Research on Accurate and Fast Learned Cardinality Estimation for Spatial Data

Submitted to
Graduate School of Information Science and Technology
Osaka University

January 2025

Yuchen JI

List of Publications

1. Journal Paper

1. Y. Ji, D. Amagata, Y. Sasaki, and T. Hara. Efficient Learned Cardinality Estimation for Dynamic Spatial Data. *IEICE Transactions on Information and Systems,* J108-D(5):OO–OO, 2025.

2. International Conference Paper

1. Y. Ji, D. Amagata, Y. Sasaki, and T. Hara. A Performance Study of One-dimensional Learned Cardinality Estimation. *International Workshop on Design, Optimization, Languages and Analytical Processing of Big Data*, pp. 86-90, Mar. 2022.

2. Y. Ji, D. Amagata, Y. Sasaki, and T. Hara. SAFE: Sampling-assisted Fast Learned Cardinality Estimation for Dynamic Spatial Data. *The International Conference on Database and Expert Systems Applications (DEXA)*, pp. 201-216, Aug. 2024.

3. D. Amagata, J. Yamada, Y. Ji, and T. Hara. Efficient Algorithms for Top-k Stabbing Queries on Weighted Interval Data. *The International Conference on Database and Expert Systems Applications (DEXA)*, pp. 146-152, Aug. 2024.

3. Domestic Conference Paper

1. Y. Ji, D. Amagata, Y. Sasaki, and T. Hara. Fast Learned Cardinality Estimation for Dynamic Spatial Data. *Proc. of DEIM Forum*, Feb. 2024.

2. Y. Ji, D. Amagata, Y. Sasaki, and T. Hara. Learned Cardinality Estimator for Intersection Queries on Two-dimensional Polygon Data. *IPSJ SIG-DBS / IPSJ SIG-IFAT / IEICE DE*, Dec. 2024.

Abstract

Spatial applications are becoming increasingly important in our daily lives. For instance, people can easily find their way to a railway station in a foreign city with mobile map applications. These applications process a large amount of queries to meet users' needs every day. Fast responses to queries are crucial for providing good user experiences. Therefore, efficient query processing is always in demand for spatial applications. Estimating a query's result size, known as the cardinality estimation, plays a significant role in query scheduling, optimization, and processing. For example, query optimization generates optimized query plans based on the estimated cardinality of subqueries. Accurate and fast cardinality estimation substantially improves query efficiency.

Recently, with the advancement of machine learning technologies, many learned methods have been proposed for cardinality estimation. Generally, they have shown greatly improved performance compared with traditional methods like histograms. However, existing methods cannot keep accurate and fast estimations facing following challenges brought by spatial data. (i) Complex data distribution: Spatial data are generally multi-dimensional and tend to have a complex data distribution. Some existing methods adopt complex neural networks to approximate the data distribution, resulting in large inference times. (ii) Frequent data updates: Data updates are frequent in spatial applications. It is challenging for learned methods to keep a good accuracy on a frequent-updated data distribution. Designing an efficient updating strategy remains an open question for current learning models. (iii) Variable data size: Complex spatial data types like polygons have variable sizes. A polygon consists of a variable number of vertices. Therefore, a set of polygons has a large range of possible data sizes. Existing learned solutions cannot handle input data of variable sizes efficiently.

In this thesis, we focus on fast and accurate learned cardinality estimation for spatial data, and address the above challenges. This thesis consists of five chapters. We introduce the research background and issues for learned cardinality estimation in Chapter 1. In Chapter 2, we demonstrate that light-weight learning models are promising

for one-dimensional cardinality estimation and have the potential for spatial data. In Chapter 3, we address the case of the dynamic spatial data and propose SAFE, a learned method that supports frequent updates. In Chapter 4, we propose PolyCard, a learned cardinality estimator specialized for intersection queries on polygons.

In Chapter 2, we investigate cardinality estimation in the one-dimensional case. We note that the latest proximity query processing methods benefit from learned models (indexes) and have shown better performances than non-learned indexing approaches. Specifically, we address the one-dimensional cardinality estimation problem and consider light-weight learning methods. We first design a prototype of a learned method for one-dimensional cardinality estimation. Second, we empirically evaluate the learned method together with existing methods. Then, we analyze the strong and weak points of these methods and find that the proposed learned method is promising and has the potential for the spatial data case.

In Chapter 3, we focus on dynamic spatial data and propose SAFE (Sampling-Assisted Fast learned cardinality Estimator). Dynamic setting assumes frequent data updates, which brings challenges for the task of cardinality estimation. The estimation methods are expected to be update-friendly and keep fast estimation. Facing these challenges, we specifically develop a sampling strategy that uses a quad-tree-based data partitioning and extracts a small subset to enable fast training of cardinality estimation models. In addition, we employ two-tier regression models to approximate the spatial data distribution while achieving accurate and fast cardinality estimation. Furthermore, we provide an incremental model update strategy to avoid re-training all models from scratch when we receive updates. We conduct experiments on real and synthetic datasets, and their results demonstrate that SAFE (i) outperforms state-of-the-art cardinality estimation models for spatial data and (ii) efficiently handles data updates while ensuring accurate and low-latency estimation.

In Chapter 4, we present PolyCard, a learned cardinality estimator for intersection queries on spatial polygons. Polygons are the most representative spatial objects and cardinality estimation for polygons has not been well studied. The commonly used bounding box approximation is intrinsically inaccurate. Existing learning methods cannot be applied to polygons directly due to the variable sizes of polygons. We develop an adaptive sampling method to transform polygon data to a fixed size, which considers the spatial distribution of the coordinates belonging to a polygon, and successfully apply learning techniques to spatial polygons. We propose a training data generator to

provide training data distributed over the data space and covering different cardinalities. Our experiments on four real-world datasets of millions of polygons demonstrate the efficiency and effectiveness of PolyCard.

In Chapter 5, we conclude this thesis and discuss our future work. Our proposed methods can achieve fast and accurate cardinality estimation for spatial data and can further accelerate query processing.

Table of contents

Chapter 1

Introduction

1.1 Background

Spatial applications are prevalent in our daily lives, greatly changing the way we interact with our surroundings [3–7, 23, 37, 39, 76, 77, 90]. For example, people can easily explore nearby sightseeing spots and restaurants, and find the routes to them with a map application on their mobile phones. To process requests from users, the systems behind the applications need to execute queries to find the corresponding information from their databases. Suppose that we want to find a Japanese restaurant within 1 kilometer (km) from our current location, the system will formulate this request into a query like finding locations satisfying the following requirements: 1) is a Japanese restaurant and 2) within 1 km of the current position. Then the system processes it and retrieves the restaurant information to answer the request.

These spatial applications handle a large volume of requests all the time, necessitating efficient query processing mechanisms. As shown in Figure 1.1, the systems employ functions such as query scheduling [14, 29, 30, 82, 114] and query optimization [2, 61, 62, 86, 105] to process queries efficiently. Specifically, they rely on query scheduling to allocate the queries to multiple servers. Ideally, the computation costs of query requests allocated to each server should be balanced. Otherwise, it will result in a waste of resources because some servers are idle. For queries with multiple filters, like the former example of finding a nearby Japanese restaurant, the system will generate the query plan and divide the query into subqueries to be executed sequentially. To obtain the most efficient query plan, the query optimization needs to optimize the sequence of the subqueries based on the query costs of different choices.

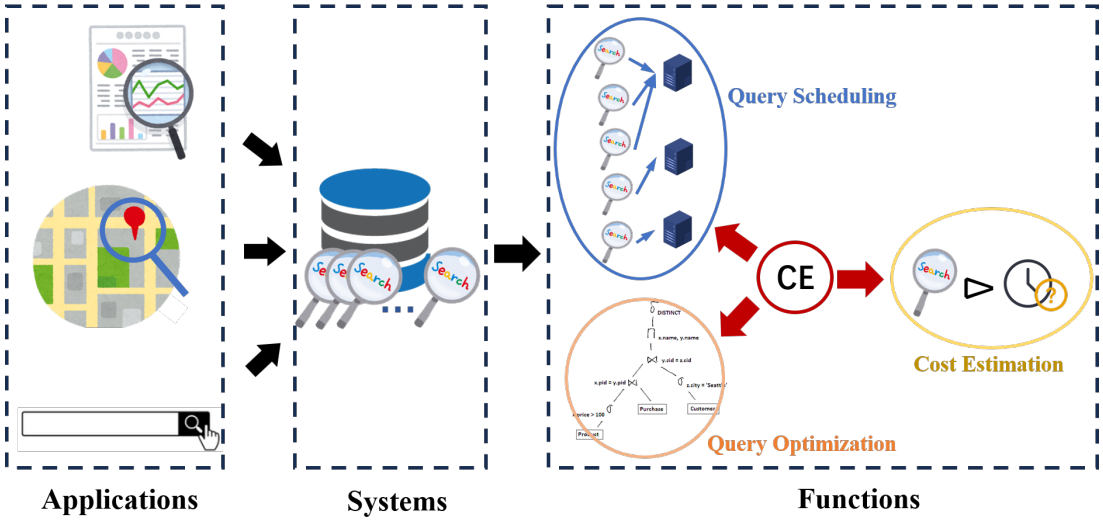


Figure 1.1: Cardinality Estimation (CE) plays a core role in cost estimation, query scheduling, and query optimization. The systems rely on these functions to process large amounts of queries from the application.

We observe that query scheduling and optimization cannot work without the knowledge of query costs. While we cannot know the exact query cost easily before the execution, we can estimate the query costs based on the cardinality as shown in Figure 1.2 [104, 109]. Cardinality is the number of data points in a dataset satisfying a query predicate. In most cases, a query with a larger cardinality has more data accesses and thus, incurs a higher computation cost [33, 54]. In common practice, cardinality estimation is necessary to estimate the query costs and plays the core role for functions including query scheduling and query optimization [34, 51, 57].

Accurate and fast cardinality estimation is always desired for the systems [2, 13, 60, 89, 97]. An accurate estimation of the cardinality can make the cost estimation of queries reliable and one step further, achieve a balanced query scheduling and an optimized query plan. Thus, accurate cardinality estimation can improve the system performance of spatial applications. Besides, the spatial applications ask for fast responses. During processing the queries, the cardinality estimation will be called frequently and contribute to the total response time. If the cardinality can be estimated in a shorter time, functions like query scheduling and optimization that rely on it can also finish earlier. Therefore, fast cardinality estimation can make the whole query processing efficient. Besides, the query time provides an upper limitation for estimation speed. Because we can obtain

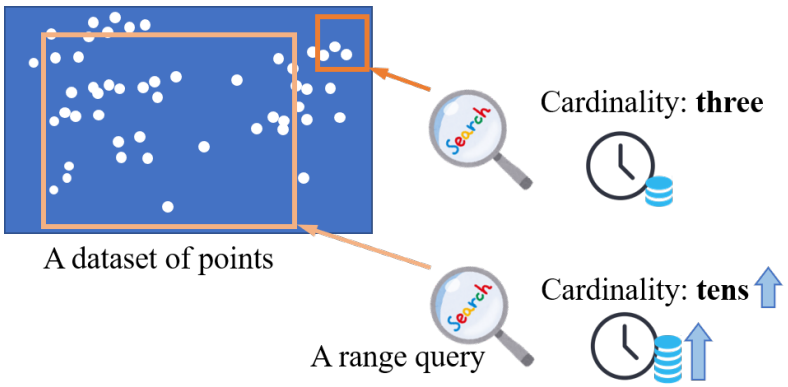


Figure 1.2: A Larger cardinality means a higher query cost.

the true cardinality (and do not need to know the cardinality anymore) after the query execution. Generally, we expect the estimation time to be shorter than one-hundredth of the query time [33, 88].

