \times k$ grid can form $m = k^2!$ candidate SFCs), thus jeopardizing the applicability of the cost model in practice.

We address the efficiency issue of SFC index cost estimation such that a query-optimal SFC index can be found efficiently. We present algorithms that compute the cost of a query in $O(1)$ time. After an $O(n)$-time initialization, the algorithms compute the cost of n queries in $O(1)$-time for each new SFC to be considered. This means that given m candidate SFCs our algorithms can find the optimal SFC index in $O(m)$ time, which is much smaller than $O(n \cdot m \cdot V)$ and enables practical application of SFC index cost estimation.

Our algorithms are based on a well-chosen family of SFCs, the BMC [73]. A BMC maps spatial points by merging the bit sequences of the point coordinates (i.e., column indices) from all d dimensions (detailed in Section 6.2.1). We consider BMCs for two reasons:

1. BMCs generalize ZC and LC, which are used in real systems [4, 22, 64, 82]. Algorithms to find optimal BMC indices can be integrated seamlessly into real systems.

2. The space of BMCs is large. For example, in a 2-dimensional space ($d = 2$) where each dimension uses 16 bits ($\ell = 16$) for a column index, there are $k = 2^\ell$ columns in each dimension of the grid. This yields about 6×10^8 (i.e., $\frac{(d \cdot \ell)!}{(\ell!)^d}$) candidate BMCs. An efficient cost model enables searching for a query-efficient SFC index in this large space.

Our algorithms model the cost of a range query based on the number and lengths of curve segments covered by the query, which in turn relate to the difference between the curve values of the end points of each curve segment. We exploit the property that

the curve values of a BMC come from merging the bits of the column indices. This property leads to a closed-form equation to compute the length of a curve segment in $O(d \cdot \ell) = O(1)$ time (given that d and ℓ are constants) for n queries. The property also enables pre-computing d look-up tables that allow computing the number of curve segments in $O(d \cdot \ell) = O(1)$ time. Thus, we achieve constant-time cost estimation.

Our cost estimation algorithms enable two different use cases.

1. We enable *online* selection of a query-efficient index when a query arrives, given a dataset indexed by multiple BMC indices (e.g., ZC and LC variants with different orderings of the bits from different dimensions). This application is minimally intrusive to existing systems and hence is highly practical.
2. Given a dataset and a query workload, we enable *offline* selection of a query-efficient BMC index through efficient exploring of a large space of curves.

In summary, this chapter makes the following contributions:

1. We propose algorithms for efficient range query cost estimation for BMC indices on spatial datasets. Our algorithms can compute the cost of a range query in $O(1)$-time as well as the cost of a workload of n queries in $O(1)$-time, after a simple scan over the queries.
2. Based on this efficient cost estimation, we propose two algorithms, Selected Bit-merging Curve (SBMC) and Learned Bit-merging Curve (LBMC), for query optimization, where SBMC selects an existing query efficient BMC index for query processing, while LBMC selects a BMC index that enables efficient processing for a query workload.
3. We evaluate the cost estimation and the SBMC and LBMC algorithms on real datasets, finding that (i) the proposed cost estimation algorithms outperform baselines consistently by up to 10^5 times, (ii) algorithm SBMC can improve query efficiency by up to 75.2% with only 0.1 ms extra time cost for each query, and (iii) algorithm LBMC has the lowest index building cost and outperforms all baselines in most settings.

Next, we start with the core concepts used in this chapter (Section 6.2), followed by our efficient cost estimation model and algorithms (Section 6.3). Based on the cost estimation, we propose algorithms to optimize SFC-based spatial indices (Section 6.4), which are verified with an empirical study (Section 6.5).

6.2 Concept Definitions

Our goal is to derive a cost model that can be efficiently computed to quantify the cost of range query processing with a BMC index, which will guide the search for a query cost-optimal BMC index. We start with core concepts underlying BMCs and BMC index search, and we list frequently used symbols in Table 6.1.

TABLE 6.1: Frequently used symbols

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

6.2.1 BMC Definition

A BMC maps spatial points by merging the *bit sequences* of the coordinates (i.e., column indices) from all d dimensions into a single bit sequence that becomes a one-dimension value [73].

Figure 6.2 plots three BMCs that are represented by YXYX, YXXY, and YYXX. Here, the ordering of the X's and Y's specify how the bits from dimensions x and y are

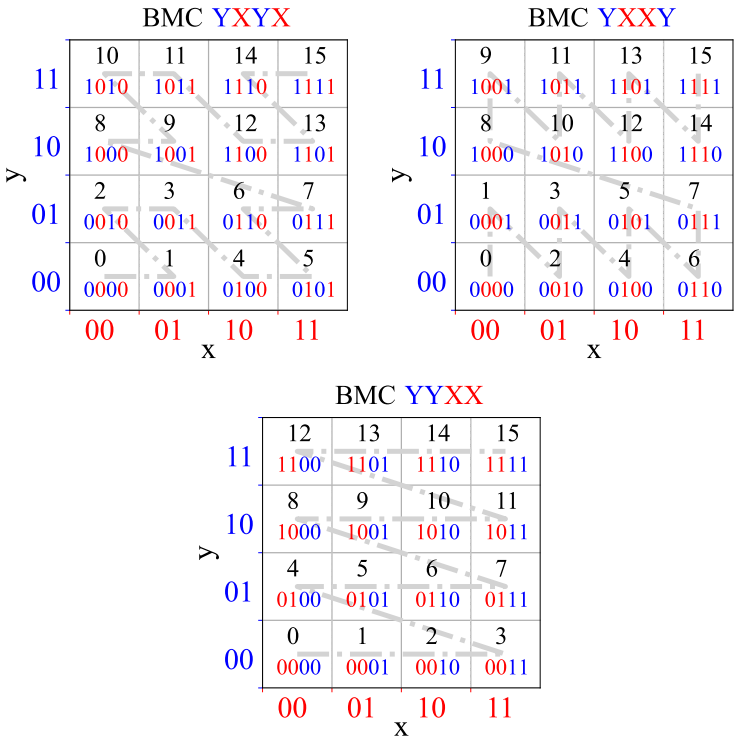


FIGURE 6.2: BMC examples ($d = 2$ and $\ell = 2$).

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

The bit length ℓ is determined by the grid resolution, a system parameter. We use the same ℓ for each dimension to simplify the discussion, and we use the little endian bit order, i.e., the rightmost bit has the lowest rank (Figure 6.3). For simplicity, we call the column indices of a point p in a cell (or the cell itself) the *coordinates* of p (or the cell).

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

$$\mathcal{F}_\sigma(p) = \sum_{i=1}^{d} \sum_{j=1}^{\ell} \alpha_i^j \cdot 2^{\gamma_i^j} \tag{6.1}$$

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

$$\sum_{j=1}^{\ell} \alpha_i^j \cdot 2^{\gamma_i^j} = x_i \tag{6.2}$$

For ease of discussion, we use examples with up to three dimensions x, y, and z. Figure 6.3 calculates $\mathcal{F}_\sigma(p)$ for $p = (2, 1, 7)$ given σ = XYZXYZXYZ. Here, $\alpha_3^1 = 1$ is the first bit value in dimension z, and the rank of the first (i.e., rightmost) Z bit in σ is zero, which means $\gamma_3^1 = 0$. To calculate the curve value of a point for a given σ, we derive each α_i^j and γ_i^j based on x_i and σ, respectively.

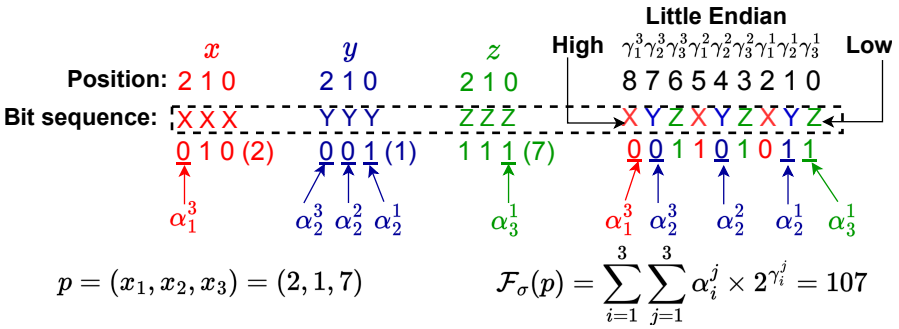


FIGURE 6.3: BMC curve value calculation ($d = 3$ and $\ell = 3$).

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

Theorem 6.1 (Monotonicity)**.** *Given* $p_1 = (x_{1,1}, \ldots, x_{1,d})$ *and* $p_2 = (x_{2,1}, \ldots, x_{2,d})$ *then* $\forall i \in [1, d](x_{1,i} \leq x_{2,i}) \rightarrow \mathcal{F}_\sigma(p_1) \leq \mathcal{F}_\sigma(p_2)$.

Proof. Given $x_{1,i} \leq x_{2,i}$, we have $\sum_{j=1}^{\ell} \alpha_{1,i}^j \cdot 2^{\gamma_{1,i}^j} \leq \sum_{j=1}^{\ell} \alpha_{2,i}^j \cdot 2^{\gamma_{2,i}^j}$ based on Equation 6.2. Since this holds for any $i \in [1, d]$, we have $\sum_{i=1}^{d}\sum_{j=1}^{\ell} \alpha_{1,i}^j \cdot 2^{\gamma_{1,i}^j} \leq \sum_{i=1}^{d}\sum_{j=1}^{\ell} \alpha_{2,i}^j \cdot 2^{\gamma_{2,i}^j}$. Thus, $\mathcal{F}_\sigma(p_1) \leq \mathcal{F}_\sigma(p_2)$ based on Equation 6.1. □

6.2.2 Range Querying Using a BMC index

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

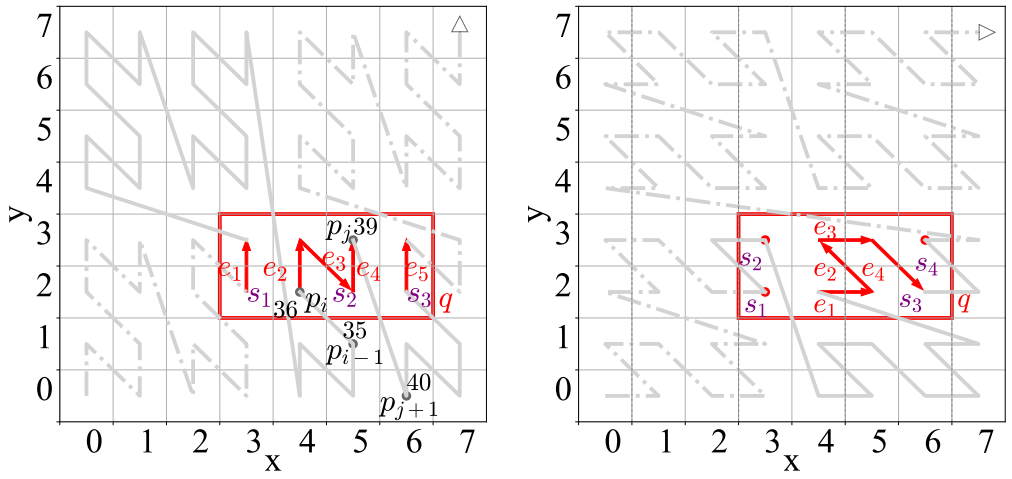
As mentioned earlier, computing a query q using different BMC indices can lead to different costs. To simplify the discussion for determining the cost of a query, we use the following corollary.

Corollary 6.2. *Given* $p_s = (x_{s_1}, \ldots, x_{s_d})$ *and* $p_e = (x_{e_1}, \ldots, x_{e_d})$*, any query* q *is bounded by the curve value range* $[\mathcal{F}_\sigma(p_s), \mathcal{F}_\sigma(p_e)]$.

Corollary 6.2 follows directly from the monotonicity of BMCs (Theorem 6.1). To simplify the discussion, we use a point p and the cell that encloses p interchangeably and rely on the context to disambiguate.

We focus on range queries as they offer a direct reflection of data point ordering, which has a substantial impact on query performance in databases. Unlike point queries, which are primarily dependent on index structure, or k-NN queries, which can be viewed as iteratively expanded range queries, the performance of range queries is more indicative of the underlying data organization.

Query section [73]. A continuous curve segment in a query q is called a *query section*. We denote a query section s with end points p_i and p_j by $[\mathcal{F}_\sigma(p_i), \mathcal{F}_\sigma(p_j)]$. Intuitively, each query section translates to a one-dimensional range query $[\mathcal{F}_\sigma(p_i), \mathcal{F}_\sigma(p_j)]$ on a B$^+$-tree index on dataset D. Thus, the number of query sections in $[\mathcal{F}_\sigma(p_s), \mathcal{F}_\sigma(p_e)]$ determines the cost of q.



(A) BMC XYXYXY (B) BMC YXYXYX

FIGURE 6.4: Query sections and directed edges in BMC indices

Example 6.1. *In Figure 6.4(a), there are three query sections s_1, s_2, and s_3, with $s_2 = [\mathcal{F}_\sigma(p_i), \mathcal{F}_\sigma(p_j)] = [36, 39]$. By definition, a point (cell) immediately preceding p_i or succeeding p_j must be outside q; otherwise, it is part of the query section. For example, p_{i-1} ($\mathcal{F}_\sigma(p_{i-1}) = 35$) and p_{j+1} ($\mathcal{F}_\sigma(p_{j+1}) = 40$) in Figure 6.4(a) are outside q. The number of query sections in q varies across different BMC indices, e.g., the same q as in Figure 6.4(a) has four query sections in Figure 6.4(b).*

Directed edge [107]. Query sections are composed by connecting a series of points (cells). The pair of two consecutive points p_i and p_j forms a *directed edge* (denoted by e) if the curve values of p_i and p_j differ by one under a given σ, i.e., $\mathcal{F}_\sigma(p_j) - \mathcal{F}_\sigma(p_i) = 1$. As each point is represented through a binary value, the difference occurs because $\mathcal{F}_\sigma(p_i) = \underbrace{...}_{prefix}\underline{0}\underbrace{1...1}_{K\ 1\text{s}}$ and $\mathcal{F}_\sigma(p_j) = \underbrace{...}_{prefix}\underline{1}\underbrace{0...0}_{K\ 0\text{s}}$, where the last K ($K \geq 0$) bits are changed from 1 to 0 and the $(K+1)$st bit is changed from 0 to 1.