1.2 Challenges

1.2.1 Challenges Brought by the Spatial Data

The spatial data, also known as geospatial data, is used to represent the geographical area and locations [43]. It is the foundation of spatial applications. Common types of spatial data include points, lines, and polygons. Compared with other data types like timestamps and user IDs, spatial data are usually represented by large volumes of values and have a more complex data distribution. Here, we introduce the following attributes of spatial data and the new challenges for cardinality estimation brought by these attributes:

- Multi-dimensional: The most representative spatial data type is the point, usually consisting of coordinates. Compared with common one-dimensional data like timestamps, these multi-dimensional data tend to have a complex data distribution, which makes cardinality estimation difficult.
- Frequent updates: Considering the spatial applications, there are large amounts of updates like adding new locations during daily operations. For example, Google

Maps has 20 million updates of various spatial information every day[1]. As a result, the task of cardinality estimation faces frequent updates. It is challenging to keep an accurate model on a changing data distribution.

- Variable sizes of data: Some complex spatial data types like polygons have a variable size of data. A polygon is represented by a variable number of vertices. Thus, a polygon dataset usually includes polygons represented by a large-scale size of data. Existing learned-based methods cannot easily handle inputs of variable sizes.

1.2.2 Limitations of Existing Methods

Traditional methods for cardinality estimation are mainly histogram-based and sampling-based methods. Histogram-based methods [1, 49] build histograms as the statistics of the data distribution. Given a query, they scan the buckets accessed by the query and give an estimated cardinality. Sampling-based methods [18] generate a set of samples from the original dataset. Then, they execute the query on this sample set and obtain an estimated cardinality for the query on the original dataset. These traditional methods are easy to build and practical for simple cases where datasets follow a uniform data distribution. However, they are inflexible and not suitable for complex data distribution of multi-dimensional spatial data. Thus, traditional solutions do not satisfy the case of spatial data.

Recent results of *learned* cardinality estimation methods show remarkable improvements compared with traditional methods (e.g., histogram-based methods) [35, 57, 68, 71, 98, 100, 102, 109]. Learned cardinality estimation methods can be categorized into query-driven models and data-driven models.

Query-driven models are built on training queries and map a training query to its true cardinality. Most of them use deep learning to capture the relationship between queries and cardinalities [78, 97]. They are suitable for the case that data distribution is too complex to be approximated by data-driven methods. However, it is laborious to collect a large number of training queries and compute their true cardinality. Besides, these solutions are effective only when datasets are static and query workloads are known, whereas the nature of "relying on training queries" is not appropriate for dynamic data and unknown query workloads. Considering the frequent updates of spatial data, if

[1] https://mapsplatform.google.com/resources/blog/9-things-know-about-googles-maps-data-beyond-map/

query-driven models need to deal with data updates, they need to collect training samples on updated datasets and retrain the models. The high costs of obtaining training data disturb the direct usage of query-driven models for spatial data.

Data-driven models usually learn the distribution of a given dataset mainly through deep neural networks [35, 71, 109]. Autoregressive models [35, 109] can approximate the joint distribution in a conditional manner and estimate the cardinality without any independent assumptions. Considering the complex data distribution of spatial data, they usually build a deep neural network to approximate the data distribution and the construction takes a long time [35]. Besides, the estimation time can be large due to the complex neural networks. This attribute limits to only static data, as they cannot support fast model updates. LHist [62] and MOSE [89] are two state-of-the-art cardinality estimators approximating the data distribution directly with simple regression models instead of neural networks. However, they also do not support model updates. More importantly, these learning-based methods face difficulties with complex spatial data of variable sizes. Direct solutions like zero-padding will greatly increase the volume of data to be processed during training and estimation. Thus, it is challenging to apply existing learning-based solutions to spatial data.

1.3 Research Contents

Facing the challenges introduced in Section 1.2, we want to design learned solutions that break the limitations of existing solutions and overcome the challenges brought by spatial data.

There are still questions about selecting different learned approaches for different cases. Data-driven learned methods can easily achieve good accuracy in most cases. However, if the data distribution is too complex to be approximated, complex data-driven models will be used for the approximation, which leads to slow estimation speed. In that case, we seek for a query-driven solution. To conclude, we aim at fast and accurate cardinality estimation and try to design learned solutions to achieve the goal.

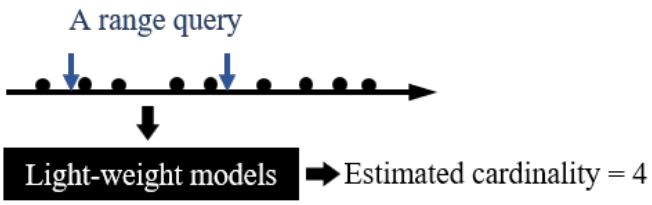


Figure 1.3: In Chapter 2, we investigate the learned cardinality estimation on a one-dimensional case by using light-weight models.

1.3.1 A Light-weight Learned Solution for the One-dimensional Cardinality Estimation

As shown in Figure 1.3, we focus on a one-dimensional case in Chapter 2. We successfully apply light-weight models to the one-dimensional cardinality estimation. Existing learned solutions have not discussed the one-dimensional case, which is simple but fundamental. They mainly use complex neural networks, which are slow and unsuitable for the simple one-dimensional case. Therefore, whether learning methods can be applied to one-dimensional cardinality estimation remains a question. We expect our investigation of the one-dimensional case to be a basis to help solve the case of spatial data.

In Chapter 2, we design a one-dimensional cardinality estimation method inspired by the learned index [21, 53, 74, 80]. A learned index uses light-weight learning models to map a query key to the corresponding position. Its core idea is to use learning models to approximate the data distribution. The data-driven cardinality estimation also uses techniques to approximate the data distribution and then estimate based on the approximation. The most important thing is that the light-weight models (usually simple linear regression models) can guarantee an extremely fast estimation speed. At the same time, they can provide a good approximation quality, which means the potential to achieve good accuracy for the cardinality estimation. With this idea, we design a structure using only light-weight models for cardinality estimation, referring to a learned index. Its estimation time for a range query is only one-tenth of other competitors. Also, its accuracy is relatively good. Through our investigation of the one-dimensional case, we find the light-weight learning models promising for cardinality estimation.

1.3.2 Learned Cardinality Estimation for Dynamic Spatial Data

As introduced in Section 1.2.1, dynamic spatial data is a common case for popular spatial applications. We have frequent data updates including insertions, deletions, and replacements in a dynamic setting, as shown in Figure 1.4. These data updates will change the data distribution, which could make the cardinality estimation inaccurate. For example, the true cardinality of the range query in Figure 1.4 changes from ten to nine after data updates. If the estimator has not been updated after the data updates, it may decrease the accuracy. Data-driven methods face continuously changing data, and their approximated data distribution will no longer be accurate unless they can frequently update their models to keep up with the changing data distribution. Query-driven methods are built on training queries with corresponding true cardinality. However, the cardinality of training queries will probably change after data updates. Though some query-driven methods adopt an incremental update strategy to continuously train on the latest queries, the outdated information still hurts the accuracy.

We note that the light-weight models discussed in Section 1.3.1 can be extended to approximate two-dimensional data. They can be organized as a hierarchy and approximate the data distribution on each dimension sequentially. This dividable structure has the potential to support frequent updates. However, how to design the update strategy and make it practical remains a challenging task. Facing frequent data updates, simply "supporting" updates is not enough. Fast model updating is also desired. In an ideal case, the model can update itself for every data update. However, the large data volume of real-world datasets and applications leads to high training costs for both the model construction and updates. Reducing the volume of data for training is a practical way. We review sampling-based solutions, and find if we can make a sample set for the model training, the training can be greatly accelerated. Notice that such a sample set has to be update-insensitive, which is a challenge.

Chapter 3 presents our solution SAFE, a learned cardinality estimator for dynamic spatial data. The design of SAFE derives from our investigation in Section 2. We use a hierarchy of regression models to approximate the data distribution and further help estimate the cardinality. We propose an incremental update strategy whose core idea is to update only models that need to be updated. Thus, for most updates, only a small portion of regression models need to be updated, which costs a little. To make an updatable sample set, we design a sampling strategy based on a quad-tree-based partition. We generate samples according to the cells partitioned by a quad-tree. With the support

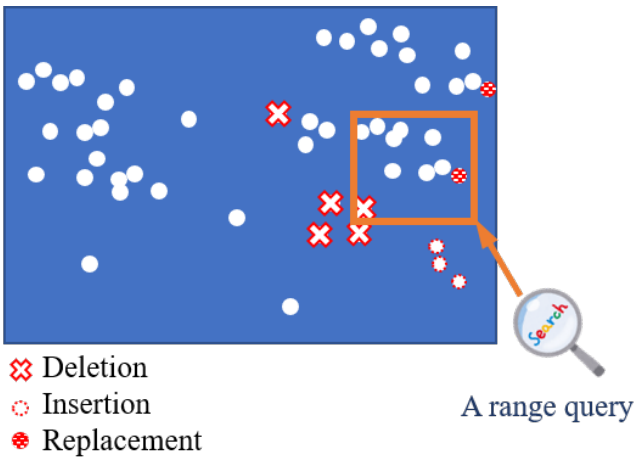


Figure 1.4: In Chapter 3, we focus on the case of dynamic spatial data.

from the quad-tree, the update of the sample set is easy. Furthermore, we can achieve fast model updates by training on the sample set.

1.3.3 Learned Cardinality Estimation for Complex Spatial Polygons

Many objects in spatial applications like districts, parks, and lakes are represented by polygons. Intersection queries on polygons find polygons intersected by the query polygons and Figure 1.5 shows an example of an intersection query on polygons. Intersection queries become prevalent due to the widespread spatial applications. Therefore, the cardinality estimation for intersection queries on spatial polygons is an important task to achieve good user experiences of spatial applications.

Currently, this task has not been well studied. Existing cardinality estimation methods for polygons typically rely on histograms built on the minimum bounding rectangles (MBRs) of polygons [55]. We can easily imagine that the approximation of polygons by MBRs is essentially inaccurate. Besides, they inherit the limitations of the histogram-based methods and cannot maintain stable performance for different datasets and queries. Unfortunately, recently proposed learned methods [35, 51, 89] cannot be used for spatial polygons due to the variable size of polygons as discussed in Section 1.2.1. The most common solution to this issue is padding all the input data with zeros [36]. Considering the possible sizes of polygons, the zero padding greatly increases the computation cost for both training and estimation, which makes it impractical for polygons.

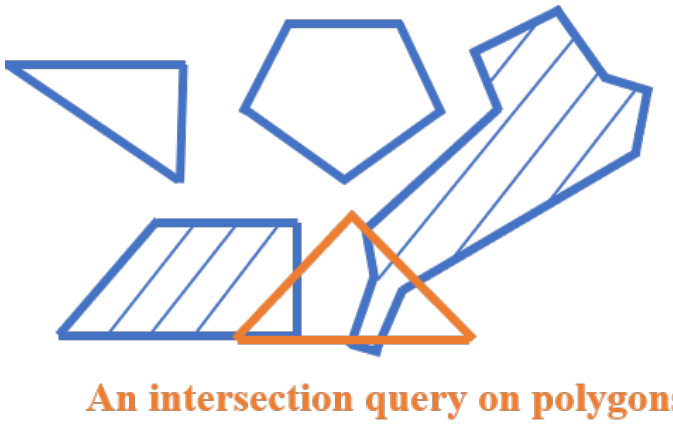


Figure 1.5: In Chapter 4, we study the learned cardinality estimation for intersection queries on polygons.

Low latency is always required for the cardinality estimation problem because the cardinality estimation is frequently called in a database system. Moreover, collecting high-quality training data is always challenging for learned cardinality estimators [51, 54]. One may come up with randomly generating query (training) polygons to address this issue. This is, however, not appropriate because the shape and the number of vertices of polygons are not fixed. Randomly generated polygons consisting of randomly generated vertices may be invalid due to crossed edges. Furthermore, the cardinality distribution of randomly generated queries has a long tail. The distribution with a long tail makes the convergence of the training of a network difficult [111]. As can be seen above, existing techniques are not appropriate for cardinality estimation of intersection queries on polygons, despite the importance of efficiently supporting this task.

Motivated by this fact, we focus on efficient and accurate cardinality estimation for intersection queries on polygons in Chapter 4. First, we tried to apply the data-driven design in Chapter 3 to the estimation for polygons. Unfortunately, it is impractical because a polygon dataset usually has a complex data distribution. Data-driven methods using only light-weight models cannot provide a good approximation and they are not suitable for this variable and relatively high dimensional case [62]. Second, we want to guarantee a fast estimation, which is not provided by data-driven methods using complex learning models. Therefore, we adopt a query-driven design with a training data generator to generate sufficient training queries with a near-uniform cardinality

distribution. To make the learning model compatible with the polygons of variable sizes, we propose an adaptive transformation method to transform polygons to a fixed size.

1.4 Organization

This thesis consists of five chapters. The organization of the remaining parts is as follows:

In Chapter 2, we investigate the cardinality estimation in the one-dimensional case and find it helpful to the further study of the spatial data case. In Section 2.1, we analyze the importance of the one-dimensional case and show our idea derived from the learned index. We define the problem and introduce related work in Section 2.2. We introduce the structure of our one-dimensional cardinality estimator in Section 2.3. We give an overview of our solution and show the efficient estimation flow. The experimental results in Section 2.4 demonstrate the effectiveness of our estimator. We discuss related work in Section 2.5. Furthermore, we conclude the work in Section 2.6.

In Chapter 3, we focus on cardinality estimation in the dynamic spatial data setting. We introduce the setting and discuss the challenges brought by it in Section 3.1. Section 3.2 gives the problem definition and shows the limitations of existing methods. We discuss why they cannot support frequent updates and the latent paths leading to an updatable design. We introduce our solution SAFE in Section 3.3. Starting with the overall framework, we present the hierarchy of regression models that supports frequent data updates. Then we show the estimation flow which has a stable fast speed. We elaborate on the details of the update flow and show the essence of the fast update speed coming from the structure design. The experimental results in Section 3.4 demonstrate the capability of SAFE for frequent data updates. We introduce related work and analyze their limitations in Section 3.5. We summarize this work in Section 3.6.

In Chapter 4, we estimate the cardinality of intersection queries on complex spatial polygons. We analyze why the existing solutions are inaccurate for polygons. Besides, we discuss the incapability of existing learned methods, and these contents are shown in Section 4.1. Section 4.2 reviews related work of this topic. We describe our proposed solution PolyCard in Section 4.3. It is a query-driven design, which avoids approximating the complex data distribution of polygons. We show how we transform the polygons of variable sizes to a fixed size and make them compatible with a learning-based design in detail. Also, we propose a training data generator and show its effectiveness to help the

training converge. The experiment results are shown in Section 4.4. The accuracy and efficiency of PolyCard demonstrate its capability for the case of complex polygons. We discuss related work in Section 4.5 and conclude this work in Section 4.6.

In Chapter 5, we summarize this thesis and discuss future works.

Chapter 2 is based on our work published in [44]. Chapter 3 is based on our works published in [45, 47, 48]. Chapter 4 is based on our work published in [46].

Chapter 2

One-dimensional Learned Cardinality Estimation by Light-Weight Regression Models

2.1 Introduction

As we introduced in Section 1.1, cardinality estimation is a fundamental problem in query optimization. Recently, proposed cardinality estimation methods mainly use deep learning models to map given query predicates to an estimated cardinality of query results because they can handle complex non-linear relationships better than traditional methods [35, 88].

However, existing learned cardinality estimation methods focus on multi-dimensional data [100, 110]. They ignore the most fundamental case, namely one dimension. One-dimensional data and queries are essential in database management systems (DBMSs). They support basic operations such as managing user IDs and queries according to timestamps. Cardinality estimation on one-dimensional data can help estimate execution costs and better plan these operations [12]. For example, traditional histogram-based and sampling-based methods are both extended from one dimension to multiple dimensions [34, 41]. The execution time of queries provides an upper time limit for cardinality estimation. As shown in Table 2.5, it takes only tens to hundreds of microseconds to obtain an exact cardinality. Existing learned methods typically require milliseconds for estimation due to their designs for more complex settings and the use of complex

deep neural networks. Consequently, developing a light-weight learned structure for one-dimensional cardinality estimation remains an unresolved challenge.

In the field of artificial intelligence for databases (AI4DB), indexes receive benefits from machine learning [58]. The learned index is first proposed for point and range queries on one-dimensional data by using light-weight learning models [53]. Learned indexes perform much better than traditional indexes, such as B-trees [52], because learning models can avoid time-consuming traverses in tree structures. This observation suggests that light-weight learning models can improve cardinality estimation for one-dimensional range queries, especially considering the estimation time. However, whether it is practical to apply light-weight learning models to cardinality estimation remains a question.

Contribution. To investigate the above research question, we make the following contributions.

(i) A light-weight learned cardinality estimation solution. We design a prototype of light-weight learning models for one-dimensional cardinality estimation in Section 2.3, to compare with existing techniques.

(ii) Empirical evaluations. We conduct experiments by using four real datasets in Section 2.4, (i) to see the performances of the learned and non-learned cardinality estimation methods and (ii) to confirm whether a light-weight learned approach is promising for one-dimensional cardinality estimation. These two evaluations are the main objective of this chapter. We analyze their strong and weak points.

To summarize, the learned method is promising, i.e., its estimation speed is the fastest and its error is small enough. Furthermore, we expect these light-weight models can be used for the multi-dimensional (spatial data) case. Also, our insights would support implementations and selections of one-dimensional cardinality estimation methods for practical DBMSs.

Organization. The rest of this chapter is organized as follows. Section 2.2 introduces the background and existing methods for one-dimensional cardinality estimation. Section 2.3 presents the detailed solution of our proposed learned estimator for the one-dimensional case. Our experimental results are reported in Section 2.4. We discuss related work in Section 2.5 and conclude this chapter in Section 2.6.

2.2 Preliminary

In this section, we define the problem of this chapter and then introduce existing solutions for one-dimensional cardinality estimation.

2.2.1 Problem Statement

Consider a dataset D consisting of sorted one-dimensional data with unique integer keys. A range query predicate Q is specified as a central key K and a range R. The query result is the set of records whose keys fall into $[K-R, K+R]$. Here, the goal of cardinality estimation is to estimate the number of records from D satisfying the query predicate. The estimated cardinality is denoted by $\widehat{card(Q)}$ and the real cardinality is denoted by $card(Q)$. (Another equivalent concept of cardinality is *selectivity*, and it represents the percentage of records satisfying the query predicate [113].)

2.2.2 Existing Methods

Sampling. This is the most straightforward approach to estimate the cardinality of a given range query. The estimated cardinality is obtained by scaling the cardinality on the samples. The performance of sampling is directly related to the number of samples (more samples yield a better accuracy but incur a higher computational cost).

Histogram. Histogram-based approaches build a histogram on a given dataset and sum up the counts of buckets intersected with a given query predicate as estimated cardinality [18]. Histograms have a long history of being used to solve these estimation tasks [41, 101]. For one-dimensional cases, there are two classical histograms: equal-width and equal-depth. A histogram is essentially a group of linear models, where each bucket is a linear model with a slope of the number of records divided by the width. This *one-level* histogram usually uses a higher computation cost than existing hierarchical learning models [53], because it needs hundreds of add operations to sum up the counts of buckets.

Direct calculation. The cardinality of range queries on one-dimensional data can be easily estimated by two-point queries because the records in one-dimensional datasets are generally sorted according to keys.

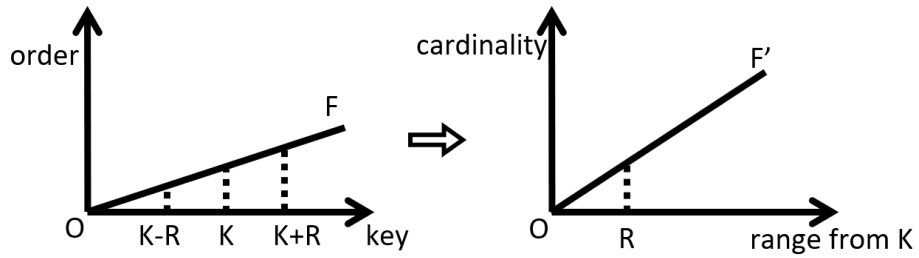


Figure 2.1: The learned index approximates the CDF to obtain the orders of the keys. The approximated CDF, F, can be used to estimate the cardinality of range queries. Suppose a range query is represented by a central key K and range R, the cardinality of the range query can be estimated by the orders of keys $(K-R)$ and $(K+R)$. The relationship between cardinality and the range started from the central key K can be approximated by F'.

More concretely, we can obtain the exact cardinality after computing the orders of records with $K-R$ and $K+R$ as keys in the dataset for the query Q:

$$card(Q) = Order(K+R) - Order(K-R) + N,$$

where $Order(X)$ indicates the order of the first record whose key is greater than or equal to X in the sorted dataset. N is 1 if there exists a record whose key equals $K+R$ in the dataset. Otherwise N is 0. Hence, the computation cost is equivalent to the cost of the two-point queries. The computational efficiency can be improved by using indexes.

2.3 Solution

2.3.1 Main Idea

We derive our idea from a learned index structure, Recursive Model Index (RMI) [69]. RMI is the first and state-of-the-art learned index for the *search* problem on one-dimensional data. It stacks light-weight learning models to approximate the cumulative distribution function (CDF) and then computes the position of a given key in a sorted array according to the CDF. Here the term "CDF" means the function mapping keys to their corresponding positions in an array.