Example 6.2. *We use the binary form of two pairs of integers that form directed edges to illustrate this concept, one for $K > 0$ and the other for $K = 0$. First, suppose that the binary representations of $\mathcal{F}_\sigma(p_i) = 15$ and $\mathcal{F}_\sigma(p_j) = 16$ are **0**$\underline{0}$1111 and **0**$\underline{1}$0000, respectively. In this case, four bits starting from the right (i.e., $K = 4$) are changed from 1 to 0, and the fifth bit is changed from 0 to 1. The last bit **0** is the shared prefix. Second, if the binary forms of $\mathcal{F}_\sigma(p_i) = 16$ and $\mathcal{F}_\sigma(p_j) = 17$ are **01000**$\underline{0}$ and **01000**$\underline{1}$, respectively, only the first bit (from the right) is changed from 0 to 1, i.e., no bits ($K = 0$) are changed from 1 to 0, and the shared prefix is **01000**.*

We explain now why the number of directed edges (denoted by $\mathcal{E}_\sigma(q)$) plus the number of query sections (denoted by $\mathcal{S}_\sigma(q)$) in a given query q yields the number of distinct points (denoted by $\mathcal{V}(q)$) in q. The intuition is that if q consists of a single section s, i.e., the curve stays completely inside s and $\mathcal{S}_\sigma(q) = 1$ then there are $\mathcal{V}(q) - 1$ directed edges connecting a given start point p_s and end point p_e of s. In other words, we obtain $\mathcal{E}_\sigma(q) + \mathcal{S}_\sigma(q) = \mathcal{V}(q) - 1 + 1 = \mathcal{V}(q)$. Every time a curve exits a query section s_i and enters the next section s_{i+1}, the last point in s_i becomes disconnected (minus one directed edge) but one new query section is added (plus 1 for the query section) when the curve reenters s_{i+1}. This leads to the following equation:

$$\mathcal{E}_\sigma(q) + \mathcal{S}_\sigma(q) = \mathcal{V}(q) \tag{6.3}$$

While $\mathcal{V}(q)$ is independent of σ, the values for $\mathcal{E}_\sigma(q)$ and $\mathcal{S}_\sigma(q)$ depend on σ. For example, in Figure 6.4(a) (σ =XYXYXY), there are 3 query sections ($\mathcal{S}_\sigma(q)$) and 5 directed edges ($\mathcal{E}_\sigma(q)$) in q; in Figure 6.4(b) (σ =YXYXYX), there are 4 query sections and 4 directed edges in q. Both figures have $\mathcal{V}(q) = 8$ points in q. Equation 6.3 is key in computing the local cost (Section 6.3.2) of a query.

6.3 Efficient BMC Index Cost Estimation

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

$$\mathcal{C}_\sigma(q) = \mathcal{C}^g_\sigma(q) \cdot \mathcal{C}^l_\sigma(q) \tag{6.4}$$

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

Note also that the cost model of QUILTS [73] uses the product of a global and a local cost. However, its definitions of global cost and local cost are different from ours, as described in Section 2.4.

Next, we present efficient algorithms computing for the global and the local costs in Section 6.3.1 and Section 6.3.2, respectively.

6.3.1 Global BMC Index Query Cost

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

Definition 6.3 (Global Cost)**.** The global cost $\mathcal{C}^g_\sigma(q)$ of query q under BMC σ is the length of the curve segment from p_s to p_e:

$$\mathcal{C}^g_\sigma(q) = \mathcal{F}_\sigma(p_e) - \mathcal{F}_\sigma(p_s) + 1 = \sum_{j=1}^{d}\sum_{k=1}^{\ell}(\alpha^k_{e,j} - \alpha^k_{s,j}) \cdot 2^{\gamma^k_j} + 1 \tag{6.5}$$

Efficient computation. We rewrite the global cost as a closed-form function for efficient computation over a set Q of n queries.

$$\begin{aligned}\mathcal{C}^g_\sigma(Q) &= \sum_{i=1}^{n}\mathcal{C}^g_\sigma(q_i) = \sum_{i=1}^{n}\sum_{j=1}^{d}\sum_{k=1}^{\ell}\underbrace{(\alpha^k_{i,e,j} - \alpha^k_{i,s,j})}_{\text{BMC independent}} \cdot \underbrace{2^{\gamma^k_j}}_{\text{BMC dependent}} + n \\ &= \sum_{j=1}^{d}\sum_{k=1}^{\ell}\underbrace{\sum_{i=1}^{n}(\alpha^k_{i,e,j} - \alpha^k_{i,s,j})}_{\text{BMC independent}} \cdot 2^{\gamma^k_j} + n = \sum_{j=1}^{d}\sum_{k=1}^{\ell} A^k_j \cdot 2^{\gamma^k_j} + n\end{aligned} \tag{6.6}$$

Here, $q_i \in Q$; $\alpha^k_{i,s,j}$ and $\alpha^k_{i,e,j}$ denote the kth bits of the coordinates of the lower and the upper end points of q_i in dimension j, respectively; $A^k_j = \sum_{i=1}^{n}(\alpha^k_{i,e,j} - \alpha^k_{i,s,j})$, which is BMC independent and can be calculated once by scanning the n range queries in Q to compute the gap between p_e and p_s on the kth bit of the jth dimension, for any BMC. Only the term $2^{\gamma^k_j}$ is BMC dependent and must be calculated for each curve because γ^j_i represents the rank of the jth bit from dimension i of a BMC (cf. Section 6.2.1). If the BMC σ is changed, e.g., from XYXY**XY** to XYXY**YX**, then $\gamma^1_1 = 1$ and $\gamma^1_2 = 0$ are changed to $\gamma^1_1 = 0$ and $\gamma^1_2 = 1$, respectively.

Algorithm costs. The above property helps reduce the cost of computing the global cost when given multiple candidate BMCs. For example, when learning the best BMC from a set of candidate BMCs (cf. Section 6.4.2), each BMC is evaluated individually in each iteration (Algorithm 9). Without an efficient cost modeling, the global cost is $O(m \cdot n \cdot d \cdot \ell)$ for m candidate BMCs over n queries (based on Equation 6.5). Based on our proposed closed form method (Equation 6.6), after an initial $O(n)$-time scan over the n queries (to compute A^k_j), the holistic global cost over n queries can be calculated in $O(m \cdot d \cdot \ell)$ time, i.e., $O(m)$ time given constant number of dimensions d and number of bits ℓ in each dimension.

6.3.2 Local BMC Index Query Cost

The local cost measures the degree of segmentation of the curve in $[\mathcal{F}_\sigma(p_s), \mathcal{F}_\sigma(p_e)]$, which indicates the number of false positive data blocks that are retrieved unnecessarily and need to be filtered. We define the local cost as the number of query sections, which has been referred to as "*number of clusters*" in previous studies [68, 107].

Definition 6.4 (Local Cost)**.** The local cost $\mathcal{C}^l_\sigma(q)$ of query q under BMC σ is the number query sections in q, i.e., $\mathcal{S}_\sigma(q)$.

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

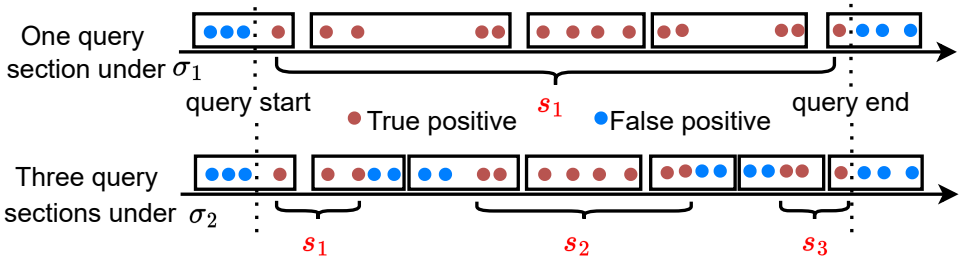


FIGURE 6.5: Query sections vs. block accesses

Example 6.3. *In Figure 6.5, we order points based on BMCs σ_1 and σ_2 and place the points in blocks where $B=4$. There are 14 true positives (i.e., $\mathcal{V}(q)=14$). There is only one query section under σ_1, which leads to a precision of $\frac{14}{5\times4}=70\%$ for 5 block accesses, whereas σ_2 has three query sections (due to a different curve). The number of block accesses is 7, and the precision drops to $\frac{14}{7\times4}=50\%$.*

Efficient computation. A simple way to compute the local cost of an arbitrary range query is to count the number of query sections by traversing the curve segment from p_s

to p_e, but this is also time-consuming. To reduce the cost, we rewrite Equation 6.3 as:

$$\mathcal{S}_\sigma(q) = \mathcal{V}(q) - \mathcal{E}_\sigma(q) \tag{6.7}$$

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

6.3.2.1 Rise and Drop Patterns

To compute $\mathcal{E}_\sigma(q)$ efficiently, we analyze how the bit sequence of a BMC changes from one point to another following a directed edge. A directed edge is formed by two consecutive points with (binary) curve values that share the same *prefix*, while the remaining bits are changed. We observe that different directed edges have the same shape when they share the same pattern in their changed bits, even if their prefixes are different. In Figure 6.6(a), consider edges $e_1 = (5, 6) = [\mathbf{0001}\underline{01}, \mathbf{0001}\underline{10}]$ and $e_2 = (13, 14) = [\mathbf{0011}\underline{01}, \mathbf{0011}\underline{10}]$. Both edges are in query q as indicated by the red rectangle, and they share the same '\' shape because the two rightmost bits in both cases change from "01" to "10". However, in Figure 6.6(a), edge $(1, 2) = [\mathbf{0000}\underline{01}, \mathbf{0000}\underline{10}]$ is not in q, and the prefix ("0000") differs from that of e_1 and e_2 above.

A query q can only contain directed edges of a few different shapes. In Figure 6.6(a), edge $(31, 32) = [\underline{011111}, \underline{100000}]$ is not in q, and the pattern of the changed bits differs from that of e_1 and e_2.

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

To generalize, recall that given a directed edge from p_i to p_j, $\mathcal{F}_\sigma(p_i) = \underbrace{...}_{prefix}\underline{0}\underbrace{\underline{1...1}}_{K\ 1\text{s}}$ and $\mathcal{F}_\sigma(p_j) = \underbrace{...}_{prefix}\underline{1}\underbrace{\underline{0...0}}_{K\ 0\text{s}}$ $(K \geq 0)$, i.e., must exist that are K rightmost bits changed from 1 to 0, while the $(K+1)$st rightmost bit is changed from 0 to 1. The bits of $\mathcal{F}_\sigma(p_i)$ and

$\mathcal{F}_\sigma(p_j)$ come from those of the column indices of p_i and p_j. Thus, the $K+1$ rightmost bits changed from $\mathcal{F}_\sigma(p_i)$ to $\mathcal{F}_\sigma(p_j)$ must also come from those of the column indices. In particular, there must be one dimension, where the column index has contributed k ($1 \leq k \leq K$) changed bits and one of the bits has changed from 0 to 1, while the rest dimensions contribute bits changing from 1 to 0.

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

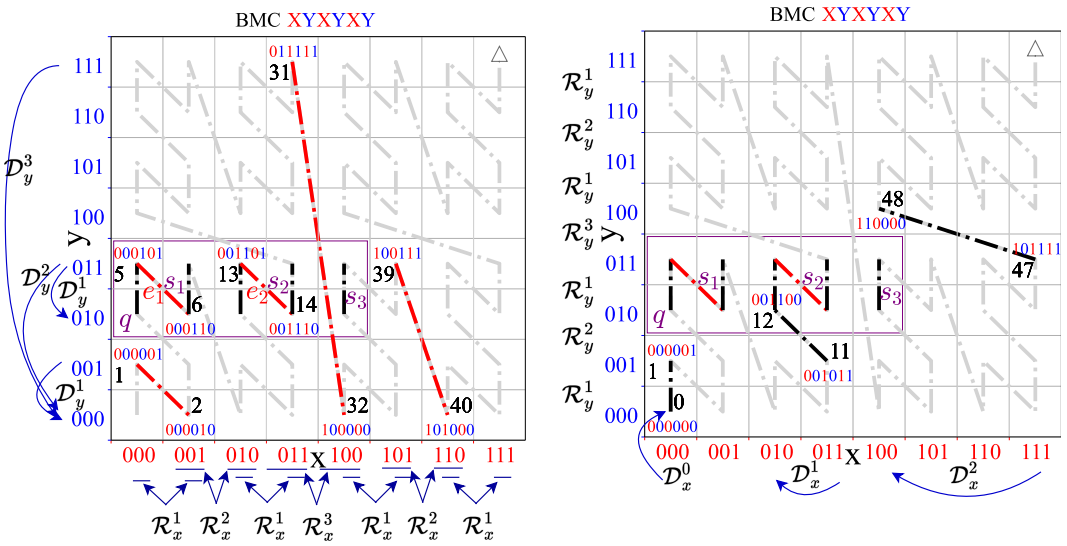
We summarize the bit-changing patterns to form a directed edge with two basic patterns: a *rise pattern* and a *drop pattern*.

Definition 6.5 (Rise Pattern)**.** A rise pattern $\mathcal{R}_b^k$ of a directed edge from p_i to p_j represents a k-bit ($k \geq 1$) change in the dimension-b coordinate of p_i (i.e., $x_{i,b}$) to that of p_j (i.e., $x_{j,b}$), where the rightmost $k-1$ bits are changed from 1 to 0 and the kth bit (from the right) is changed from 0 to 1, i.e., $x_{i,b} = \underbrace{...}_{prefix}\underline{0}\ \underbrace{1...1}_{(k-1)\ 1\text{s}}$ and $x_{j,b} = \underbrace{...}_{prefix}\underline{1}\ \underbrace{0...0}_{(k-1)\ 0\text{s}}$.

Definition 6.6 (Drop Pattern)**.** A drop pattern $\mathcal{D}_b^k$ of a directed edge from p_i to p_j represents a rightmost k-bit ($k \geq 0$) 1-to-0 change in the dimension-b coordinate of p_i (i.e., $x_{i,b}$) to that of p_j (i.e., $x_{j,b}$), i.e., $x_{i,b} = \underbrace{...}_{prefix}\underbrace{1...1}_{k\ 1\text{s}}$ and $x_{j,b} = \underbrace{...}_{prefix}\underbrace{0...0}_{k\ 0\text{s}}$.