As shown in Figure 2.1, the approximated CDF, F, which maps keys to the corresponding positions, can also be used for cardinality estimation. The mapping done by

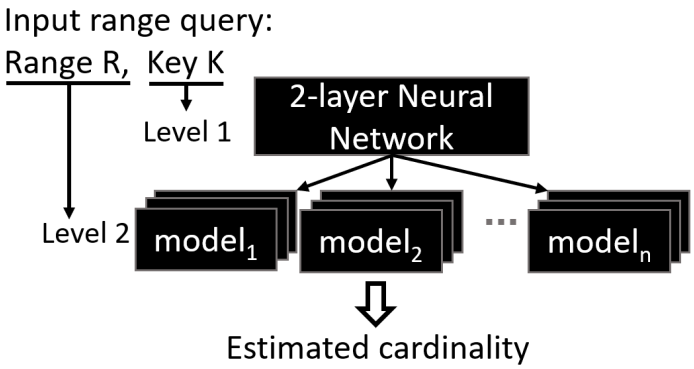


Figure 2.2: The prototype learning model for one-dimensional cardinality estimation has a two-level hierarchical structure. The first level directs to models in the second level according to a given query. The second level is responsible for cardinality estimation.

indexes has an intrinsic relationship with the mapping from range queries to estimated cardinality. We can approximate the relationship between cardinality and the range starting from K by F':

$$F'(R) = (F(K+R) - F(K-R))/2.$$

RMI usually adopts light-weight models like linear models to approximate F and shows good performance. Therefore, we also employ linear models to approximate F'.

2.3.2 Design

We design a cardinality estimation by refining the original RMI based on our observation in 2.3.1. As shown in Figure 2.2, this model has a two-level hierarchy. For an input query, the first level directs to a specific model in the second level, and models in the second level then predict the cardinality. We use a two-layer neural network in the first level. In the second level, we organize many multi-scale linear models into an array. Each multi-scale linear model is responsible for a subset of the original dataset.

For the first level, another choice is a linear model with less computation cost than a neural network. However, choosing a linear model will result in a more skewed distribution of subsets for the second level [65], thus negatively affecting performance and training. The neural network is a better choice. It is hard for a simple linear model in the second level to cover range queries with totally different scales of selectivity.

Therefore, we train some models with different scales for each subset in the second level[1].

During the estimation procedure, the first level takes the central key K of the given query as input and directs it to the corresponding model in the second level. The selected second-layer model estimates the cardinality according to the query range.

From the recent success of learned indexes, we expect that this learning model works well, i.e., it will provide high efficiency and accuracy. We report its practical performance in the next section.

2.4 Experiments

This section reports our experimental results. All experiments were conducted on a server with an Intel Xeon Gold 6254 @3.10GHz CPU and 768GB RAM, running Ubuntu 18.04 LTS. The evaluated methods were implemented in Python with a single thread. We implement the neural network model with the PyTorch library and conduct training and estimation on CPU.

2.4.1 Setup

Datasets. We used four real-world datasets with different distributions from the SOSD benchmark [69]. These datasets have recently been used for evaluating learned index works [19, 87]. Each of them contains 200 million 64-bit key/value records:

- amzn: book popularity data from Amazon.
- face: Facebook user IDs sampled randomly.
- osm: cell IDs from Open Street Map.
- wiki: timestamps from Wikipedia.

Figure 2.3 plots CDFs of the four datasets. Zoomed regions show small segments of datasets. The face dataset contains tens of outliers with much larger keys than the others.

[1]The selectivity is application dependent. Hence, applications can train this model by specifying the range of selectivity.

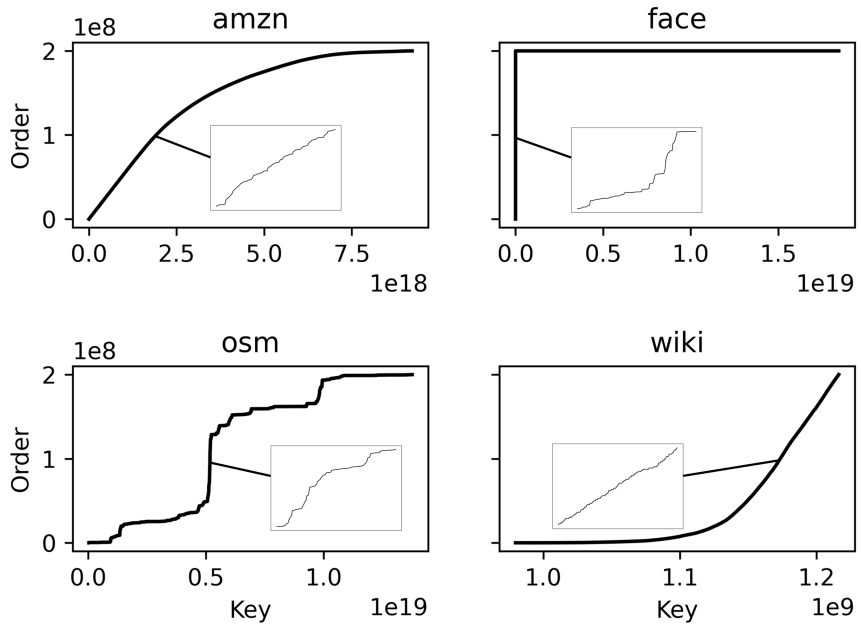


Figure 2.3: CDFs of evaluation datasets. Zoomed regions show plots of 200 consecutive keys.

Error metrics. We evaluate the Mean Absolute Percentage Error (MAPE) and q-error to see the accuracy. These errors are widely used for cardinality estimation problems [88, 100, 104] and are respectively defined as:

$$\text{MAPE} = \left| \frac{\widehat{card(Q)} - card(Q)}{card(Q)} \right|$$

and

$$\text{q-error} = \frac{\max\left(\widehat{card(Q)}, card(Q)\right)}{\min\left(\widehat{card(Q)}, card(Q)\right)}.$$

Evaluated methods. We evaluated the methods listed in Table 2.1. In the case of sampling methods, we estimated the cardinality based on the calculated cardinality of sampled datasets by binary search. An equal-depth histogram has the same number of records in each bucket. For the direct calculation method, we built an RMI index on the original dataset. Compared with classical B-Trees [17], RMI shows better computational and space costs [69]. Recall that direct calculation returns the exact cardinality, so we measured the computation time and model size for direct calculation. For the learned

Table 2.1: Overview of evaluated methods

Methods	Descriptions
LC	The learning model designed in Section 2.3
SA2	Sampling 1×10^{-2} records of original dataset
SA3	Sampling 1×10^{-3} records of original dataset
SA4	Sampling 1×10^{-4} records of original dataset
HM	An equal-depth histogram having 1×10^{5} buckets
DC	Direct calculation using RMI as the index

Table 2.2: Model sizes of evaluated methods (MB)

Methods	LC	HM	SA2	SA3	SA4	DC
Model sizes	1.6	0.8	16	1.6	0.16	1.6

method, we used a two-layer neural network with 16 neurons in the hidden layer for the first level. We set the number of models in the second level to 1×10^{5} to leverage the storage cost and accuracy. Table 2.2 shows the model sizes of all methods. We set the sizes (space consumptions) of LC, HM, SA3, and DC to be almost the same.

Training of LC. For the first level, we assume that a given dataset is equally divided into subsets for models of the second level. We composed keys and corresponding IDs of models into pairs as the training data. After the training of the first level was done, we partitioned the original dataset according to the output of the first level. For the second level, we trained the group of models sequentially. We prepared training data of multiple scales for each model. For each scale, we generated 1000 query predicates (K,R) and calculated the true cardinality as the training data.

The keys and IDs were both scaled to a maximum of 100 during the training and evaluation. To train the neural network of the first level, we used Adam as the optimizer [50]. We set the learning rate to 1×10^{-4}. It took two epochs to finish the training. The loss remained large when the network was trained on osm, because a two-layer neural network cannot approximate the skewed distribution well. We divided the datasets into subsets for models in the second level according to the predicted results of the trained neural network. Therefore, the high loss of the neural network did not matter. We fitted the linear models of the second level to the training data by using non-linear least squares. In the case of the face dataset, there were tens of outliers that made it hard for training. We removed the outliers when we scaled the keys for model training.

Table 2.3: Estimation errors for cardinality estimation using queries with a selectivity of around 10^{-4}.

Datasets	amzn						face					
Metric	MAPE			q-error			MAPE			q-error		
	50th	95th	99th	50th	95th	99th	50th	95th	99th	50th	95th	99th
LC	5.5×10^{-3}	1.2×10^{-1}	3.3×10^{-1}	1.0055	1.13	1.34	3.2×10^{-2}	9.0×10^{-2}	3.6×10^{-2}	1.032	1.097	1.14
HM	6.3×10^{-3}	1.0×10^{-1}	2.0×10^{-1}	1.0063	1.11	1.22	1.4×10^{-2}	4.4×10^{-2}	6.2×10^{-2}	1.014	1.045	1.064
SA2	7.8×10^{-3}	4.5×10^{-2}	8.2×10^{-2}	1.0078	1.045	1.082	8.9×10^{-3}	1.3×10^{-2}	1.6×10^{-2}	1.0089	1.014	1.016
SA3	8.2×10^{-2}	3.8×10^{-1}	1.0	1.087	1.46	3.1×10^{2}	9.0×10^{2}	1.4×10^{-1}	1.6×10^{-1}	1.098	1.16	1.19
SA4	1.0	2.6	6.1	1.4×10^{3}	5.4×10^{3}	5.4×10^{3}	1.0	1.5	1.6	1.9×10^{4}	2.5×10^{4}	2.7×10^{4}
Datasets	osm						wiki					
Metric	MAPE			q-error			MAPE			q-error		
	50th	95th	99th	50th	95th	99th	50th	95th	99th	50th	95th	99th
LC	1.1×10^{-1}	3.4×10^{-1}	8.9×10^{-1}	1.11	1.37	1.96	6.5×10^{-3}	3.0×10^{-2}	1.2×10^{-1}	1.0065	1.031	1.12
HM	1.2×10^{-3}	5.6×10^{-2}	3.2×10^{-1}	1.0012	1.058	1.41	1.0×10^{-3}	1.0×10^{-2}	7.0×10^{-2}	1.0010	1.010	1.072
SA2	1.1×10^{-3}	2.8×10^{-2}	1.3×10^{-1}	1.0011	1.029	1.14	1.2×10^{-3}	6.8×10^{-3}	2.4×10^{-2}	1.0013	1.0068	1.024
SA3	1.1×10^{-2}	2.8×10^{-1}	1.0	1.011	1.33	6.54	1.2×10^{-2}	7.1×10^{-2}	2.6×10^{-1}	1.012	1.072	1.32
SA4	1.1×10^{-1}	2.4	1.3×10^{1}	1.12	10.88	2.3×10^{4}	1.2×10^{-1}	1.0	1.34	1.12	4.5×10^{2}	5.9×10^{3}

Workloads. For query predicates (K,R), first, we selected scales of selectivity[2] for query range R from $\{10^{-5},10^{-4},10^{-3}\}$. Then we randomly generated 1000 pairs of (K,R) for each scale of selectivity. The keys generated are limited within the range of existing keys in the datasets. The generated R was floated around the selected selectivity with possible magnitudes within $(10^{-1},10^{1})$.