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

Example 6.4. *In Figure 6.6(a), consider the directed edge from p_i to p_j, where $\mathcal{F}_\sigma(p_i) =$ 1 (00000) and $\mathcal{F}_\sigma(p_j) =$ 2 (00010), i.e., the '\' segment at the bottom left. The x-coordinate of p_i changes from 000 to 001 to that of p_j (i.e., rise pattern $\mathcal{R}_x^1$). The y-coordinate of p_i changes from 001 to 000 to that of p_j (i.e., drop pattern $\mathcal{D}_y^1$). Thus, this directed edge can be represented by a combination of $\mathcal{R}_x^1$ and $\mathcal{D}_y^1$, denoted as $\mathcal{R}_x^1 \oplus \mathcal{D}_y^1$. This same combination also applies in other directed edges, such as that from $\mathcal{F}_\sigma(p_i) = 13$ to $\mathcal{F}_\sigma(p_j) = 14$, which is another '\'-shaped segment. Other directed edges may use a different combination, e.g., $\mathcal{R}_x^3 \oplus \mathcal{D}_y^3$ for the one from $\mathcal{F}_\sigma(p_i) = 31$ to $\mathcal{F}_\sigma(p_j) = 32$, and $\mathcal{R}_x^2 \oplus \mathcal{D}_y^2$ for the one from $\mathcal{F}_\sigma(p_i) = 39$ to $\mathcal{F}_\sigma(p_j) = 40$.*



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

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

FIGURE 6.6: Example of forming a directed edge with rise and drop patterns: for BMC XYXYXY ($d = 2$ and $\ell = 3$), each directed edges is formulated by a rise and a drop pattern.

Figure 6.6(a) has shown the rise patterns $\mathcal{R}_x^k$ in dimension-x and the drop patterns $\mathcal{D}_y^k$ in dimension-y. Combining a rise and a drop pattern from these patterns forms a directed edge in red in the figure.

Similarly, we show in Figure 6.6(b) the rise patterns $\mathcal{R}_y^k$ in dimension-y and the drop patterns $\mathcal{D}_x^k$ in dimension-x. Combining a rise and a drop pattern from these patterns forms a black directed edge.

The *pattern combination operator* '$\oplus$' applied on two (rise or drop) patterns means that the $(K+1)$-bit change of a directed edge is formed by the two patterns.

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

for each dimension, there are only ℓ valid combinations of a rise and a drop pattern, and this number generalizes to $d \cdot \ell$ in a d-dimensional space given a BMC.

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

$$\mathcal{E}_\sigma(q) = \sum_{i=1}^{\ell} \Big(\mathcal{N}(\mathcal{R}_x^i) \cdot \mathcal{N}(\mathcal{D}_y^{r_y}) + \mathcal{N}(\mathcal{R}_y^i) \cdot \mathcal{N}(\mathcal{D}_x^{r_x}) \Big) \tag{6.8}$$

Here, $\mathcal{N}(\cdot)$ counts the number of times that a pattern occurs in q, and r_x (r_y) is a parameter depending on the drop patterns that can be combined with $\mathcal{R}_x^i$ ($\mathcal{R}_y^i$). In Figure 6.6, for $q = ([0,4] \times [2,3])$, there are two $\mathcal{R}_x^1$, one $\mathcal{R}_x^2$, and one $\mathcal{R}_x^3$, i.e., $\mathcal{N}(\mathcal{R}_x^1) = 2$, $\mathcal{N}(\mathcal{R}_x^2) = 1$, and $\mathcal{N}(\mathcal{R}_x^3) = 1$. Next, there is one $\mathcal{D}_y^1$, zero $\mathcal{D}_y^2$, and zero $\mathcal{D}_y^3$ that are valid to match with these rise patterns, i.e., $\mathcal{N}(\mathcal{D}_y^1) = 1$, $\mathcal{N}(\mathcal{D}_y^2) = 0$, and $\mathcal{N}(\mathcal{D}_y^3) = 0$. Similarly, $\mathcal{N}(\mathcal{R}_y^1) = 1$, and $\mathcal{R}_y^1$ can be matched with $\mathcal{D}_x^0$, where $\mathcal{N}(\mathcal{D}_x^0) = 5$. Recall that $\mathcal{D}_x^0$ is the special case with no bit value drop. It is counted as the length of the query range in dimension x. Overall, $\mathcal{E}_\sigma(q) = 2 \times 1 + 1 \times 5$. Thus, there are $10 - 7 = 3$ query sections in q according to Equation 6.7, which is consistent with the figure.

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

Similarly, a drop pattern $\mathcal{D}_b^k$ represents a change from $x_{i,b} = a \cdot 2^k + 2^k - 1$ to $x_{j,b} = a \cdot 2^k + 0$ ($a \geq 0 \wedge a \in \mathbb{N}$). Here, $a \cdot 2^k$ is the prefix, while $2^k - 1$ (i.e., $\underbrace{1 \ldots 1}_{k\ 1\text{s}}$) and 0 (i.e., $\underbrace{0 \ldots 0}_{k\ 0\text{s}}$) represent the changed bits. We can count each pattern by calculating $\lfloor (x_{e,b} + 1)/2^k \rfloor - \lceil x_{s,b}/2^k \rceil$, again in $O(1)$ time.

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

Definition 6.7 (Drop Pattern Collection)**.** For a directed edge in d-dimensional space, a drop pattern collection $\mathcal{D}^{k'}$ represents the bit combination over $d-1$ drop patterns: $\mathcal{D}^{\sum_{i=1,i\neq b}^{d-1} k_i} = \biguplus_{i=1,i\neq b}^{d} \mathcal{D}_i^{k_i}$ ($k' = \sum_{i=1,i\neq b}^{d} k_i = K - k$), where b is the dimension with a rise pattern. Here, '$\biguplus$' is a pattern combination operator (like $\oplus$ above). We note that $\mathcal{D}^{k'}$ and $\mathcal{D}_b^k$ are interchangeable if $d = 2$. For simplicity, we call $\mathcal{D}^{k'}$ a drop pattern when the context eliminates any ambiguity.

Now, in a d-dimensional data space, a directed edge can be formed by combining one rise pattern and $d-1$ drop patterns, i.e., $\mathcal{R}_b^k \oplus \mathcal{D}^{\sum_{i=1,i\neq b}^{d} k_i} = \mathcal{R}_b^k \oplus (\biguplus_{i=1,i\neq b}^{d} \mathcal{D}_i^{k_i})$ where $k' = \sum_{i=1,i\neq b}^{d} k_i$. Equation 6.8 is then rewritten as:

$$\mathcal{E}_\sigma(q) = \sum_{j=1}^{d} \sum_{i=1}^{\ell} \mathcal{N}(\mathcal{R}_j^i) \cdot \mathcal{N}(\mathcal{D}^r) \tag{6.9}$$

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

6.3.2.2 Pattern Tables

We have shown how to compute the local cost of a query efficiently. Given a set Q of n range queries ($q_i \in Q$), their total local cost based on Definition 6.4 is:

$$\mathcal{C}_\sigma^l(Q) = \sum_{i=1}^{n} \mathcal{C}_\sigma^l(q_i) = \sum_{i=1}^{n} \mathcal{V}(q_i) - \sum_{i=1}^{n} \mathcal{E}_\sigma(q_i) \tag{6.10}$$

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

We eliminate this bottleneck by a look-up table called a *pattern table* that stores pre-computed numbers of rise-and-drop pattern combinations to form the directed edges at different locations, which are BMC independent. Since each directed edge is a combination of a rise pattern in some dimension b and $d-1$ drop patterns, we proceed to show how to pre-compute d pattern tables, each recording the rise patterns of a dimension.

TABLE 6.2: Pattern table $Table^b$ for dimension b using ℓ bits on each dimension.

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

Definition 6.8 (Pattern Table)**.** The pattern table for dimension b, denoted by $Table^b$, contains ℓ rows, each corresponding to a rise pattern in the dimension, and $\ell\cdot(d-1)+1$ columns, each corresponding to a drop pattern in the other $d-1$ dimensions. As shown in Table 6.2, the value in row i and column j is the product of the numbers of rise pattern $\mathcal{R}_b^i$ and drop pattern $\mathcal{D}^j$.

There is a total of $\ell\cdot(d-1)+1$ drop patterns in the $d-1$ dimensions because there are $\ell\cdot(d-1)$ bits in those dimensions, i.e., $k'\in[0,\ell\cdot(d-1)]$ for $\mathcal{D}^{k'}$. Further, since the rise and drop patterns correspond to only the bit sequences in each dimension and not the curve values, the values in the pattern tables can be computed once given a set of queries Q and can then be reused across local cost estimation for different BMCs. Algorithm 6 summarizes the steps to compute pattern table $Table^b$ based on its definition.

Algorithm 6: Generate pattern table (GPT)

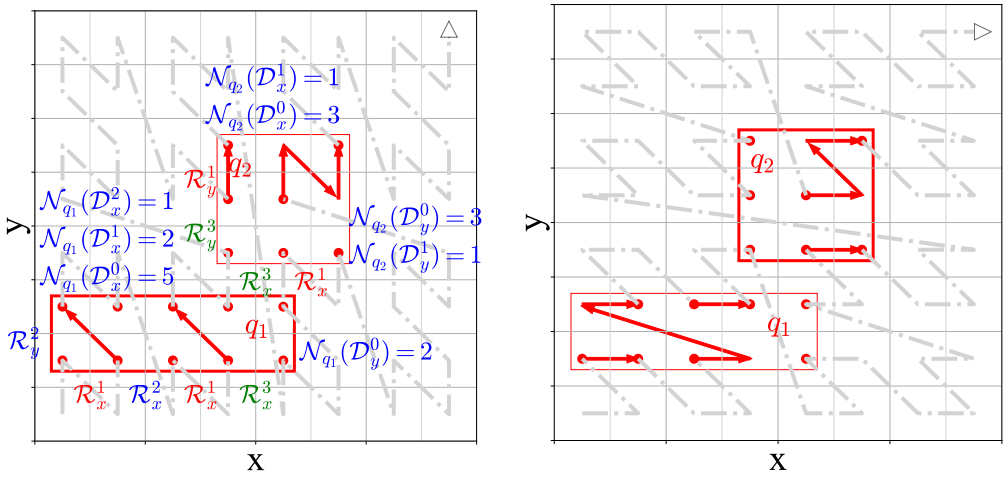
Input: Query set Q, target dimension b, data dimensionality d, number of bits per dimension ℓ

Output: Pattern table $Table^b$

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

Example 6.5. *In Figure 6.7(a), we show two queries q_1 and q_2, and the pattern tables $Table^x$ and $Table^y$ are shown in Tables 6.3 and 6.4, respectively. In the tables, we use '+' to denote summing up the pattern table cell values (i.e., $\mathcal{N}(\mathcal{R}_b^i)\cdot\mathcal{N}(\mathcal{D}^j)$, and $\mathcal{N}(\mathcal{D}^j)$ is $\mathcal{N}(\mathcal{D}_x^j)$ or $\mathcal{N}(\mathcal{D}_y^j)$) computed for q_1 and q_2. For example, in q_1, $\mathcal{N}(\mathcal{R}_x^1)=2$ (the two $\mathcal{R}_x^1$ are labeled for q_1 in Figure 6.7(a)) and $\mathcal{N}(\mathcal{D}_y^0)=2$ (the value range of*

q_1 in dimension y is 2). Meanwhile, in q_2, $\mathcal{N}(\mathcal{R}_x^1) = 1$ (one $\mathcal{R}_x^1$ is labeled for q_2 in Figure 6.7(a)) and $\mathcal{N}(\mathcal{D}_y^0) = 3$ (the value range of q_2 in dimension y is 3). Thus, in Table^x, the cell Table^x[1][0] (corresponding to $\mathcal{R}_x^1 \oplus \mathcal{D}_y^0$) is the sum of $\mathcal{N}(\mathcal{R}_x^1) \cdot \mathcal{N}(\mathcal{D}_y^0)$ in q_1 and q_2, i.e., $2 \times 2 + 3 \times 1$.



(A) Six directed edges (σ = XYXYXY)

(B) Nine directed edges (σ = YXYXYX)

FIGURE 6.7: Rise and drop pattern counting example ($d = 2, \ell = 3$). The results are shown in pattern tables in Tables 6.3 and 6.4.

TABLE 6.3: *Table^x*

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

TABLE 6.4: *Table^y*

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

6.3.2.3 Local Cost Estimation with Pattern Tables

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

Algorithm 7 shows how to compute the local cost using the pattern tables. Each dimension j is considered for the rise patterns (Line 2). Then, we consider each rise pattern in the dimension, i.e., each row i in *Table^j* (Line 3). We locate the corresponding drop pattern (i.e., the table column index) based on i and a given BMC σ, which is done by the `get_col` function (Line 4). Then, we add the cell value to the number of directed

Algorithm 7: Compute local cost with pattern tables

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

Output: Total local cost of n queries

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

edges $\mathcal{E}_\sigma$ (Line 5). Note that all ℓ rise patterns in each dimension are considered because a BMC has ℓ bits on each dimension, which can all be the bit that changes from 0 to 1. We return the total local cost by subtracting the total number of directed edges from the total number of cells in Q.

Example 6.6. *Based on Example 6.5, given BMC XYXYXY, from* $Table^x$*, we read cells* ($\mathcal{R}_1^1$, $\mathcal{D}_2^1$), ($\mathcal{R}_1^2$, $\mathcal{D}_2^2$), *and* ($\mathcal{R}_1^3$, $\mathcal{D}_2^3$), *i.e., the cells with "wavy" lines. Similarly, we read the cells with "wavy" lines from* $Table^y$*. These cells sum up to 6, which is the number of directed edges (segments with arrows) in Figure 6.7(a). Similarly, the cells relevant to BMC YXYXYX are underlined, which yields a total of nine directed edges in Figure 6.7(b).*

Algorithm costs. In general, for each rise pattern, the total number of possible drop pattern combinations is $(\ell + 1)^{d-1}$ based on Definition 6.7. The time complexity of generating the d pattern tables is $O(d \cdot \ell \cdot (\ell + 1)^{d-1})$, where d denotes the number of dimensions, ℓ denotes the number of rows, and $(\ell + 1)^{d-1}$ denotes the accumulated number of drop patterns (equal to $(\ell+1)$ when $d = 2$). After initialization, the retrieval time for the pattern tables is $O(d \cdot \ell) = O(1)$, i.e., we retrieve ℓ cells from each table.

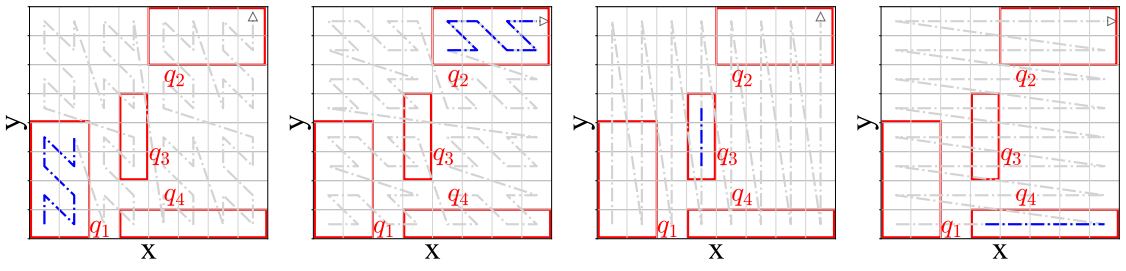
We generate d pattern tables, each with $\ell \cdot (\ell + 1)^{d-1}$ keys. Thus, the space complexity for the pattern tables is $O(d \cdot \ell \cdot (\ell + 1)^{d-1})$. For example, when $d = 3$ and $\ell = 32$, all the tables take 1.6 MB (1.2 MB for keys and 0.4 MB for values).