2.4.2 Overall Performance

Table 2.3 shows the estimation errors on all datasets and queries with a selectivity of around 10^{-4}. The 50th/95th/99th values show the corresponding percentile errors. Table 2.4 shows estimation errors on the wiki dataset with selectivities of around 10^{-5}, 10^{-4}, and 10^{-3}.

The MAPE errors and q-errors do not show significant differences. Most MAPE errors are much smaller than 1, with zeros after their decimal points. According to the definitions, the similarity of MAPE errors and q-errors is easy to understand. However, for SA4, whose selectivity of sampling hardly matches the queries' selectivity, its q-errors are extremely large.

Exp-1 (LC vs. SA). Sampling methods can easily find a trade-off between accuracy and storage cost. According to the errors shown in Tables 2.3 and 2.4, we study the following: (i) Generally, LC in the current setting has a competitive accuracy over SA3. (ii) The 50th errors of LC are usually slightly larger than those of SA3, whereas LC has smaller 95th and 99th errors. This means that the distribution of LC's errors is more

[2]We omit the term "1×".

Table 2.4: Estimation error comparison at different scales of range queries using the wiki dataset.

Metric	MAPE								
	50th			95th			99th		
Selectivity	10^{-3}	10^{-4}	10^{-5}	10^{-3}	10^{-4}	10^{-5}	10^{-3}	10^{-4}	10^{-5}
LC	2.7×10^{-2}	6.5×10^{-3}	1.2×10^{-2}	5.4×10^{-2}	3.0×10^{-2}	8.7×10^{-2}	6.1×10^{-2}	1.2×10^{-1}	$3.0\times10^{-}1$
HM	1.1×10^{-4}	1.0×10^{-3}	1.0×10^{-3}	1.1×10^{-3}	1.0×10^{-2}	7.1×10^{-2}	8.4×10^{-3}	7.0×10^{-2}	$2.1\times10^{-}1$
SA2	1.3×10^{-4}	1.2×10^{-3}	1.2×10^{-2}	6.6×10^{-4}	6.8×10^{-3}	6.7×10^{-2}	2.6×10^{-3}	2.4×10^{-2}	$2.7\times10^{-}1$
SA3	1.3×10^{-3}	1.2×10^{-2}	1.2×10^{-1}	6.5×10^{-3}	7.1×10^{-2}	1.0	2.7×10^{-2}	2.6×10^{-1}	1.4
SA4	1.3×10^{-2}	1.2×10^{-1}	1.0	6.6×10^{-2}	1.0	3.6	2.2×10^{-1}	1.3	6.1
Metric	q-error								
	50th			95th			99th		
Selectivity	10^{-3}	10^{-4}	10^{-5}	10^{-3}	10^{-4}	10^{-5}	10^{-3}	10^{-4}	10^{-5}
LC	1.03	1.0065	1.012	1.054	1.031	1.09	1.06	1.12	1.35
HM	1.00011	1.0010	1.0098	1.0011	1.010	1.075	1.0085	1.072	1.24
SA2	1.00013	1.0013	1.012	1.00066	1.0068	1.068	1.0026	1.024	1.32
SA3	1.0013	1.012	1.13	1.0065	1.072	33	1.027	1.32	5.7×10^{2}
SA4	1.013	1.12	1.9×10^{3}	1.068	4.5×10^{2}	3.8×10^{3}	1.25	5.9×10^{3}	4.4×10^{3}

even than that of SA3. (iii) Sampling methods have larger errors for a smaller selectivity. In contrast, the variation of LC's errors for different scales of selectivity is more stable. The MAPE errors of the three sampling methods are linear to their sampling scales. Sampling methods with different scales can be treated as a trade-off between accuracy and storage cost.

Exp-2 (LC vs. HM). Tables 2.3 and 2.4 show that HM generally returns smaller errors than LC. Similar to SA, HM shows degraded performance with smaller selectivity of queries. HM has a more significant advantage over LC for queries with a selectivity of 10^{-3}.

Exp-3 (Performances on osm). LC has the worst accuracy on osm among all datasets, whereas the other methods do not show decreased performances on osm. This is mainly attributed to the distribution of osm. Figure 2.3 shows that osm is the most skewed. The small segment shown in the zoomed region of osm shows that the distribution lacks local structures, making it difficult for LC to learn the CDF. Similar situations are met in indexing works [69]. HM and the sampling methods are not bothered a lot by the distribution of osm, because they do not need to learn the distribution.

Exp-4 (Performances on amzn, face, and wiki). The CDFs of amzn and wiki show very smooth distributions. The zoomed small segments of amzn and wiki also look similar. Considering zoomed segments of face, face also lacks local structures, similar to osm. As a result, LC has more significant errors on face compared with its errors on amzn and wiki.

Table 2.5: Estimation time comparison for different methods on different datasets (microseconds)

Datasets	LC	HM	SA2	SA3	SA4	DC
amzn	15.9	139.2	44821.5	1176.6	129.8	77.6
wiki	15.9	145.0	42313.6	1162.3	129.9	91.8
face	16.9	148.4	44743.2	1230.1	138.5	93.0
osm	17.6	153.4	47847.8	1271.0	143.1	196.6

Exp-5 (Estimation time). Table 2.5 shows the average estimation times on different datasets with a selectivity of 10^{-4}. LC is much faster than the other methods. DC takes less time than HM and SA4 on amzn, face, and wiki. When RMI is used as the index, the time cost of a point query consists of the inference time (evaluated by RMI) and the search time (for searching around the predicted order). As DC uses two-point queries to estimate cardinality, it is reasonable that DC needs a longer time than LC. We see that LC, HM, and SA have stable times on different datasets. DC needs more time on osm than the other datasets. The point queries on osm incur a long search time [69]. HM scans the border values of buckets, and this scanning contributes to a major part of the time costs. The sampling methods with large samples involve many record accesses, thereby SA2 and SA3 are very slow.

2.4.3 Detailed Analysis

Overall, LC has the best computational efficiency. Its errors are a bit larger than those of the other models but is absolutely small. DC provides the true value of cardinality, showing the best accuracy. At the same time, its estimation time normally outperforms those of the sampling methods and HM. Consequently, LC and DC are the options. However, for the *estimation* purpose, we consider that LC is better suited, as it yields high efficiency and small error. The main advantage of the prototype learned method LC is its computational efficiency. The advantage comes from its structure, which is based on light-weight learning models.

As mentioned in Section 2.2, HM is a group of linear models. LC, HM, and DC with RMI all make use of models. The model differences lead to differences in their performances. DC calculates cardinality by finding the orders of two border keys of the query range. It involves twice the computation cost of RMI. Compared with DC, LC infers the difference between the orders of two border keys instead of inferring the

orders. Hence, LC reduces the computation cost to one pass of the learning models. Models of HM take charge of counting their buckets. When the query range covers multiple buckets, which is a typical case, HM accumulates counts of all covered buckets. Inferences by models need to be executed only on the border buckets intersected by the query range. The errors in HM come from estimations of the border buckets.

The training cost of LC consists of two parts. (i) For model training, LC shares similar costs with the original RMI model, which DC uses. In our evaluation, to train the prototype learning model on a dataset consisting of 200 million records, it took about a half hour for a simple neural network and 10 minutes for 1×10^5 linear models. (ii) For the preparation of training data, LC has two ways, passive and active. The query results produced by DBMSs can be utilized for training. However, this passive way cannot guarantee sufficient amounts of data to train the models. Besides, an active collection of training data for learning models can be expensive [35]. LC needs a longer time than DC to collect training data because DC does not need to obtain true cardinality. However, both training data collection and model training are done *once*, so the offline cost of LC would not be a drawback in practice.

2.5 Related Work

Learned indexes. Learned indexes have recently been proposed to use learning models to replace traditional index structures like trees [53, 64, 87]. They avoid the time-consuming traverses in a tree structure and achieve fast query processing. RMI is the first proposed learned index and preserves the state-of-the-art performance in the one-dimensional case [53, 66, 69]. Our idea of approximating the data distribution in this work also derives from RMI as introduced in Section 2.3. Flood [74] and Tsunami [21] are two latest works on multi-dimensional indexes. They divide the data space into buckets on a selected dimension and consider the data distribution during the division, which can optimize the number of buckets that queries access and further improve the query efficiency. Focusing on different data types, several learned indexes have been proposed for the strings [107] and variable-length keys [15]. These works extend the application scenarios for learned indexes and we can get inspiration about how to approximate data distribution for cardinality estimation.

Learned cardinality estimation. Learned cardinality estimation methods use learning models to estimate the cardinality. In this chapter, we focus on light-weight cardi-

nality estimation methods for the one-dimensional case and achieve fast estimation. MSCN [51] is an early and relatively fast learned cardinality estimator. It is a query-driven method and relies on generated training queries with corresponding true cardinalities. In particular, MSCN uses multilayer perceptrons to learn the mapping from queries to corresponding cardinalities. Successors mainly focus on more complex settings like high dimensions and large joins across multiple labels [13, 54, 61], and they are not suitable for our task. As discussed in Section 2.1, existing works have not studied the one-dimensional case and we investigate the problem in this chapter.

2.6 Conclusion

In this chapter, we addressed an unanswered question: *Are learned methods suitable for one-dimensional cardinality estimation?* We designed a prototype of a light-weight learned structure for one-dimensional cardinality estimation. We empirically evaluated the performances of the proposed learned method and existing methods to investigate their advantages and disadvantages. The evaluation results answer the question: A light-weight learned method can achieve fast and accurate one-dimensional cardinality estimation.

Though results in Section 2.4 have shown the effectiveness of our proposed LC, we are seeking further improvements. If we treat LC as a baseline, better methods for one-dimensional cardinality estimation should have smaller estimation errors and competitive computational efficiency. A possible way is to add some extra models to LC to achieve more accurate estimations. However, complex models are not suitable because they incur high time costs. Therefore, the issue is how to make LC more accurate while retaining simple structures. If we seek to optimize models compared with DC, we will need to cut down its computation time. With RMI as an index, DC consumes about 20 multiplication operations (varying among the different models used in RMI) and tens of add operations. Less computation than DC means that only several multiplication operations are allowed. This is a new research challenge.

We expect that our investigation in this chapter can further inspire our study on the multidimensional spatial data case (mentioned in Section 2.1). The point here is, light-weight regression models can provide extremely fast inference speed compared with commonly used deep neural networks for cardinality estimation. From the structures of our proposed LC in Section 2.3, we can observe that a combination of light-weight

models can easily approximate the data distribution. If we treat regression models as a basis and design a cardinality estimator for spatial data, we can expect that the estimator will easily achieve fast estimation speed, which is desired for spatial applications as discussed in Section 1.1.

Chapter 3

SAFE: Sampling-Assisted Fast Learned Cardinality Estimation for Dynamic Spatial Data

3.1 Introduction

We have introduced spatial data in Section 1.1 and one of the fundamental technologies for spatial data is *range queries*, which find spatial points within a specified range. For example, navigation/map applications display Point-of-Interest information in specific areas, and monitoring systems count the number of people on each floor. Due to widespread usages of spatial applications, the volume of spatial range queries continuously increases [23, 90]. Reducing the latency of range queries is essential to provide comfortable applications for users. Cost-based query optimization is often used to obtain the most efficient query plan. Cardinality estimation, which estimates the result size of a given query, is one of the most important core functions for query optimization. Because spatial data applications usually need to deal with a significant number of range queries, the accuracy and efficiency of cardinality estimation for range queries are directly related to the efficiency of dealing with a workload of range queries [62, 85, 86].

It is well known that spatial datasets are generally dynamic, i.e., they allow frequent updates, including insertions and deletions [5, 6, 93] as mentioned in Section 1.2. Therefore, cardinality estimators should be updatable according to dataset updates. This is a trivial requirement, as, if a given dataset is subject to updates, static estimators

cannot consider the latest data distribution and the estimation results will be inaccurate. This chapter addresses the cardinality estimation problem on dynamic spatial data and designs a fast, accurate, and efficiently updatable cardinality estimator.

Motivation and challenge. Many studies proposed efficient and accurate cardinality estimation approaches. As introduced in Section 1.2, recent results of *learned* cardinality estimation methods [22, 35, 40, 57, 68, 71, 75, 98, 100] show remarkable improvements compared with traditional methods, such as histogram-based methods [18]. Learned cardinality estimation methods can be categorized into query-driven models (e.g., [33, 97]) and data-driven models (e.g., [71, 78]).

Query-driven models are built on training queries and map a training query to its true cardinality. Most of them use deep learning to capture the relationship between queries and cardinalities [22, 97, 78]. Although this approach is effective when datasets are static and query workloads are known, the nature of "relying on training queries" is not appropriate for dynamic data and unknown query workloads. If query-driven models need to deal with dataset updates and unknown query distributions, they need to collect training queries and compute their true cardinality, which leads to laborious costs.

On the other hand, data-driven models usually learn the distribution of a given dataset mainly through deep neural networks [38, 71, 109]. They hence take a long time to build a model (at least several "minutes" of training time are required for large datasets) [35, 110]. This property limits to only static data, as they cannot support fast model updates (i.e., re-training is required whenever datasets are updated). LHist [62] is a state-of-the-art cardinality estimator for spatial data and does not use complex models like neural networks, but it also does not support model updates. Traditional methods, such as histogram and sampling, also can be considered as data-driven models, because they generate some statistics from original datasets. These traditional methods are employed as components of major DBMSs such as PostgreSQL [102]. Although their structures are easy to build, their accuracy is lower than those of learning-based models [33, 78].

We here summarize the challenges that need to be solved to deal with dynamic spatial data effectively. (1) **Stable performance**: The estimation accuracy and speed should be stable even if a given dataset is updated. (2) **Fast estimation**: Cardinality estimation is frequently called, so fast estimation is desirable (e.g., microsec-level). (3) **Update-friendly**: Estimation models should be easily (efficiently) updatable according to data updates. As seen from the above discussion, existing approaches cannot satisfy

these requirements at the same time. This motivates us to design a new cardinality estimator that satisfies these requirements.

Contribution. To solve the above challenges, we propose SAFE, a sampling-assisted fast and accurate learned cardinality estimator. Specifically, we solve the cardinality estimation problem on dynamic spatial data by making the following contributions:

(i) SAFE. Our estimator specializes in solving frequent data updates while keeping efficient and accurate cardinality estimations. It enables fast training by a sampling strategy using data partitioning based on a quad-tree. Our key observation is that we can accelerate the model updates from two perspectives: (i) updating only the necessary models and (ii) fast training.

As for the first perspective, existing cardinality estimation methods typically need to update/train their whole structures/models from scratch whenever they receive an update. To avoid this issue, we employ 2-tier regression models, based on our insight that each model can be updated independently. In addition, there are various regression models, and their performances depend on the data distributions. To make an appropriate selection among these models, we design a cost model that considers accuracy, estimation time, and construction time. This cost model helps select a model with a balanced trade-off between time and accuracy.

Next, as for fast training, we propose a sampling strategy using data partitioning based on a quad-tree. A smaller sample size alleviates the training time while learning the data distribution as samples follow the data distribution. Thanks to our idea of these two perspectives, SAFE provides stable performance.

(ii) Incremental model update strategy for fast dealing with data updates. To enable fast model updates, we propose an incremental model update algorithm for SAFE. The SAFE structure can avoid model updates from scratch, and, for a given update (e.g., point insertion), only several regression models are affected. The SAFE structure enables finding such models quickly, and only affected models are updated.

(iii) Experiments. We conduct experiments on both real-world and synthetic datasets. Our results demonstrate that SAFE outperforms state-of-the-art cardinality estimation methods and can efficiently handle dataset updates.

Organization. The rest of this chapter is organized as follows. Section 3.2 provides preliminary information to present our proposed method. Section 3.3 presents our efficiently-updatable learned estimator SAFE. Our experimental results are reported in

Table 3.1: Overview of symbols frequently used in this chapter

Symbol	Description
P	Spatial dataset
p	Spatial point in P
N	Number of points in P
R	Range query
l	Left-bottom point of a query range R
u	Right-top point of a query range R
$card(R)$	True cardinality of R
$\widehat{card(R)}$	Estimated cardinality of R
f_x	Regression model for the x-dimension
f_y^i	i-th regression model for the y-dimension

Section 3.4. Related work is reviewed in Section 3.5. Finally, we conclude this chapter in Section 3.6.

3.2 Preliminary

In this section, we first define our problem. Then, we provide preliminaries on learning-based cardinality estimation. Table 3.1 summarizes the symbols used throughout this chapter.

3.2.1 Problem Statement

Let P be a dataset of N two-dimensional (i.e., geospatial) points, and each point p in P is represented as $p = (x, y)$.

Query predicate. This chapter considers two-dimensional orthogonal (axes-parallel) range queries. Let R be the specified range of a given orthogonal range query, and the orthogonal range *reporting* query outputs all points in P that overlap R. Notice that a range R can be represented by its left-bottom point l and right-top point u. For conciseness, we use "range queries" to denote orthogonal range queries. Also, R is used to denote a range query hereinafter.

Cardinality of a range query R is the number of points in P that overlap R. We use $card(R)$ to denote the cardinality of R. The notion of *selectivity* is close to cardinality and represented as $\frac{card(R)}{N}$.

We address the cardinality *estimation* problem, which is to infer $card(R)$. We use $\widehat{card(R)}$ to denote an estimated cardinality of R. Our objective is to compute $\widehat{card(R)}$ efficiently and accurately. Notice that computing cardinality by using existing range search (reporting) algorithms (e.g., R-tree) is meaningless (and slow), as applications require only cardinality. Although there exists an $O(\sqrt{N})$ time range *counting* algorithm (i.e., kd-tree [10]), it is still slow, because it requires many point accesses. We therefore consider a learning-based approach, because it achieves $O(1)$ time estimation (the estimation time is dependent only on a model). We assume that the query workload is unknown (i.e., is not given in advance) for generality.

3.2.2 Learning Data Distribution

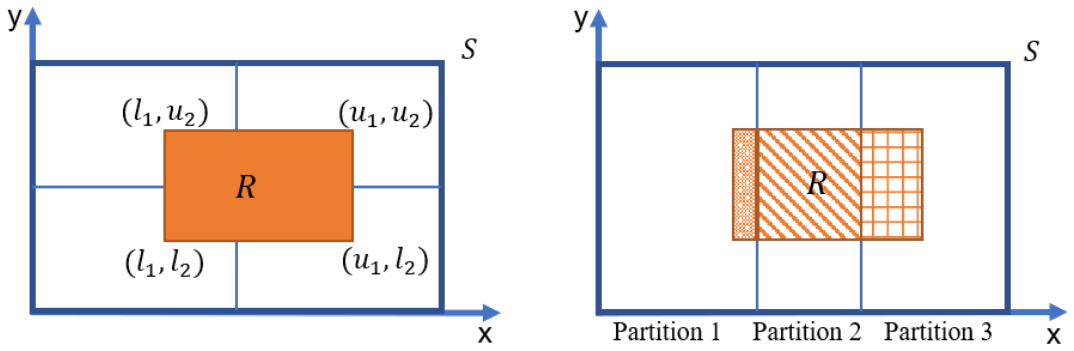
Learned cardinality estimation methods map queries to their cardinality. As introduced in Chapter 2, we can use light-weight regression models to approximate one-dimensional data distribution and estimate the cardinality for range queries. To extend the one-dimensional approach for multi-dimensional cases, the joint distribution of multi-dimensions should be considered. Particularly, the correlations between dimensions need to be captured for cardinality estimation. One existing approach is to approximate the CDF of multi-dimensional data directly. As shown in Figure 3.1(a), MOSE [89] uses the lattice regression model for the approximation:

$$\widehat{card(R)} = f_{MOSE}(u_1,u_2) - f_{MOSE}(u_1,l_2) - f_{MOSE}(l_1,u_2) + f_{MOSE}(l_1,l_2), \tag{3.1}$$

where f_{MOSE} approximates the joint CDF of the two-dimensions[1]. MOSE captures the data distribution well, but it employs a supervised approach, i.e., it requires many (e.g., thousands of) training queries and their true cardinality to train f_{MOSE}. This approach cannot be used for unknown query workloads and dynamic data, as it needs to recompute the true cardinality of each query when the dataset is updated, which is too expensive.

Another way is to approximate the data distribution on *each* dimension. LHist [62] partitions the data space into multi-dimensional grid cells and employs hierarchical regression models, which is shown in Figure 3.1(b). Specifically, LHist approximates the distribution of each dimension based on the one-dimensional histogram. The estimated

[1]For example, $f_{MOSE}(u_1,u_2)$ approximately counts the number of points falling in the rectangle whose left-bottom and right-top coordinates are respectively $(0,0)$ and (u_1,u_2). For details, refer to [89].



(a) MOSE (b) LHist

Figure 3.1: Existing solutions for approximating data distributions. (a) MOSE approximates the overall distribution directly by using the lattice regression model. (b) LHist recursively learns the data distribution of each dimension, similar to an equal-depth histogram.

cardinality is computed by:

$$\widehat{card(R)} = \sum_i f^i_{LHist}(D_i),$$

where $f^i_{LHist}(D_i)$ approximates the cardinality of the i-th partition D_i.

This recursive way can adapt to frequent updates, because each regression model can be trained independently. By selecting only the models that need to be updated, we can avoid training all models from scratch and efficiently keep accurate models even on dynamic data. However, LHist does not have this idea, and it does not support model updates. SAFE implements this new idea and employs 2-tier regression models in an update-friendly manner.

3.2.3 Update Solutions of Existing Methods

We briefly describe the update solutions of existing methods by answering the following two questions.