6.4 Query Optimization with Efficient Cost Estimation

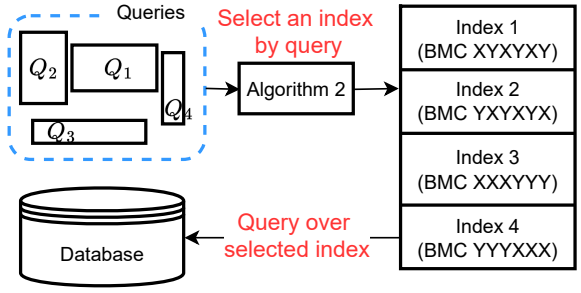
We show how our cost estimation methods can be used to optimize query efficiency in database systems, in both loosely and tightly coupled manners in Section 6.4.1 and Section 6.4.2, respectively.

6.4.1 Index Selection Based on Cost Estimation

A direct application of our cost estimation methods is to support database systems in selecting a query-efficient BMC index at query time. For example, two specific BMCs, ZC and LC, are implemented in Hudi [4] and SQL Server [64], and by taking different orders of the dimensions (i.e., index key columns), such as XYXYXY, YXYXYX, XXXYYY, or YYYXXX (cf. Figures 6.8(a) to 6.8(d)), multiple indices can be built using ZC and LC. This offers an opportunity to create multiple BMC indices on a data table and to select the most efficient index for a query. In Figure 6.8, the four queries q_1, q_2, q_3, and q_4 contain only one curve segment for the curves in Figures 6.8(a), 6.8(b), 6.8(c), and 6.8(d), which should be selected for query execution, respectively. Our cost modeling methods enables us to select such index efficiently.



(A) BMC XYXYXY (B) BMC YXYXYX (C) BMC XXXYYY (D) BMC YYYXXX



(E) Overview of BMC selection.

	XYXYXY		YXYXYX		XXXYYY		YYYXXX	
	c_g	c_l	c_g	c_l	c_g	c_l	c_g	c_l
Q_1	10	2	**7**	**2**	18	3	11	2
Q_2	**8**	**1**	12	2	12	2	26	4
Q_3	33	5	17	3	33	5	**5**	**1**
Q_4	13	2	25	3	**3**	**1**	17	3

(F) Costs of queries over BMCs.

FIGURE 6.8: A BMC index selection example.

Given a query q, our index selection algorithm, named SBMC, is summarized in Algorithm 8. Without loss of generality, consider a list L of pairs of BMC σ and corresponding index I. The cost of each BMC index for q is calculated using our global and local cost estimation methods described in the last section (the $\texttt{cost}(\cdot)$ function at Line 4). The BMC index with minimal cost is returned for query execution.

Algorithm 8: Select BMC index (SBMC)

Input: A list of BMC and index pairs L, query q
Output: The optimal index for q

```
1 I_opt ← NULL;
2 for ⟨σ, I⟩ ∈ L do
3     if I_opt = NULL or cost_opt > cost(σ, q) then
4         cost_opt ← cost(σ, q);
5         I_opt ← I;
6 return I_opt;
```

Discussion. Our index selection algorithm only adds $O(|L|)$ time to query processing given $|L|$ indices ($|L|$ is a small constant) as our cost estimation takes only $O(1)$ time. Importantly, *it can be used without any change to an existing database system*, i.e., we can modify an SQL query q to add the index to be used for q, such as:

```
SELECT * FROM lineitem USE INDEX(I_opt)
    WHERE discount BETWEEN 0.1 and 0.2
    AND shipdate BETWEEN "1995-01-01" and "1996-12-31";
```

6.4.2 BMC Learning Based on Cost Estimation

Next, we consider an advanced optimization scenario, where a dataset D and a query set Q are given, and there is no constraint on the BMCs. Our aim is to find the optimal BMC σ_{opt} that minimizes the costs of Q. While using BMCs reduces the number of curve candidates from $(2^\ell)^d!$ to $\frac{(d\cdot\ell)!}{(\ell!)^d}$ (Section 6.1), it is still non-trivial to find the optimal BMC from the $\frac{(d\cdot\ell)!}{(\ell!)^d}$ candidates. We present an efficient learning-based algorithm named *LBMC* for this search.

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

To reduce the search space, we impose two constraints on the bit swaps: (a) we only swap two adjacent bits each time, and (b) two bits from the same dimension cannot be swapped (which guarantees valid BMCs after swaps, cf. Section 6.2.1). Any bit then takes at most $(d-1)\cdot\ell$ swaps to reach its optimal position, when such a position is known. Given $d\cdot\ell$ bits, at most $d\cdot(d-1)\cdot\ell^2$ swaps are needed to achieve the optimal BMC guided by an oracle.

In practice, an ideal oracle is unavailable. Now the problem becomes how to run the bit swaps without an ideal oracle. There are two approaches: (a) run a random swap (i.e., *exploration*) each time and keep the result if it reduces the query cost, and (b) select a position that leads to the largest query cost reduction each time (i.e., *exploitation*). Using either approach yields local optima. We integrate both approaches by leveraging Deep Reinforcement Learning (DRL) to approach a global optimum, since DRL aims to maximize a long-term objective [63] and balance exploration and exploitation.

BMC learning formulation. We formulate BMC learning as a DRL problem:

1. State space $\mathcal{S}$, where a state (i.e., a BMC) $\sigma_t \in \mathcal{S}$ at time step t is a vector $\langle\sigma_t[d\cdot\ell], \sigma_t[d\cdot\ell-1], \ldots, \sigma_t[1]\rangle$, and $\sigma_t[i]$ is the ith bit. For example, if σ_t =XYZ, $\sigma_t[3]$=X, $\sigma_t[2]$=Y, and $\sigma_t[1]$=Z.

2. Encoding function $\phi(\cdot)$, which is used to encode a BMC to fit the model input. We use one-hot encoding. For example, X, Y, and Z can be encoded into $[0,0,1]$, $[0,1,0]$, and $[1,0,0]$, respectively, and XYZ by $[0,0,1,0,1,0,1,0,0]$.

3. Action space $\mathcal{A}$, where an action $a \in \mathcal{A}$ is the position of a bit to swap. When the ath bit is chosen, we swap it with the ath bit (if $a+1 \leq d\cdot\ell$). Thus, $\mathcal{A} = \{a \in \mathbb{Z} : 1 \leq a \leq d\cdot\ell - 1\}$.

4. The reward, r: $\mathcal{S}\times\mathcal{A}\times\mathcal{S}\rightarrow r$, is the query cost reduction when reaching a new BMC σ_{t+1} from σ_t. Since an oracle is unavailable, we use our cost model to estimate the query cost of a BMC. The reward r_t at step t is calculated as $r_t = (\mathcal{C}_{\sigma_t} - \mathcal{C}_{\sigma_{t+1}})/\mathcal{C}_{\sigma_1}$, where $\mathcal{C}_{\sigma_t} = \mathcal{C}^g_{\sigma_t}(Q)\cdot\mathcal{C}^l_{\sigma_t}(Q)$ is the cost of σ_t estimated by Equations 6.6 and 6.10.

5. Parameter ϵ, which is used to balance exploration and exploitation in the learning process to avoid local optima.

Based on this formulation, we use *deep Q-learning* [67] in our LBMC algorithm to learn a query-efficient BMC index.

The LBMC algorithm. We summarize LBMC in Algorithm 9 where the input σ_1 can be any initial BMC, e.g., a ZC. The key idea of LBMC is to learn a policy $\pi : \mathcal{S} \to \mathcal{A}$ that guides the position selection for a bit swap given a status, to maximize a value function $\mathtt{Q}^*(\phi(\sigma_t), a)$ (i.e., the reward) at each step t. Such a policy π can be learned by training a model (a *deep Q-network*, DQN) with parameters θ over existing "*experience*" (previously observed state transitions and their rewards), which is used to predict the position a to maximize the value function (i.e., $\max_a \mathtt{Q}^*(\phi(\sigma_t), a; \theta)$). After a number of iterations, the learned BMC σ^*_{opt} is expected to approach σ_{opt}, which is returned as the algorithm output.

We initialize a storage MQ to store the latest N_{MQ} bit-swapping records (i.e., the experience, Line 1). We learn to approach σ_{opt} with M episodes and T steps per episode (Lines 2 and 3). In each episode, we start with σ_1 encoded by $\phi(\cdot)$. To select a swap position a_t at step t, we generate a random number in $[0, 1]$, if the number is greater than ϵ, we randomly select a position a_t, otherwise, we set a_t as the position with the highest probability to obtain a maximal reward, i.e., $\max_a \mathtt{Q}^*(\phi(\sigma_t), a; \theta)$ (Line 4). The prediction is based on the current state σ_t and model weights θ. We execute a_t ($\mathtt{E}(\sigma_t, a_t)$ at Line 5) and compute reward r_t using our cost model (Line 6). We record the new transition in MQ and train the DQN (i.e., update θ) over sampled data in MQ (Lines 7 and 8). The training uses gradient descent to minimize a loss function $L_t(\theta_t) = \mathbb{E}_{\phi(\sigma), a\sim\rho(\cdot)}\left[(y_t - Q(\phi(\sigma), a; \theta_t))^2\right]$ where y_t is the target from iteration t and $\rho(\cdot)$ is the action distribution [67]. We use σ^*_{opt} to record the new BMC from each swap (Line 9), which is returned in the end (Line 10).

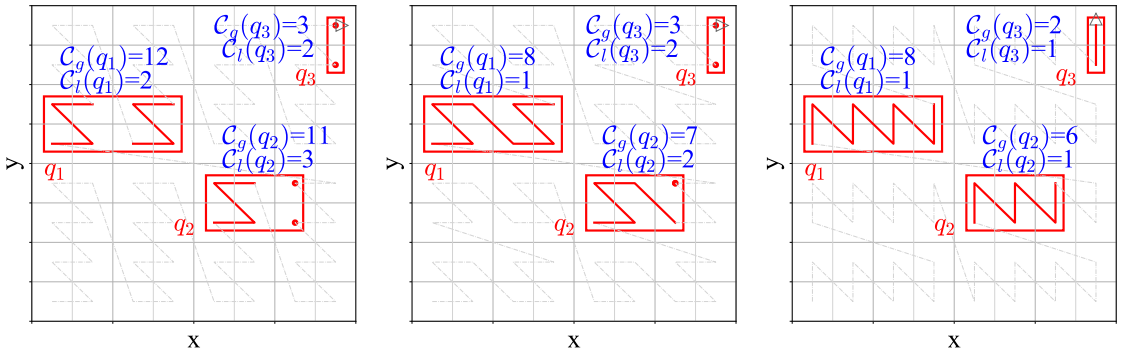
Example 6.7. *Figure 6.9 illustrates LBMC with $\ell = 3$ and three queries q_1, q_2, and q_3. The initial BMC $\sigma_1 = YX\underset{\sim}{X}\underline{Y}YX$ has an (estimated) query cost of $\mathcal{C}_1 = 175$ (Figure 6.9(a)). We select position $a_1 = 3$ and swap the 3rd and the 4th bits to get $\sigma_2 = YX\underline{Y}\underset{\sim}{X}\underline{Y}\underline{X}$ such that the cost is decreased to $\mathcal{C}_2 = 90$ (Figure 6.9(b)). Next, we select position $a_2 = 1$ and swap the 1st and the 2nd bits to get $\sigma_3 = YXYX\underline{\underline{X}}Y$ with cost $\mathcal{C}_3 = 48$ (Figure 6.9(c)). We store all the intermediate results into memory MQ for learning the DQN model in Figure 6.9(d), where we show the BMCs without encoding. Figure 6.9(e) shows the cost ratios, i.e., $\mathcal{C}_t/\mathcal{C}_1$, which decrease as t increases (Figures 6.9(a) to 6.9(c) are three of the steps). The learned BMC approaches the optimum in this process.*

Algorithm 9: Learn BMC (LBMC)

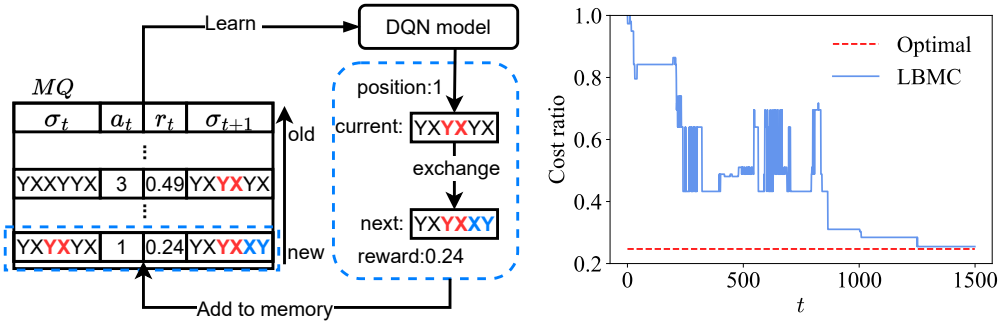
Input: Initial BMC σ_1

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

```
Initialize replay memory MQ with capacity N_MQ;
for episode ∈ [1, M] do
    for t ∈ [1, T] do
        With probability ϵ select a random position a_t, or a_t ← max_a Q*(φ(σ_t), a; θ);
        σ_{t+1} ← E(σ_t, a_t);
        Compute reward r_t;
        Store transition (φ(σ_t), a_t, r_t, φ(σ_{t+1})) in MQ;
        Train model θ with sampled transitions from MQ;
        σ*_opt ← σ_{t+1};
return σ*_opt;
```



(A) YXXYYX, $\mathcal{C}_1 = 175$ (B) YXYXYX, $\mathcal{C}_2 = 90$ (C) YXYXXY, $\mathcal{C}_3 = 48$



(D) Learning through LBMC (E) Cost ratio vs. number of steps

FIGURE 6.9: A BMC learning example.

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

6.5 Experiments

We answer three questions with an empirical study:

- **Q1:** Are global and local cost estimation efficient?
- **Q2:** Are the costs effective and efficient at supporting index selection?
- **Q3:** Are the costs effective and efficient at supporting BMC learning?

6.5.1 Experimental Setting

We design an evaluation framework to compare different indices which are implemented in C/C++. The indexed data points are stored in blocks with a capacity of 100, i.e., $B = 100$. Only our BMC learning algorithm LBMC is implemented in Python to use the deep learning package TensorFlow. We run experiments on a desktop computer running 64-bit Ubuntu 20.04 with a 3.60 GHz Intel i9 CPU, 64 GB RAM, and a 500 GB SSD.

For the SFCs, the number of bits ℓ is a key parameter as it impacts the curve value mapping. By default, we set $\ell = 16$ to balance the costs of computing the curve values and the cost estimation.

Datasets. We use three real datasets: **OSM** [76], **NYC** [98], and **TPC-H** [99]. OSM contains 100 million 2-dimensional location points (2.2 GB). NYC contains some 150 million yellow taxis transactions in 2015 (8.4 GB). After cleaning incomplete records, some 143 million records remain. Following QUILTS [73], we use this dataset to simulate a spatio-temporal analysis application where each record has three dimensions (latitude, longitude, and pick-up time). TPC-H includes three dimensions (quantity, discount, and ship date), which are frequently queried from the data table `lineitem` (5.8 GB), with 120 million records (scale factor 20).

Queries. OSM and NYC contain data records without query workloads. We generate 1,000 synthetic queries for each *range query template* as a query workload. For OSM, a range query template is defined by (1) a *size ratio* δ, i.e., the initial query span relative to the data domain in each dimension, and (2) an *aspect ratio pattern* (e.g., 1×4), i.e., the relative query span in the two dimensions, which is multiplied with δ in each dimension to form the final query size (e.g., $1\delta \times 4\delta$). For NYC, a range query template

is represented by (ω, τ), where ω is the area of the location dimensions (i.e., latitude and longitude) and τ is the range of time dimension (i.e., hour, day, and month). TPC-H comes with query templates, five of which relate to table `lineitem`: `sql1`, `sql3`, `sql6`, `sql7`, and `sql17`. We also generate 1,000 queries for each template as a workload.

To simulate real applications, for each dataset, we mix workloads from different query templates and randomly select 1,000 range queries to form a **mixed query workload**.

Evaluation metrics. The core evaluation metrics used are: (1) the **cost estimation time** of different algorithms, (2) the **query time** and (3) the **block access ratio** of different indices, and (4) the **index build time** of the indices. Here, the block access ratio refers to the number of block accesses of a query q over an index relative to the optimal number of block accesses for q. This ratio is reported following previous studies [5, 41], because the raw numbers of block assesses of different queries can vary widely. The query performance over a workload may be dominated by queries with large sizes (requiring many block accesses).

Note that, unlike in previous chapters, for simplicity, we do not report results with varying query execution order for the cost estimation and query processing experiments. For cost estimation, the query execution order does not impact the time cost. For query processing, the block access ratio and query time remain stable across runs with different query execution order.

We summarize the parameters tested in Table 6.5 where the default values are in boldface. We test ℓ up to 18 because the computation of a baseline naive local cost becomes prohibitively expensive.

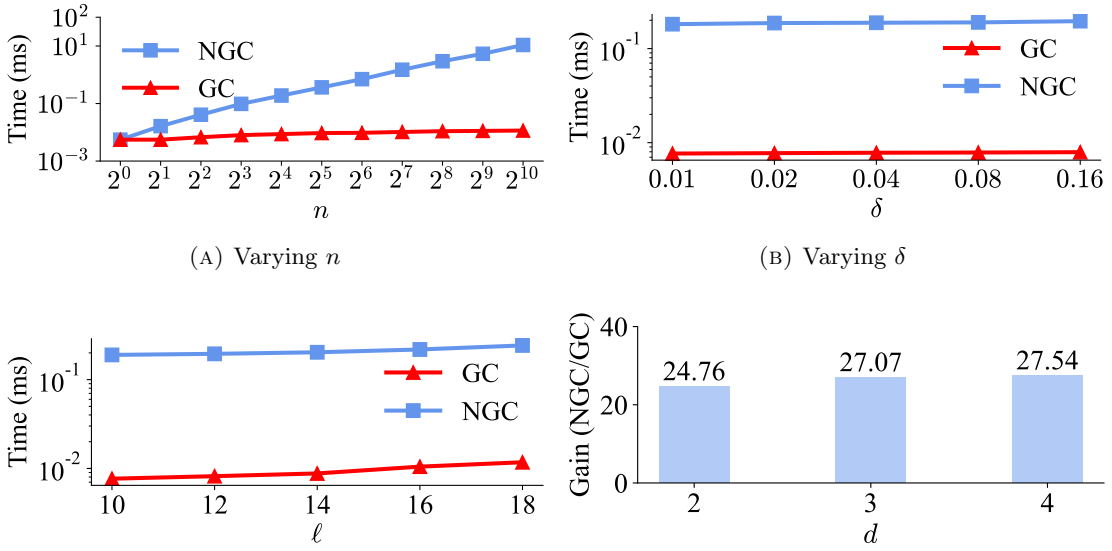
Table 6.5: Parameter settings.

Experiments	Parameter	Setting
Cost estimation efficiency (Q1)	n	2^0, 2^1, 2^2, 2^3, $\mathbf{2^4}$, 2^5, 2^6, 2^7, 2^8, 2^9, 2^{10}
	δ	**0.01**, 0.02, 0.04, 0.08, 0.16
	ℓ	**10**, 12, 14, 16, 18
	d	**2**, 3, 4
Query efficiency (Q2 & Q3)	$\delta(\times 10^{-4})$	**1**, 2, 4, 8, 16
	aspect ratio	1×1, 1×4, $\mathbf{1 \times 16}$, 1×64, 1×256
	τ	an hour, a day, **a week**, a month, a quarter

6.5.2 Efficiency of Cost Estimation (Q1)

We first evaluate the efficiency of our algorithms (excluding initialization) to compute the global cost **GC** and the local cost **LC** (Algorithm7), which are based on Equations 6.6 and 6.9, respectively. We use **IGC** and **ILC** to denote our initialization steps of the two costs, respectively. As there are no efficient algorithms to compute these costs, we compare with baseline algorithms based on Equations 6.5 and 6.10, denoted by **NGC** and **NLC**, respectively.

We vary the number of queries n, the query size (via δ), and the number of bits ℓ. We run experiments for 2- to 4-dimensional spaces. For succinctness, we focus on the 2-dimensional space (the comparative results are similar for $d \in \{3, 4\}$). As the cost estimation is data independent, a dataset is not needed to study their efficiency. The queries are generated at random locations with a square shape and without query templates.



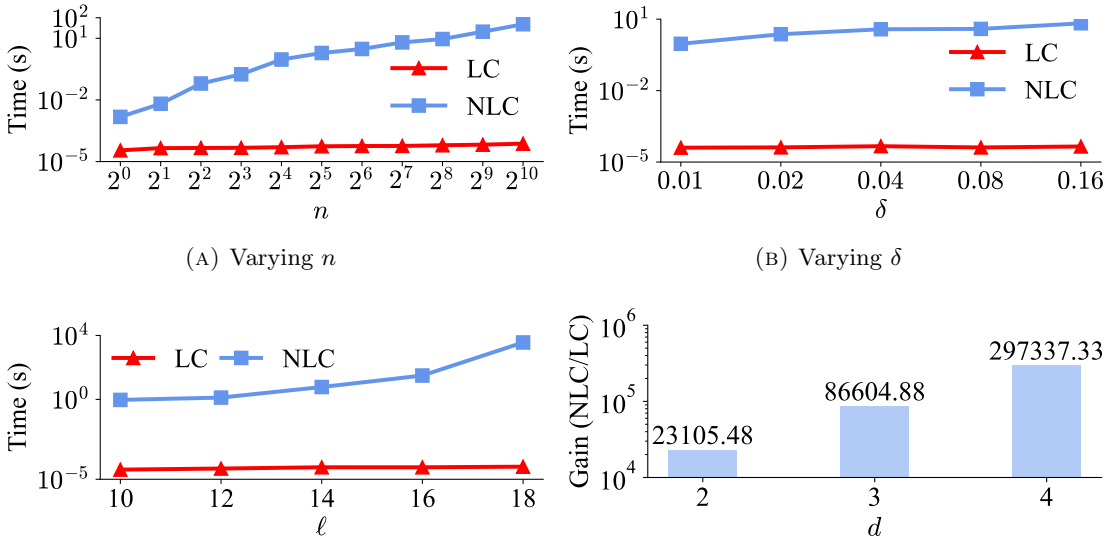
(A) Varying n (B) Varying δ

(C) Varying ℓ (D) Varying d

FIGURE 6.10: Running times of global cost estimation.

Efficiency of GC. Figures 6.10(a) and 6.10(b) show the impact of n and δ, respectively. Since GC takes $O(d \cdot \ell)$ time to compute (after the initialization step), its running time is not impacted by n and δ. NGC takes $O(n \cdot d \cdot \ell)$ time. Its running time grows linearly with n and is not impacted by δ as shown in the figures. Figure 6.10(c) shows that the running times of GC and NGC both increase with ℓ, which are consistent with their time complexities. Since the relative performance of our algorithm and the baseline is stable when ℓ is varied, we use a default value of 10 instead of 16 as mentioned earlier,

to streamline this set of experiments. Figure 6.10(d) further shows the impact of d. Here, we show the performance gain (i.e., the running time of NGC over that of GC) instead of the raw running times, which are of different scales when d varies such that it is difficult to observe the relative performance. We see that GC is faster than NGC for more than 24 times. Overall, GC is faster than NGC consistently, with up to over an order of magnitude performance gain, which confirms the high efficiency of GC.



(A) Varying n (B) Varying δ

(C) Varying ℓ (D) Varying d

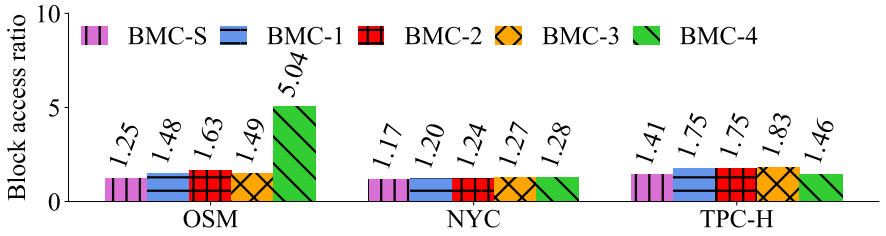
FIGURE 6.11: Running times of local cost estimation.

Efficiency of LC. Figures 6.11(a) to 6.11(d) show the running times for computing the local costs. The performance patterns of LC and NLC are similar to those observed above for GC and NGC, and they are consistent with the cost analysis in Section 6.3.2. The performance gains of LC are even larger, for that its pre-computed pattern table enables extremely fast local cost estimation. As Figure 6.11(d) shows, LC outperforms NLC by over 10^5 times when $d = 4$.

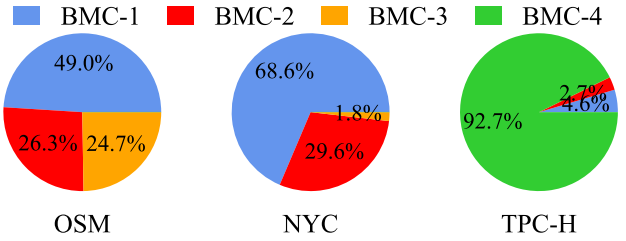
TABLE 6.6: Initialization costs of GC and LC (varying n).

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

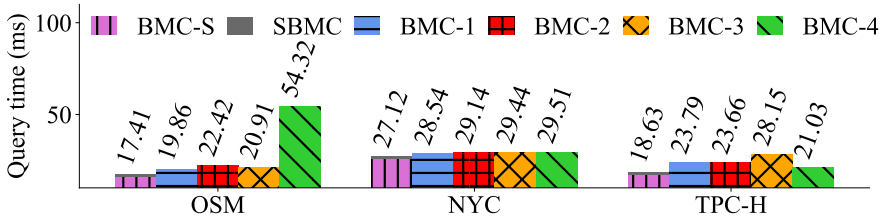
Initialization costs of GC and LC. Table 6.6 further shows the running times of IGC and ILC, which increase with n, because the initialization steps need to visit all range



(A) Average block access ratio



(B) Index selection ratio



(C) Average query time (SBMC is above BMC-S)

FIGURE 6.12: Query efficiency with and without index selection.

queries to compute a partial global cost and prepare the pattern tables, respectively. These running times are smaller than those of NGC and NLC further confirming the efficiency of our proposed cost computation algorithms. Similar patterns are observed when varying δ, ℓ, and d, which are omitted for succinctness. We do not report the result when $n = 2^0$ (i.e., $n = 1$) as an initialization is not needed for a single query.

6.5.3 Query Efficiency with Index Selection (Q2)

This set of experiments simulates an index selection scenario for query processing, where a database system supports ZC and LC.

Four BMC indices, denoted by **BMC-1**, **BMC-2**, **BMC-3**, **BMC-4**, are built on each of the OSM, NYC, and TPC-H datasets. Query processing with an index selected from these indices using our Algorithm 8 is denoted by **BMC-S**.

For OSM ($d = 2$), BMC-1 and BMC-2 are ZCs where the leading bits of the curve values come from the x- and the y-dimensions, while BM3-3 and BMC-4 are LCs that first go through the x and the y dimensions, respectively. For NYC ($d = 3$), all four indices use ZCs, because the latitude and the longitude dimensions are correlated much more closely than the pick-up time dimension, such that LC is sub-optimal empirically and will not be selected for query processing. The four ZCs used here interleave bits with repeated patterns of ZXY, XZY, YZX, and ZYX, respectively. For TPC-H ($d = 3$), BMC-1 and BMC-2 use ZCs that interleave bits with repeated patterns of XYZ and ZYX, respectively. BMC-3 and BMC-4 use LCs that go through the dimensions in the order of X, Y, and Z, and the order of Z, Y, and X, respectively.

We run the mixed query workload (cf. Section 6.5.1), to show the effectiveness of BMC-S in index selection for different queries.

Block access ratio. As Figure 6.12(a) shows, BMC-S has consistently lower average block access ratios than those of any of the BMCs, with an advantage of up to 75.2% (1.25 vs. 5.04 on OSM). Different curves may perform differently on different datasets, e.g., BMC-4 (LCs) is the least efficient for OSM and the most efficient for TPC-H because of different degrees of correlation among the dimensions and different query ranges over different dimensions. Accordingly, BMC-S can identify efficient indices for different queries. Figure 6.12(b) shows the ratios of different BMCs being selected by BMC-S. BMC-4 is most often selected for TPC-H and is not selected for OSM at all. These confirm the effectiveness of our cost model-based index selection algorithm.

Query time. Figure 6.12(c) shows the query times, where **SBMC** denotes the extra time incurred by our index selection step. We see that index selection incurs negligible overheads (i.e., about 0.1 ms), and BMC-S also yields the lowest query times consistently.