Can they avoid reconstructions? The most straightforward approach to dealing with updates is reconstruction (i.e., re-training all models). In this case, the update time corresponds to the construction time. For example, histogram-based solutions, e.g., LHist,

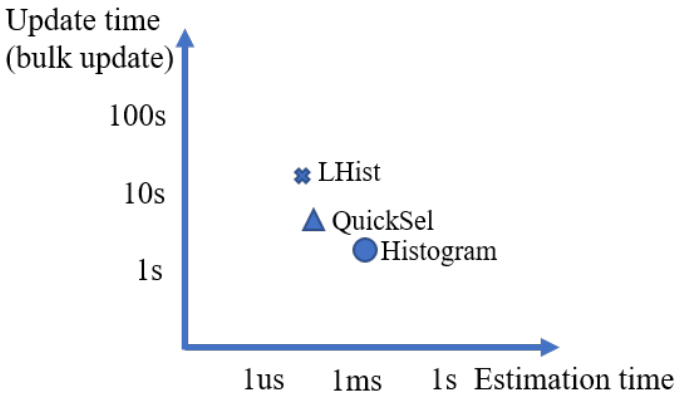


Figure 3.2: Comparison of estimation time and update time of existing cardinality estimation methods on a Gaussian dataset with 20 million points

need to reconstruct their models to deal with dataset updates. Clearly, reconstruction is laborious and cannot deal with frequent updates.

Adapting to data updates continuously is preferable for saving time. QuickSel [78] employs a continuous learning approach to gradually learn from incoming queries. That is, QuickSel does not learn updated data distributions if no new queries are given, so its accuracy becomes worse as it receives more updates. It is possible that QuickSel uses all existing queries to learn the up-to-date data distribution, but it trivially takes a significantly large cost.

What are their time costs? Figure 3.2 compares three existing methods' update and estimation times on a Gaussian dataset with 20 million two-dimensional points. For LHist and Histogram, the update time equals the construction time because they need reconstructions to update models. QuickSel builds its models from scratch. It continuously records incoming queries and true cardinalities. Its update time cost increases as more queries are accumulated. In Figure 3.2, the number of recorded queries is 300 for QuickSel. The update and estimation times of QuickSel increase with more queries, so, when data updates are frequent, the update cost becomes a heavy burden. In particular, learning-based methods (i.e., LHist and QuickSel) incur a long update time, which is not acceptable in the dynamic setting. This is the main issue of existing learning-based cardinality methods. We overcome this challenge and design a solution that achieves fast estimations and updates.

3.3 SAFE

3.3.1 Main Idea

SAFE is a learned cardinality estimator designed for dynamic spatial data. To accurately learn the 2-dimensional CDF, we employ 2-tier regression models, each of which is a light-weight (e.g., polynomial) model. The first tier (resp. second tier) corresponds to the x (resp. y)-dimension[2]. The idea behind employing light-weight regression models is twofold. First, they can infer and be trained quickly, whereas complex models (e.g., deep neural networks) do not have these properties. Second, by dividing the x-dimensional space into disjoint buckets, we can focus on the y-dimensional CDF on each bucket, and the CDF tends not to have a complex curve. More precisely, the first tier model is responsible for dividing the data space into disjoint subspaces based on x-dimensional values, whereas the second tier models are responsible for cardinality estimation.

The accuracy of a learned cardinality estimator becomes worse on dynamic data if models are not updated. As we assume dynamic spatial data with frequent updates (e.g., insertions and deletions), cardinality estimators should be robust against this setting. Then, we come up with two ideas: (i) fast training with small samples and (ii) updating only the necessary models. To implement these ideas, we propose to use "good" samples for accurately training the regression models, and we design the structure of SAFE to identify which model has to be updated easily. With these techniques, we can incrementally update a SAFE, i.e., we update only models that need to be updated even when we receive dataset updates.

3.3.2 Overview

Figure 3.3 shows an overview of SAFE construction. We first obtain samples by using data partitioning based on a quad-tree. The samples are used to quickly train regression models. We then train 2-tier regression models (the first tier corresponds to the CDF on the x-dimension while the second tier corresponds to the CDF on the y-dimension). Note that each cardinality estimation model for the corresponding subspace does not affect the others, thereby it can be trained independently. Figure 3.3(d) illustrates how (i) the data space is divided into multiple (three in the figure) buckets based on the x-dimensional

[2]The order of the x- and y-dimensions is changeable.

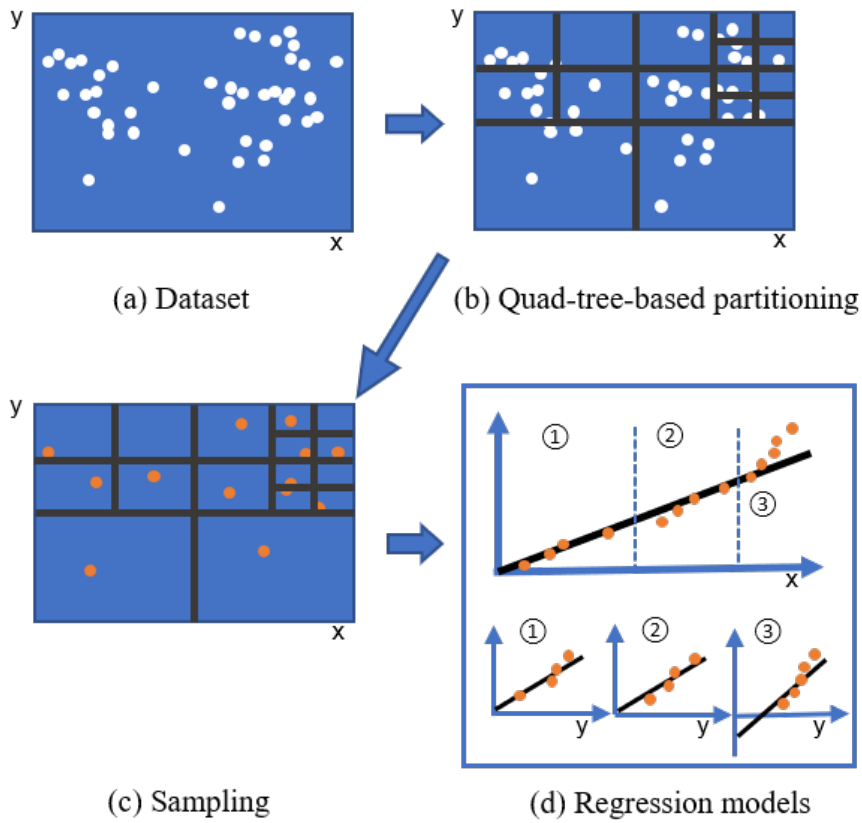


Figure 3.3: Construction of a SAFE. (a) Given a dataset P, (b) we partition P into disjoint subspaces by using quad-tree-based partitioning. (c) We sample a random point from each partition, and (d) train 2-tier regression models by using the samples. The regression model on the x-dimension (the first tier) partitions P into m buckets based on the x-dimensional values, and a regression model is created for each bucket (the second tier).

values and (ii) a model in each bucket is trained while considering the y-dimensional CDF. Algorithm 1 describes the pseudo-code of our construction algorithm.

3.3.3 Sampling by Data Partitioning based on Quad-tree

When a model needs to be updated, fast training is required for dynamic data. To this end, we consider using samples. That is, we train our regression models from samples obtained from a given dataset. This is a simple-yet-effective approach, as this approach makes the training easier and faster. We here note that straightforward random

Algorithm 1: CONSTRUCTION

Input: P (a spatial dataset) and m (the number of buckets)
Output: f_x, $f_y^1, ..., f_y^m$ (regression models)

/* Generate samples */
$P' = \{P_1, P_2, ...\} \leftarrow$ PARTITION-DATASET(P) // based on a quad-tree
$S \leftarrow$ GET-SAMPLES(P') // sample a random point from each partition
/* Train regression models */
$P_x \leftarrow$ a set of x-dimensional values in S
$f_x \leftarrow$ TRAIN(P_x) // learning the CDF curve in x-dimension by least squares
$B_1, B_2, ..., B_m \leftarrow \varnothing$
foreach $p \in S$ **do**
    $B_j \leftarrow B_j \cup \{p\}$, where $j = \lfloor \frac{m}{N} f_x(x_i) \rfloor$
foreach $i \in [1, m]$ **do**
    $P_y^i \leftarrow$ a set of y-dimensional values in B_i
    $f_y^i \leftarrow$ TRAIN(P_y^i) // learning the CDF curve in y-dimension by least squares

sampling is not suitable, as it may fail to sample points in sparse regions, leading to wrong learning.

To avoid this issue, we partition the whole data space into disjoint fine-grained subspaces so that we can sample a point from each subspace. We achieve this by using quad-tree-based partitioning. A quad-tree decomposes a given data space into equal-sized four disjoint subspaces [25]. This is repeated until a given space has a small number of points. As seen from Figures 3.3(b) and (c), this approach does not fail to sample points from sparse regions. We pick one random sample from each partition. Other (virtual) sampling approaches are also considered:

- Averaged point. We compute the average (or median) x-y coordinates of all points in a given partition and use it as a sample. It is expected that this approach can reflect the distribution in the partition.
- Center point: The geometric center of a partition is considered as the partition's coordinates. This is also easy to compute.

We compare the performance differences between these strategies in Section 3.4.2 and confirm that random points are sufficient. Note that we do not take samples from empty partitions.

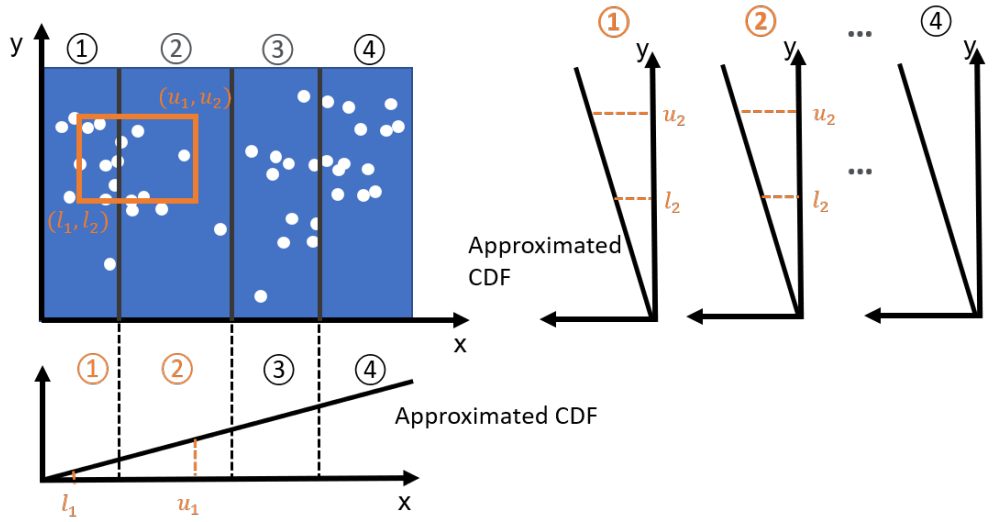


Figure 3.4: Illustration of the two-dimensional layout for regression models. For the x-dimension, a regression model divides the data space into m buckets according to the data values of the x-dimension ($m = 4$ in the figure). Then, regression models are built on each bucket to approximate the y-dimensional CDF. For a query R, the regression model for the x-dimension locates the buckets that need to be accessed. The estimated $\widehat{card}(R)$ is obtained by summing the results from all the accessed buckets.

3.3.4 Regression Models for Cardinality Estimation

Considering data-driven models for cardinality estimation without relying on queries, existing works employ hierarchical regression models. However, they do not provide any ideas of how we efficiently update such models or they cannot avoid re-training all models from scratch [62]. Our structure is designed based on the idea that each regression model can be independently updatable. In addition, we design a cost model to select a suitable model for a given dataset to achieve a good trade-off between accuracy and estimation time.