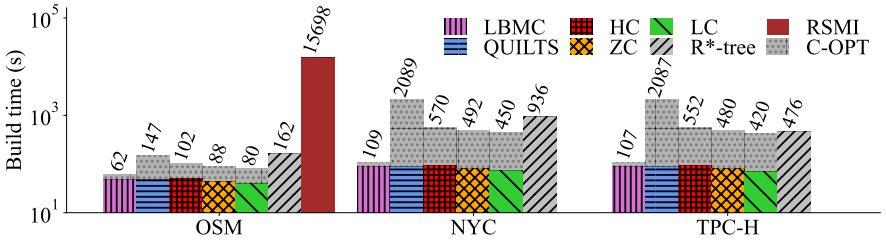
6.5.4 Query Efficiency with BMC Learning (Q3)

We now test the BMC learning efficiency of **LBMC** and the query efficiency of the index built using the learned BMCs.

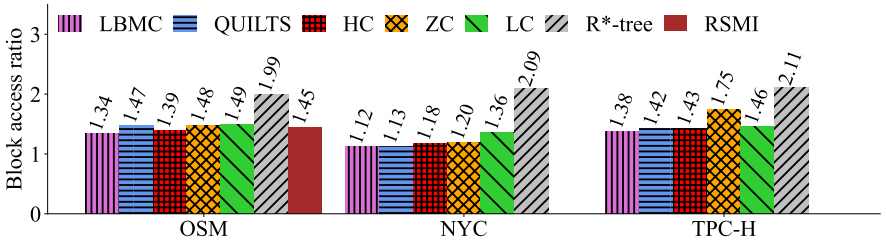
Competitors. We compare with four indices based on SFCs. (1) **QUILTS** [73] orders data points by a BMC derived by a curve design method described in Section 2.4. We implement it following its paper as its source code is unavailable. (2) **ZC** [77] orders

data points by their Z-curve values. (3) **HC** [47] orders data points by their Hilbert curve values. (4) **LC** orders data points by their dimension values. For ZC, HC, and LC, we test different curve variants using different dimension ordering as described in the last subsection, and we report results of the best variant. For all these methods including our LBMC, we bulk-load a B^+-tree based on curve ordering. We store the *minimum bounding rectangle* (MBR) with each tree node, making the tree effectively an R-tree. Queries are then processed by the standard R-tree query algorithms to exploit the pruning capability of the MBRs. Note that while the cost models are designed for B^+-trees, they also implicitly reflect the pruning power of R-trees, as fewer and longer curve segments in a query q implies fewer R-tree nodes intersecting q.

We further compare with a traditional and a recent learned spatial index, i.e., (5) **R*-tree** [10] and (6) **RSMI** [84]. We use the released code of the R*-tree and the implementation of RSMI from Chapter 3. Other recent learned indices either have no released code (e.g., Qd-tree [108]) or is implemented in Python (e.g., LISA [58]), which are not comparable in this chapter.



(A) Index build time



(B) Average block access ratio

FIGURE 6.13: Performance of LBMC (mixed query workloads).

Overall results. We report the build time and query costs of the indices on the three datasets, based on the mixed query workload which may be a more practical workload.

Index build time. Figure 6.13(a) reports the index build time results, where "**C-OPT**" denotes the extra curve optimization (e.g., curve learning for LBMC) for the SFC indices

LBMC, QUILTS, HC, ZC, and LC. We see that LBMC is the fastest to build, even with a seemingly expensive DRL-based curve learning step. It takes just about about 5 s on OSM and 10 s on NYC and TPC-H for the initialization of cost estimation and another 10 s for the BMC learning on each dataset. In comparison, QUILTS has a much more expensive curve design procedure. It computes the entropy of the ratio of the interval lengths between query sections, which requires to compute the length of all query sections leading to a much higher curve optimization cost (about 100 s on OSM and 2,000 s on NYC and TPC-H). Since QUILTS designs the bit distributions of a curve based on a given query shape, a total of 20 curves are considered in QUILTS for a a mixed workload. For HC, ZC, and LC, we enumerate all dimension orderings and use the variant that has the minimum *actual* average block access ratio for query processing. The total build times of all variants are reported.

R*-tree and RSMI have no curve optimization. However, R*-tree is still slower than LBMC for its dynamic point insertion, and RSMI is expensive to build as it needs to train neural networks to predict point partitions. Since our RSMI implementation currently supports two-dimensional data only, no results are reported for NYC and TPC-H for $d = 3$. We omit the index build times for the later experiments as the relative performance patterns remain similar.

Query costs. Figure 6.13(b) reports the average block access ratio when the indices built are used for query processing. Since it has been shown earlier that the query times follow a similar pattern to that of the block access ratios, we omit the query times for simplicity. LBMC consistently outperforms the state-of-the-art (SOTA) BMC design method, QUILTS, on all three datasets. LBMC is also the most query-efficient index on all datasets, confirming the effectiveness of LBMC to find query-efficient BMCs. ZC and LC (worst on NYC as discussed earlier) are not as competitive, while R*-tree lacks efficiency because its data blocks are not full.

Results on OSM. We show the impact of query templates on each dataset, starting from OSM. We vary the size ratio δ from 10^{-4} to 16×10^{-4} and the aspect ratio pattern from 1×1 to 1×256.

Varying δ. We first vary the size ratio δ in Figure 6.14. Since OSM is 2-dimensional, when δ increases linearly, the query range size has a quadratic growth. We see that the block access ratios of all indices decrease with δ, which is reasonable. This is because

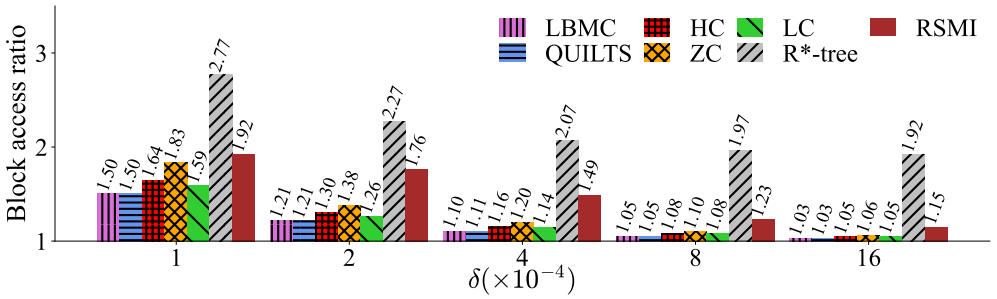


FIGURE 6.14: Varying δ (aspect ratio pattern $= 1 \times 16$)

when query range size grows, the minimum number of block accesses needed outgrows that of the number of false positive blocks accessed. Here, LBMC shows slightly better or equivalent performance than QUILTS, while it can be more efficiently derived (cf. Figure 6.13(a)). LBMC is also consistently better than other baselines.

Varying aspect ratio. Figure 6.15 shows that LBMC again consistently outperforms (or performs as well as) the SOTA method QUILTS as well ZC and LC, because the curve search space of LBMC is a superset of those of ZC and LC. As QUILTS only considers the shape of queries but not their locations, its curve may not be optimized for queries of a known shape that are located at different positions of the data space. HC is the most efficient on the more square-shaped patterns 1×1 and 1×4, while LBMC outperforms it on the skewed patterns 1×16, 1×64, and 1×256. Note that HC is known to be highly efficient for query processing in average cases (e.g., in the mixed-query settings) [5, 43].

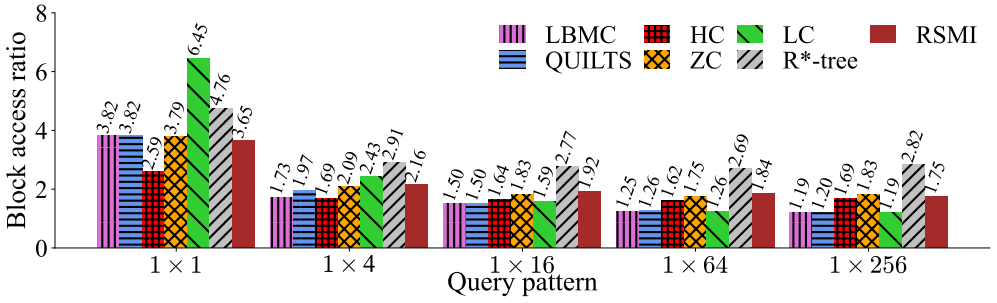


FIGURE 6.15: Varying aspect ratio pattern (OSM, $\delta = 10^{-4}$)

Results on NYC. For NYC, we vary τ (default at a week) for the time dimension, where the default ω is (10^{-6} with pattern 1×16) of the two locations dimensions. We omit to vary ω because it yields a similar result pattern as in Figure 6.14.

Varying τ. Figure 6.16 shows that the block access ratios decrease when longer time spans (i.e., larger τ) are queried. This can be explained by that a larger query range

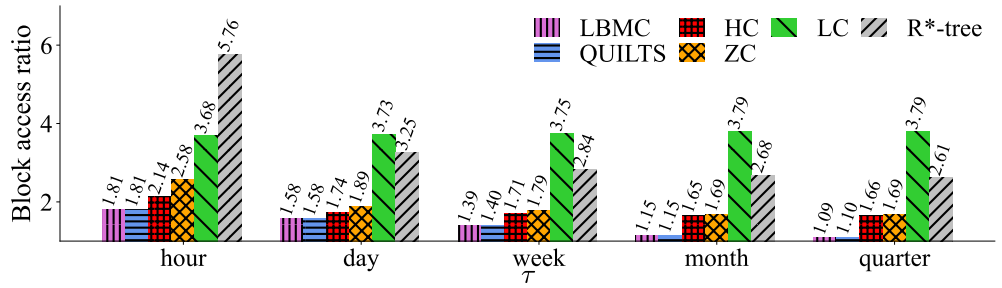


FIGURE 6.16: Varying τ (NYC, $\omega = 10^{-6}$)

includes more data points such that the related scale of extra points (and blocks) accessed decreases. LBMC and QUILTS both outperform HC, which is consistent with the results above, since a week is about 2×10^{-2} of the time dimension on NYC which is larger than that of the query length of the location dimensions, i.e., $\omega = 10^{-6}$ with pattern 1×16, thus creating a skewed query that is more challenging for HC.

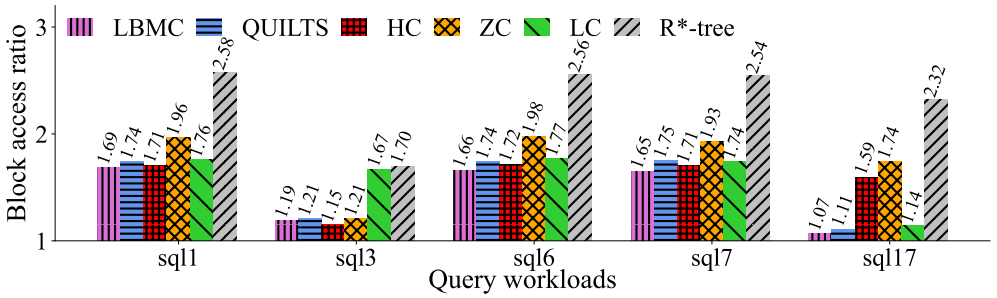


FIGURE 6.17: Varying query templates (TPC-H)

Results on TPC-H. Figure 6.17 shows that LBMC has the smallest query costs under all query templates except for `sql3`, since these templates are more skewed. HC is slightly better on `sql3`, where the queries are close to square in the first two dimensions, which is similar to the 1×1 pattern in Figure 6.15. These results confirm the robustness of LBMC and its practical applicability on real data and query workloads.

6.6 Summary

We studied efficient cost estimation for a family of SFCs, i.e., the BMCs. Our algorithms can compute the global and the local query costs of BMCs for a given query in constant time. We extended these algorithms to compute the global and the local query costs of BMCs in constant time given n queries, after an $O(n)$-time initialization.

These results bring two optimization opportunities: (1) online selection of a query-efficient BMC-based index at query time, when multiple BMC-based indices are available over the dataset to be queried; and (2) offline learning of a query-efficient BMC for data indexing, when a query workload is available beforehand. We exploit both opportunities for query processing. Experimental results on real datasets show that online selection of BMC-based indices helps reduce the number of data block accesses by up to 75.2%. For offline BMC learning, we propose a reinforcement learning-based algorithm. The resultant learned BMCs achieve range query costs that are lower than those of the state-of-the-art SFC design method and other baselines under nearly all settings tested.

In conclusion, the proposed cost estimation is efficient and the learned BMC improves the query efficiency. However, this study has some limitations. Firstly, we have primarily focused on real-world datasets in alignment with the experimental settings used in QUILTS [73]. To address this, future work will extend our inquiry into the performance of BMCs on synthetic datasets, e.g., skewed datasets. We aim to generalize the applicability and robustness of our algorithms.

Secondly, while our methodologies offer considerable benefits for BMCs, the SFC family is far more diverse. Extending our algorithms to accommodate this spectrum, especially to HCs, introduces a set of non-trivial challenges. These arise from the inherent differences between BMCs and HCs, including monotonicity and rotational transformations involved in HC construction. As part of our future research trajectory, we are committed to formulating new methodologies to bring the advantages of our current algorithms to the HC domain, thereby broadening the effectiveness of our approach.

Chapter 7

Conclusions and Future Directions

This thesis investigated the advantages and limitations of learned spatial indices and proposed methods and a system to improve the efficiency of machine learning-based optimization for spatial indices. In this chapter, we will outline some future directions including the practical application of learned spatial indices and learning different space-filling curves.

7.1 Conclusions

In this thesis, we have validated the query efficiency of learned spatial indices through a series of comprehensive experiments. We observed that the build time for these indices is substantial, which led us to develop a framework referred to as ELSI. This framework is designed to enhance index building efficiency while maintaining stable query performance. Additionally, we proposed an SFC-based index learning method, which improved both the building and query efficiency. From Chapter 3 to Chapter 6, we studied the empirical performance of learned spatial indices and reported their superior performance in query processing. As a part of our future direction, we aim to implement these indices in a DBMS.

In Chapter 2, we reviewed both traditional indices and the latest development of indexing techniques, i.e., the learned indices. For the traditional indices, we classified them by

dimensionality, i.e., one-dimensional indices and spatial indices. We further investigated spatial indices by their building methods. For the learned indices, we also classified them by dimensionality first. For the learned one-dimensional indices, we discussed two main types of building methods. For the learned spatial indices, we studied them based on whether the indices are learned with the knowledge of query workloads. Additionally, we discussed SFC-related techniques as SFCs are widely used in both traditional and learned spatial indices. Finally, we summarized three common types of spatial queries and the datasets used in recent studies on learned spatial indices.

In Chapter 3, we studied the empirical effectiveness and efficiency of two learned spatial indices ZM and RSMI by comparing them with a series of traditional spatial indices, providing a comprehensive basis for future studies on learned spatial indices. The experimental results on both real and synthetic data showed that RSMI outperforms traditional indices for processing point queries, range queries, and kNN queries. Nevertheless, the build time cost of the learned spatial indices is an issue, which can be about two orders of magnitude higher than that of the traditional spatial indices [58, 84, 103], which hinders the practical applicability of RSMI and ZM.

In Chapter 4, we proposed a versatile system called ELSI, which accelerates the building and re-building of a common class of learned spatial indices while retaining the high query efficiency of the indices. ELSI is applicable to learned spatial indices that follow the map-and-sort indexing paradigm and the predict-and-scan query paradigm. Most existing learned spatial indices satisfy these requirements, such as ZM, LISA, ML-index, and RSMI, which are all studied with the integration of ELSI. ELSI encompasses a suite of methods for constructing small and representative training datasets for index learning and rebuilds. Given an input dataset, ELSI can adaptively choose a training set reduction method that produces a learned index with high query efficiency. Experiments on real datasets with over 100 million points show that ELSI can consistently reduce the build times of four different learned spatial indices, i.e., ZM, LISA, ML-index, and RSMI, by up to two orders of magnitude. ELSI is also able to maintain highly efficient query performance for all the query types tested.

In Chapter 5, we proposed to use pre-trained models and fine-tuning for indexing new datasets to address the building overhead of learned indices. We proposed a CDF-based synthetic dataset generation method to generate a number of pre-trained models and use

EMD as a similarity metric to select candidate pre-trained models for our learned index. We further added fine-tuning to update the parameters, which can improve the prediction accuracy of the reused models. We demonstrated the effectiveness of the proposed techniques by applying them to the RMI learned indices [53]. Experimental results on both synthetic and real data show that our model reuse and fine-tuning techniques can improve the building and the lookup performance of RMI. We also extended model reuse and fine-tuning to learned spatial indices, which shows substantial improvement in index build time efficiency and retaining the query time efficiency.

In Chapter 6, instead of proposing another new index structure, we focused on optimizing traditional spatial indices. We proposed a deep reinforcement learning-based algorithm named LBMC to learn query-efficient SFCs, by formulating the learning as a bit-swapping process. LBMC works under both query-aware and query-agnostic scenarios, and our resultant indices are based on B^+-trees which are commonly available in modern database systems. These make LBMC highly applicable. To guide the learning process, we proposed a cost model that enables highly effective and efficient (constant time after an initial data scan) reward computation, which significantly outperforms naive the baseline methods. For the performance of LBMC experimental results on real datasets show that SFCs learned by LBMC have range query costs that are lower than those of the state-of-the-art SFC design method and other baselines under nearly all settings tested.

7.2 Future Directions

This thesis opens up new opportunities for future studies. We consider exploring future directions from the following directions.

7.2.1 Unlocking the Potential: Introducing Learned Spatial Indexing into Database Systems

Currently, traditional indices such as B^+-trees, quadtrees, and kd-trees are still the dominating index structures in commercial database management systems (DBMS). Despite the strong query performance of learned spatial indices as reported in recent studies, the practical usability of such indices has not been investigated in a real DBMS.

However, it is important to investigate how such indices will perform when they are implemented as part of a DBMS, and it is imperative to carry out further empirical studies to test the practical performance of such indices in a DBMS under real-world settings. Below, we discuss a few prominent technical challenges:

1. *There are no machine learning libraries in DBMS.* Learned indices require training over the dataset before they can be used. For example, RSMI uses LibTorch [83] to train models. However, machine learning libraries are not inherently supported by DBMS. Currently, linear models and neural networks are mostly used. In addition, RL is an important technique commonly used to optimize existing index structures. Therefore, integrating learning-based indices into a DBMS, and enabling applicable machine learning models can pave the way for learning data distribution.

2. *There is no query algorithms for complex query types over learned spatial indices.* While learned indices show promise for certain types of queries, for example, point queries, range queries, and kNN queries, they do not support more complex types of queries, which may be easier for the traditional spatial indices to address. For example, spatial join queries, which combine two spatial datasets based on their spatial relationships, and intersection queries, which find areas that overlap with a given region, are unavailable. Therefore, it is important to develop novel query algorithms for the learned spatial indices to support more query types.

3. *There is no query performance guarantee on learned spatial indices.* We have seen the strong query performance of learned spatial indices in many works. However, this is no guarantee that learned spatial indices can still be efficient in extreme cases. For example, there are hundreds of outliers in the *face* dataset [50] with 200 million data records. The query performance of learned indices can be impacted when the outliers are considered in the index building process. Thus, it is important to discover the worst-case performance of learned spatial indices. To simplify the discussion, we can use naive machine learning models, e.g., linear models to avoid the uncertainty of complex models, e.g., neural networks. Recall that the query cost of learned indices comes from prediction and search, where the complexity of prediction is $O(1)$. The complexity of the search needs to be studied based on the average search range, which might be theoretically derived from the training errors of the models.

4. *There is no simple update handling strategy for learned spatial indices.* Traditional database indices, such as B-trees or R-trees, can handle updates incrementally. When a single data point is updated, only a small part of the index structure needs to be updated. This makes them well suited to DBMSs where the data changes frequently.

   However, learned indices are based on machine learning models trained on a static dataset. When the data changes, the models may need to be rebuilt, which is usually a costly operation. Even RSMI, which can be partially updated, still requires a relatively long time to rebuild the model. Thus, the lack of efficient update handling is blocking the integration of learned spatial indices into real DBMS. Moreover, the model training process is usually not just costly but also complex, as it requires selecting appropriate features, model parameters, and training algorithms. When a model is retrained, there is no guarantee that the new model will maintain the same query performance as the old one.

5. *There is no fallback strategy when learned spatial indices show worse performance than traditional indices.* DBMS may choose to fall back to scan-based query processing when a traditional index has deteriorating query performance, e.g., due to data or query distribution changes. Similar strategies will need to be developed for learned spatial indices. A possible option is to build a parallel index structure, where a traditional spatial index, e.g., an R-tree, is maintained together with a learned spatial index, which is activated for query processing when the learned index has become less efficient in query processing.

7.2.2 An Effective Index Learning Framework for Query-Aware Learned Spatial Indices

As discussed in Chapter 2, learned spatial indices are classified into two types: query-aware and query-agnostic. Our ELSI framework has been designed for query-agnostic learned spatial indices. A new framework design for the efficient learning of query-aware learned spatial indices is in need.

To support query-aware indices, directly adding a module to our ELSI framework to learn from the query workloads cannot solve the problem, because the index building

process of the indices do not satisfy the assumption of ELSI. For example, the building process of RLR-tree and Qd-tree do not meet the map-and-sort paradigm.

To design such a framework, the key challenges are as follows:

1. *There is no standard formulation of query workloads.* In DBMS, a query workload refers to a set of queries that the system is expected to handle over a certain period of time. Such a workload can be described by the types of queries, the query frequency, distribution, selectivity, and complexity. When using query workloads to direct the learning of spatial indices, all these factors may have a strong impact on the structure of the resultant index. However, there is no standard formulation of workloads. Different studies have required knowledge on different aspect of an expected query workload, based on which index optimization algorithms were designed. It is challenging to design an index learning framework given the highly different requirements on and formulation of the query workloads.

2. *These is no unified query-aware spatial index learning algorithms.* Unlike the query-agnostic indices which largely follow the map-and-sort paradigm to build, the query-aware indices such as Flood [70], Tsunami [26], Qd-tree [108], and RLR-tree [35] all have very different ways to buid and utilize the query workloads.

   For example, Qd-tree uses an intermediately built index to run queries over provided query workloads, the cost of which is used as the reward in its reinforcement learning-based index optimization algorithm. The query cost calculation is triggered in each iteration during the learning process. In comparison, Flood [70] and Tsunami [26] learn indices by data space partitioning. They use the query workloads to learn a cost model, which is used to direct the partitioning of the data space. Designing a unified framework that can support efficient building of all these indices is challenging.

7.2.3 Learning to Optimize Hilbert Curve-Based Spatial Indices

In Chapter 6, we proposed a method to learn bit-merging curves (BMC) which produces index structures with strong query performance, outperforming both ZC and HC-based index structures. We showed that index learning based on BMCs offers an efficient

exploration of a subspace of the search space to find a query-optimal SFC. However, other types of SFCs have not been considered in the problem space, e.g., HCs.

While it may be challenging to explore the rest of the search space, a promising subspace is a generalization of HC. This is because index structures based on HC have reported better query performance than ZC [85, 107].

A HC is created through an iterative process. The first iteration is a simple 'U' shaped curve. For each subsequent iteration, the 'U' shaped curve is rotated and connected with other curves with a unit segment, which follows the previous scaled-down version. As the number of iterations approaches infinity, the curve becomes a fractal and fills the entire space. The original HC follows this pattern and cannot be learned dynamically.

To resolve this issue, we suggest to make the rotation and connection of 'U' shaped curves more flexible. Following our design for learning BMCs, where the bits can be swapped in the bit distribution pattern, we rotate the unit 'U' shaped curve for HCs which can also change the ordering of spatial points mapped by HCs.

The learning process also requires an efficient measurement of the query cost of HCs to guide the learning process. We proposed an efficient cost model for BMCs in Chapter 6, while it cannot be applied to HCs directly. This is because the cost model is based on the bit merging operations, which cannot fit the measurement of 'U' shaped curves. Existing cost models for HCs are also challenging because they need to go through the curves multiple times to calculate the cost, which is prohibitively expensive.

To address this challenge, a possible solution is to analyze the building pattern of HCs, i.e., the underlying principles of the 'U' shaped curves. To model this continuous pattern, we can classify the connections between two 'U' shaped curves into two categories, i.e., a horizontal or a vertical segment, assuming a two-dimensional case for simplicity.

7.2.4 A Benchmark for Learned Spatial Indices

To compare the empirical performance of one-dimensional learned indices, a benchmark named SOSD [50] has been presented that measures the index size, query performance, and build cost of the learned indices. A similar benchmark for learned spatial indices is a natural next step. Such a benchmark is non-trivial to achieve. It needs to be

comprehensive and representative to ensure that the learned spatial indices can be fairly evaluated, while the results are truthful reflections of the practical performance of the learned spatial indices. The following questions need to be answered when designing such a benchmark, which makes it another important future direction.

1. Which datasets should be used for the empirical comparison of learned spatial indices? The spatial datasets chosen need to be representative of a variety of different characteristics, e.g., distribution, density, dimensionality, and scale. We consider data sources including the OSM [76] dataset and synthetic datasets. OSM is a real dataset, which contains the POIs all over the world separated by different continents. Synthetic datasets are used to test different data distributions, which can be uniform, normal, or skewed distributions, with the dataset cardinality ranging from 1 million to 1 billion.

2. Queries: We also define the query methods, which should be common spatial queries well-applied in real applications. The queries should cover a range of different query types, such as point queries, range queries, and kNN queries. Query size, query distribution, and query selectivity are also important. We consider query workloads generated based on the data distribution of a given dataset. For point queries, the query point should be sampled from the dataset. For range query, the query range varies from 0.01% to 0.1% to simulate different selectivity. The aspect ratio of the range ranges should also vary, for example, from 1 to 100 in a two-dimensional space, where 1 means a square shape and 100 represents a narrow rectangle. For kNN queries, the query points can either be sampled from the datasets or generated based on the data distribution.

3. Metrics: The benchmark should record and report the index build time, index size, query response time, throughput, and other performance metrics. We record the index build time, which includes the data processing time, the model training time, and the index structure construction time. The query response time includes the prediction time and search time, which are two components of the query procedure in learned indices. The index size can be computed by checking the size of the index file or the memory it takes. The throughput is typically calculated as the number of operations per unit of time.

Bibliography

[1] Daniar Achakeev, Bernhard Seeger, and Peter Widmayer. Sort-based query-adaptive loading of R-trees. In *Proceedings of the Conference on Information and Knowledge Management (CIKM)*, pages 2080–2084, 2012.

[2] Amazon AWS. https://aws.amazon.com/quickstart/architecture/amazon-redshift, 2016. Accessed: 2023-05-31.

[3] Panagiotis Antonopoulos, Hanuma Kodavalla, Alex Tran, Nitish Upreti, Chaitali Shah, and Mirek Sztajno. Resumable online index rebuild in SQL server. *Proceedings of the VLDB Endowment (PVLDB)*, 10(12):1742–1753, 2017.

[4] Apache Hudi. https://hudi.apache.org/blog/2021/12/29/hudi-zorder-and-hilbert-space-filling-curves, 2021. Accessed: 2023-05-31.

[5] Lars Arge, Mark De Berg, Herman Haverkort, and Ke Yi. The Priority R-tree: A practically efficient and worst-case optimal R-tree. *ACM Transactions on Algorithms (TALG)*, 4(1):9:1–9:30, 2008.

[6] Manos Athanassoulis and Anastasia Ailamaki. BF-Tree: Approximate tree indexing. *Proceedings of the VLDB Endowment (PVLDB)*, 7(14):1881–1892, 2014.

[7] Rudolf Bayer. The universal B-tree for multidimensional indexing: General concepts. In *Proceedings of the International Conference on Worldwide Computing and Its Applications (WWCA)*, pages 198–209, 1997.

[8] Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica (AIM)*, 1:173–189, 1972.

[9] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In

Proceedings of the ACM International Conference on Management of Data (SIGMOD), pages 322–331, 1990.

[10] Norbert Beckmann and Bernhard Seeger. A revised R*-tree in comparison with related index structures. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 799–812, 2009.

[11] Shai Ben-David, John Blitzer, Koby Crammer, Alex Kulesza, Fernando Pereira, and Jennifer Wortman Vaughan. A theory of learning from different domains. *Machine Learning*, 79(1-2):151–175, 2010.

[12] Michael A. Bender, Erik D. Demaine, and Martin Farach-Colton. Cache-oblivious B-trees. *SIAM Journal on Computing*, 35(2):341–358, 2005.

[13] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[14] Stefan Berchtold, Christian Böhm, and Hans-Peter Kriegel. The Pyramid-technique: Towards breaking the curse of dimensionality. In *Proceedings of the Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 142–153, 1998.

[15] Elisa Bertino. A survey of indexing techniques for object-oriented database management systems. In *Query Processing for Advanced Database Systems*, pages 383–418. 1991.

[16] Christian Böhm. Space-filling curves for high-performance data mining. *CoRR*, abs/2008.01684, 2020.

[17] Chee-Yong Chan and Yannis E. Ioannidis. Bitmap index design and evaluation. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, page 355–366, 1998.

[18] Lisi Chen, Gao Cong, Christian S. Jensen, and Dingming Wu. Spatial Keyword Query Processing: An Experimental Evaluation. *Proceedings of the VLDB Endowment (PVLDB)*, 6(3):217–228.

[19] Kimberly C. Claffy, George C. Polyzos, and Hans-Werner Braun. Application of sampling methodologies to network traffic characterization. In *Proceedings of the*

ACM Special Interest Group on Data Communication (SIGCOMM), pages 194–203, 1993.

[20] Douglas Comer. Ubiquitous B-Tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[21] Corinna Cortes and Mehryar Mohri. Domain adaptation in regression. In *Proceedings of the International Conference on Algorithmic Learning Theory (ALT)*, pages 308–323, 2011.

[22] Databricks Engineering Blog. https://databricks.com/blog/2018/07/31/processing-petabytes-of-data-in-seconds-with-databricks-delta.html, 2018. Accessed: 2023-05-31.

[23] Angjela Davitkova, Evica Milchevski, and Sebastian Michel. The ML-Index: A multidimensional, learned index for point, range, and nearest-neighbor queries. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 407–410, 2020.

[24] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*, pages 4171–4186, 2019.

[25] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, David Lomet, and Tim Kraska. ALEX: An updatable adaptive learned index. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 969–984, 2020.

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

[27] Earth Mover's Distance. https://en.wikipedia.org/wiki/Earth_mover%27s_distance, 2021. Accessed: 2023-05-31.

[28] Mohamed Y. Eltabakh, Ramy Eltarras, and Walid G. Aref. Space-partitioning trees in postgresql: Realization and performance. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, page 100, 2006.

[29] Christos Faloutsos and Shari Roseman. Fractals for secondary key retrieval. In *Proceedings of the Principles of Database Systems (PODS)*, pages 247–252, 1989.

[30] Paolo Ferragina and Giorgio Vinciguerra. The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds. *Proceedings of the VLDB Endowment (PVLDB)*, 13(8):1162–1175, 2020.

[31] Raphael FinkelJon and Louis Bentley. Quad trees: A data structure for retrieval on composite keys. *Acta Informatica*, 4(1):1–9.

[32] Volker Gaede and Oliver Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, 1998.

[33] Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. Fiting-tree: A data-aware index structure. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 1189–1206, 2019.

[34] Yván J. García R, Mario A. López, and Scott T. Leutenegger. A greedy algorithm for bulk loading R-trees. In *ACM International Conferences on Advances in Geographic Information Systems (SIGSPATIAL)*, pages 163–164, 1998.

[35] Tu Gu, Kaiyu Feng, Gao Cong, Cheng Long, Zheng Wang, and Sheng Wang. The RLR-Tree: A Reinforcement Learning Based R-Tree for Spatial Data. *Proceedings of the ACM on Management of Data (PACMMOD)*, 1(1):63:1–63:26, 2023.

[36] Na Guo, Yaqi Wang, Haonan Jiang, Xiufeng Xia, and Yu Gu. TALI: An Update-Distribution-Aware Learned Index for Social Media Data. *Mathematics*, 10(23), 2022.

[37] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 47–57, 1984.

[38] Ali Hadian and Thomas Heinis. Considerations for handling updates in learned index structures. In *Proceedings of the International Workshop on Exploiting Artificial Intelligence Techniques for Data Management (aiDM)*, pages 3:1–3:4, 2019.

[39] Ali Hadian and Thomas Heinis. Interpolation-friendly b-trees: Bridging the gap between algorithmic and learned indexes. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, 2019.

[40] Hash Indexes. https://www.postgresql.org/docs/current/hash-intro.html, 2022. Accessed: 2023-01-20.

[41] Herman Haverkort and Freek V. Walderveen. Four-dimensional hilbert curves for r-trees. *Journal of Experimental Algorithmics*, 16:3.4:3.1–3.4:3.19, 2008.

[42] Claire E. Heaney, Yuling Li, Omar K. Matar, and Christopher C. Pain. Applying convolutional neural networks to data on unstructured meshes with space-filling curves. *CoRR*, abs/2011.14820, 2020.

[43] H. V. Jagadish. Linear clustering of objects with multiple attributes. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, page 332–342, 1990.

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

[45] Christian S. Jensen, Dan Lin, and Beng Chin Ooi. Query and update efficient B^+-tree based indexing of moving objects. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 768–779, 2004.

[46] Dmitri V. Kalashnikov, Sunil Prabhakar, and Susanne Hambrusch. Main memory evaluation of monitoring queries over moving objects. *Distributed and Parallel Databases*, 15(2):117–135, 2004.

[47] Ibrahim Kamel and Christos Faloutsos. Hilbert R-tree: An improved R-tree using fractals. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 500–509, 1994.

[48] Daniel Kang, John Guibas, Peter D. Bailis, Tatsunori Hashimoto, and Matei Zaharia. TASTI: semantic indexes for machine learning-based queries over unstructured data. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 1934–1947, 2022.

[49] Andreas Kipf, Ryan Marcus, Alexander an Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. RadixSpline: A single-pass learned index. In *Proceedings of the International Workshop on Exploiting Artificial Intelligence Techniques for Data Management (aiDM)*, pages 5:1–5:5, 2020.

[50] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. SOSD: A benchmark for learned indexes. *NeurIPS Workshop on Machine Learning for Systems*, 2019.

[51] Kolmogorov-Smirnov Test. https://en.wikipedia.org/wiki/Kolmogorov-Smirnov_test, 2021. Accessed: 2023-05-31.

[52] Tim Kraska, Mohammad Alizadeh, Alex Beutel, Ed H. Chi, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. SageDB: A learned database system. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, 2019.

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

[54] Warren M. Lam and Jerome M. Shapiro. A class of fast algorithms for the peano-hilbert space-filling curve. In *Proceedings of the International Conference on Image Processing (ICIP)*, pages 638–641, 1994.

[55] Ju-Hong Lee, Guang-Ho Cha, and Chin-Wan Chung. A Model for k-Nearest Neighbor Query Processing Cost in Multidimensional Data Space. *Information Processing Letters*, 69(2):69–76, 1999.

[56] Scott T. Leutenegger, J. M. Edgington, and Mario A. López. STR: A simple and efficient algorithm for R-Tree packing. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 497–506, 1997.

[57] Guoliang Li, Xuanhe Zhou, and Lei Cao. AI Meets Database: AI4DB and DB4AI. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, page 2859–2866, 2021.

[58] Pengfei Li, Hua Lu, Qian Zheng, Long Yang, and Gang Pan. LISA: A learned index structure for spatial data. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 2119–2133, 2020.

[59] Yaliang Li, Daoyuan Chen, Bolin Ding, Kai Zeng, and Jingren Zhou. A pluggable learned index method via sampling and gap insertion. *CoRR*, abs/2101.00808, 2021.

[60] Yishay Mansour, Mehryar Mohri, and Afshin Rostamizadeh. Domain adaptation: Learning bounds and algorithms. In *Proceedings of the Conference on Learning Theory (COLT)*, 2009.

[61] Ryan Marcus, Andreas Kipf, Alexander van Renen, Mihail Stoian, Sanchit Misra, Alfons Kemper, Thomas Neumann, and Tim Kraska. Benchmarking learned indexes. *Proceedings of the VLDB Endowment (PVLDB)*, 14(1):1–13, 2020.

[62] Ryan Marcus, Emily Zhang, and Tim Kraska. CDFShop: Exploring and optimizing learned index structures. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, page 2789–2792, 2020.

[63] Stephen McAleer, Forest Agostinelli, Alexander Shmakov, and Pierre Baldi. Solving the rubik's cube without human knowledge. *CoRR*, abs/1805.07470, 2018.

[64] Microsoft. https://docs.microsoft.com/en-us/sql/relational-databases/indexes/clustered-and-nonclustered-indexes-described?view=sql-server-ver15, 2022. Accessed: 2023-05-31.

[65] Mayank Mishra and Rekha Singhal. RUSLI: real-time updatable spline learned index. In Rajesh Bordawekar, Yael Amsterdamer, Oded Shmueli, and Nesime Tatbul, editors, *Proceedings of the Workshop in Exploiting AI Techniques for Data Management (aiDM)*, pages 1–8, 2021.

[66] Michael Mitzenmacher. A model for learned bloom filters and optimizing by sandwiching. In *Proceedings of the Neural Information Processing Systems (NeurIPS)*, pages 462–471, 2018.

[67] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

[68] Bongki Moon, H. V. Jagadish, Christos Faloutsos, and Joel H. Saltz. Analysis of the clustering properties of the hilbert space-filling curve. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 13(1):124–141, 2001.

[69] Kimia Nadjahi, Alain Durmus, Pierre E. Jacob, Roland Badeau, and Umut Şimşekli. Fast approximation of the sliced-wasserstein distance using concentration of random projections. In *Proceedings of the Neural Information Processing Systems (NeurIPS)*, pages 12411–12424, 2021.

[70] Vikram Nathan, Jialin Ding, Mohammad Alizadeh, and Tim Kraska. Learning multi-dimensional indexes. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 985–1000, 2020.

[71] Navicat Blog: Oracle Rebuild. https://www.navicat.com/en/company/aboutus/blog/1303-how-to-tell-when-it-s-time-to-rebuild-indexes-in-oracle, 2020. Accessed: 2023-05-31.

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

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

[74] Sarana Nutanong, Rui Zhang, Egemen Tanin, and Lars Kulik. Analysis and evaluation of v*-$k$nn: an efficient algorithm for moving $k$nn queries. *The VLDB Journal*, 19(3):307–332, 2010.

[75] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. The log-structured merge-tree (LSM-Tree). *Acta Informatica (AIM)*, 33(4):351–385, 1996.

[76] OpenStreetMap US Northeast data dump. https://download.geofabrik.de/, 2018. Accessed: 2023-05-31.

[77] Jack A. Orenstein. Spatial query processing in an object-oriented database system. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 326–336, 1986.

[78] Jack A. Orenstein and T. H. Merrett. A class of data structures for associative searching. In *Proceedings of the Principles of Database Systems (PODS)*, pages 181–190, 1984.

[79] Keiron O'Shea and Ryan Nash. An introduction to convolutional neural networks. *CoRR*, abs/1511.08458, 2015.

[80] Sachith Pai, Michael Mathioudakis, and Yanhao Wang. Towards an instance-optimal Z-index. In *Proceedings of the International Workshop on Applied AI for Database Systems and Applications (AIDB)*, 2022.

[81] PostGIS. https://postgis.net, 2022. Accessed: 2023-05-31.

[82] PostgreSQL. https://www.postgresql.org/docs/current/indexes-multicolumn.html, 2022. Accessed: 2023-05-31.

[83] PyTorch. https://pytorch.org, 2016. Accessed: 2023-05-31.

[84] Jianzhong Qi, Guanli Liu, Christian S. Jensen, and Lars Kulik. Effectively learning spatial indices. *Proceedings of the VLDB Endowment (PVLDB)*, 13(11):2341–2354, 2020.

[85] Jianzhong Qi, Yufei Tao, Yanchuan Chang, and Rui Zhang. Theoretically optimal and empirically efficient R-trees with strong parallelizability. *Proceedings of the VLDB Endowment (PVLDB)*, 11(5):621–634, 2018.

[86] Jun Rao and Kenneth A. Ross. Cache conscious indexing for decision-support in main memory. In *Proceedings of the VLDB Endowment (VLDB)*, page 78–89, 1999.

[87] John T. Robinson. The K-D-B-tree: A search structure for large multidimensional dynamic indexes. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 10–18, 1981.

[88] Nick Roussopoulos, Stephen Kelley, and Frédéic Vincent. Nearest neighbor queries. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 71–79, 1995.

[89] Nick Roussopoulos and Daniel Leifker. Direct spatial search on pictorial databases using packed R-trees. In *Proceedings of the Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 17–31, 1985.

[90] S2 Geometry. http://s2geometry.io, 2022. Accessed: 2023-05-31.

[91] Sartaj Sahni. Tries. In *Handbook of Data Structures and Applications*. 2004.

[92] Hasim Sak, Andrew W. Senior, and Françoise Beaufays. Long Short-Term Memory Based Recurrent Neural Network Architectures for Large Vocabulary Speech Recognition. *CoRR*, abs/1402.1128, 2014.

[93] Timos K. Sellis, Nick Roussopoulos, and Christos Faloutsos. The R^+-Tree: A dynamic index for multi-dimensional objects. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 507–518, 1987.

[94] Avi Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts, Seventh Edition*. McGraw-Hill Book Company, 2020.

[95] STX B+ Tree. https://panthema.net/2007/stx-btree, 2007. Accessed: 2023-05-31.

[96] Yufei Tao, Dimitris Papadias, and Qiongmao Shen. Continuous nearest neighbor search. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*, pages 287–298, 2002.

[97] TIGER/Line Shapefiles. https://www.census.gov/geo/maps-data/data/tiger-line.html, 2006. Accessed: 2023-05-31.

[98] TLC Trip Record Data. https://www.nyc.gov/site/tlc/about/tlc-trip-record-data.page, 2022. Accessed: 2023-05-31.

[99] TPC-H. http://www.tpc.org/tpch/, 2022. Accessed: 2023-05-31.

[100] Panagiotis Tsinganos, Bruno Cornelis, Cornelis Jan, Bart Jansen, and Athanassios Skodras. The effect of space-filling curves on the efficiency of hand gesture recognition based on semg signals. *International Journal of Electrical and Computer Engineering (IJECE)*, 12:23–31, 2021.

[101] Peter Van Sandt, Yannis Chronis, and Jignesh M. Patel. Efficiently searching in-memory sorted arrays: Revenge of the interpolation search? In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, page 36–53, 2019.

[102] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proceedings of the Advances in Neural Information Processing Systems (NeurIPS)*, volume 30, 2017.

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

[104] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G. Andersen. Building a Bw-Tree takes more than just buzz words. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, page 473–488, 2018.

[105] Jiacheng Wu, Yong Zhang, Shimin Chen, Jin Wang, Yu Chen, and Chunxiao Xing. Updatable learned index with precise positions. *Proceedings of the VLDB Endowment (PVLDB)*, 14(8):1276–1288, 2021.

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

[107] Pan Xu and Srikanta Tirthapura. Optimality of clustering properties of space-filling curves. *ACM Transactions on Database Systems (TODS)*, 39(2):10:1–10:27, 2014.

[108] Zongheng Yang, Badrish Chandramouli, Chi Wang, Johannes Gehrke, Yinan Li, Umar Farooq Minhas, Per-Åke Larson, Donald Kossmann, and Rajeev Acharya.

Qd-tree: Learning data layouts for big data analytics. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 193–208, 2020.