Layout & estimation. As shown in Figure 3.3(d), we employ 2-tier regression models. For the x-dimension, we divide the data space into m subspaces called buckets, based on a regression model that approximates the x-dimensional CDF. On each bucket, we train a regression model to approximate the y-dimensional CDF curve of points falling in this bucket. The first tier has a single model f_x, while the second tier has m regression models $f_y^1, ..., f_y^m$.

Algorithm 2: ESTIMATION

Input: f_x, $f_y^1, ..., f_y^m$ (regression models) and R (a range query)
Output: $\widehat{card}$

$\mathscr{B} \leftarrow$ GET-BUCKETS(f_x, R) // get a set of all buckets overlapping R
$\widehat{card} \leftarrow 0$
foreach $B_i \in \mathscr{B}$ **do**
    $\widehat{card} \leftarrow \widehat{card} + (f_y^i(u_2) - f_y^i(l_2))$
$\widehat{card} \leftarrow \alpha \times \widehat{card}$ // α is the average number of points in each partition

Table 3.2: Types of regression models (θ_i is a parameter and $z = x$ or $z = y$)

Name	Description
Polynomial	$f(z) = \theta_0 + \theta_1 z + ... + \theta_n z^n$
Spline Linear	$f(z) = \begin{cases} \theta_0 + \theta_1 z & (z \leq \theta_3) \\ \theta_2 z + \theta_1 \theta_3 + \theta_0 & (z > \theta_3) \end{cases}$
Log Linear	$f(z) = \exp(\theta_0 z + \theta_1)$

To estimate the cardinality of a query R, the first tier model is used to find the buckets overlapping R. Figure 3.4 illustrates the estimation flow. Because the model for the x-dimension, f_x, divides the data space into m buckets, the left-most and right-most buckets overlapping R are respectively computed by $\lfloor \frac{m}{N} f_x(l_1) \rfloor$ and $\lfloor \frac{m}{N} f_x(u_1) \rfloor$, where l_1 and u_1 are respectively the left-most and right-most x-dimensional values of R. Then, we estimate the cardinality by considering a set B of all overlapping buckets:

$$\widehat{card(R)} = \alpha \sum_{i \in B} (f_y^i(u_2) - f_y^i(l_2)) \tag{3.2}$$

where l_2 and u_2 are respectively R's bottom and top values in the y-dimension, and α is the average number of points in each partition obtained by the quad-tree-based partitioning. Recall that our regression models are trained on *sample* points (i.e., a small subset of P). Therefore, the cardinality estimated by f_y needs to be scaled up, and α is used to achieve this.

For computing $\widehat{card(R)}$, we access at most $1 + m = O(1)$ regression models, and each model needs $O(1)$ time, see Table 3.2. Therefore, Equation (3.2) needs $O(1)$ time. A pseudo-code of our estimation algorithm appears in Algorithm 2.

Selection of regression models. Regression models are responsible for approximating one-dimensional CDFs, and LHist [62] uses only polynomial models. It is trivial that an appropriate regression model is dependent on the data distributions. Specifically, considering skewed data distributions in real datasets, it is difficult for a simple linear regression model to approximate the CDF curve well. An exponential or polynomial model with high degrees may achieve better results. Therefore, it is better to consider which model we should select, but automatically selecting it is challenging.

To address this task, we propose a strategy to select the most suitable type of regression model. Table 3.2 lists the representative types of regression models, which are candidates for SAFE. We design a cost model that considers the following three parts:

- **Estimation error.** This is a core part we care about when selecting which model to be used. However, it is not practical to obtain estimation errors during construction, and we need query samples to measure them. Instead, we consider the training error of a given regression model, because the training error represents how the model approximates the data distribution. We measure the least square error as the training error of a given regression model.
- **Estimation time.** Fast estimations highly rely on regression models with low computation time. Considering the types of candidates, it is easy to get an expected estimation time by feeding in some default values during construction.
- **Training time.** The dynamic setting asks for frequent and efficient model updates, so training time is also important. Measuring the training time of a given model is trivial.

Then, we measure the cost for a given regression model f as follows:

$$Cost(P, f) = E_{train} + T_{est} + T_{train},$$

where E_{train}, T_{est}, and T_{train} represent the estimation error, estimation time, and training time, respectively. We collect E_{train}, T_{est}, and T_{train} of all candidates, and each of them is normalized to [0, 1] (so $Cost(P, f) \in [0, 3]$)[3]. Based on the costs, we use the model with the lowest cost. This cost model evaluation is done for each dimension.

[3]It is also possible to use weights $\{w_1, w_2, w_3\}$ for the criteria.

Training of regression models. For a given regression model, we use the least square method to obtain its parameters. We introduce the training of polynomial models. (The training for spline linear and log linear models is essentially similar to the training for a linear regression model, which is trivial.)

Assume that we have a training dataset containing pairs of (p_i, r_i), where r_i is the rank of p_i in the dataset determined by the sorted order of a given dimension (x or y). For a polynomial model $f(z) = \theta_0 + \theta_1 z + ... + \theta_n z^n$, where z is the x- or y-dimensional value of a given point, we can write the residual sum of squares as:

$$RSS = \sum_{i=1}^{N'} (f(z_i) - r_i)^2,$$

where $N' \leq N$ is the number of input points. RSS can be written in a matrix format [56]:

$$RSS = (Z_v\theta - Q_\gamma)^T (Z_v\theta - Q_\gamma),$$

where

$$Z_v = \begin{bmatrix} 1 & z_1 & z_1^2 & \cdots & z_1^n \\ 1 & z_2 & z_2^2 & \cdots & z_2^n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & z_{N'} & z_{N'}^2 & \cdots & z_{N'}^n \end{bmatrix}, \theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_n \end{bmatrix}, Q_\gamma = \begin{bmatrix} r_1 \\ r_2 \\ \vdots \\ r_{N'} \end{bmatrix}.$$

Here Z_v is the Vandermonde matrix. The optimal $\theta = \{\theta_0, \theta_1, ..., \theta_n\}$ should minimize RSS. We have:

$$\frac{\partial RSS}{\partial \theta} = Z_v^T Z_v \theta - Z_v^T Q_\gamma = 0.$$

Hence, we can calculate θ by:

$$\theta = (Z_v^T Z_v)^{-1} Z_v^T Q_\gamma.$$

Regression models are arranged in a 2-tier manner (the first tier corresponds to the x-dimension with a single model f_x whereas the second tier corresponds to the y-dimension with m models f_y^1, ..., f_y^m). We train these regression models in a top-down style. After training f_x, we divide the dataset into m buckets and assign each point to the j-th bucket if its x-dimensional value satisfies

$$j = \lfloor \frac{m}{N} f_x(x_i) \rfloor.$$

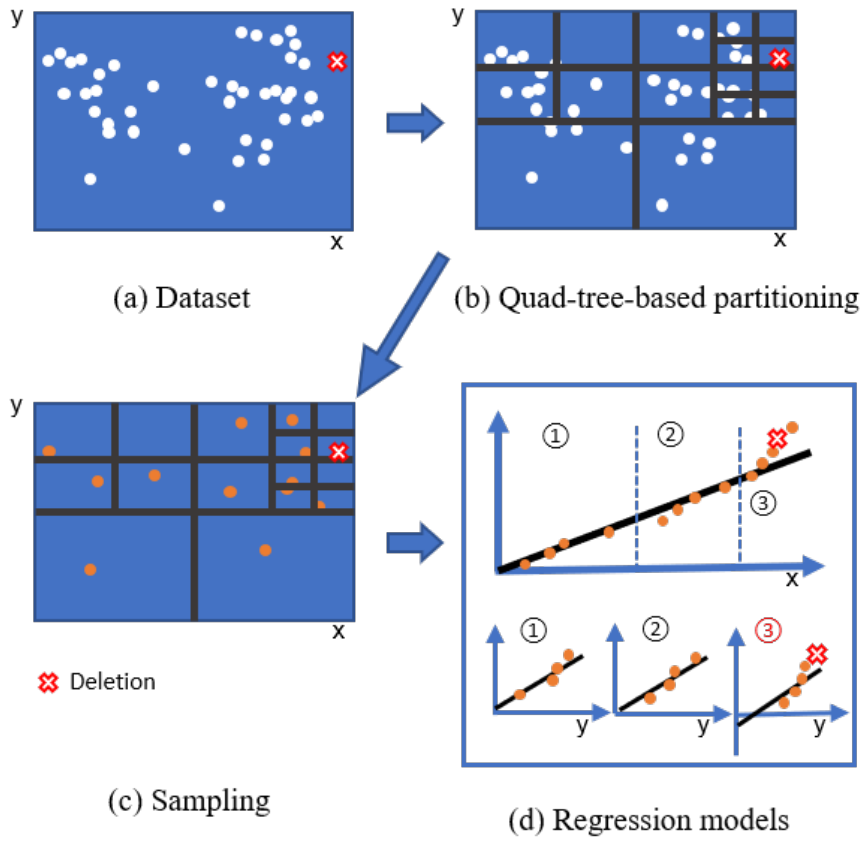


Figure 3.5: The workflow of model update. The red token represents a point deletion.

Then, we train the regression model for each bucket based on the y-dimensional values. The training flow is also described in Algorithm 1.

3.3.5 Model Update

We consider point insertions, deletions, and replacements as updates. Figure 3.5 illustrates the flow of our model update when a deletion is given.

Given an update, we first check whether we need to update the quad-tree-based partitions, which is shown in Figure 3.5(b). If an update makes the number of data points belonging to a node larger than the node capability of the quad-tree, the partition will split. If an update reduces the total number of data points in the four child nodes (partitions) belonging to the same parent node in a quad-tree (where each parent node has four child nodes) to a value that is no larger than the node's capacity, the four child

nodes (partitions) will merge into a single node (partition). When necessary, the samples also need to be updated. (If not, we update α in Equation (3.2), and the update is over.) We conclude the following three cases that we need to update the samples from the corresponding partitions.

- A partition is split (mainly due to an insertion).
- A partition is merged (mainly due to a deletion).
- The current sample is removed (due to a deletion or replacement).

In these cases, we create update records. Specifically, we create $\langle s, ins \rangle$ for an insertion and $\langle s, del \rangle$ for a deletion, where s is the original sample of the partition that needs to be updated. Also, each of new partitions picks a random sample. These new samples are assigned to the corresponding buckets. Notice that replacements are regarded as deletion-then-insertion.

As each sample s is associated with a unique bucket, we can identify the bucket to which s belongs by using a hash table, i.e., by using s as a key. From this approach and update records, we can identify the models that need to be updated, as each model is trained on the samples in the bucket. We update the corresponding regression models by using the training method (like in Section 5).

We note that, given an update, a quad-tree partition rarely changes, so we rarely update the models. Even if some partitions are updated, we need to update only a small number of models, as such updated partitions exist locally. This mechanism helps minimize the time for updating the SAFE structure, which is empirically confirmed in Section 3.4.3.

3.4 Experiments

This section reports our experimental results. We conducted experiments on a server with an Intel Xeon Gold 6254@3.10GHz CPU and 768GB RAM, running Ubuntu 18.04 LTS. The evaluated methods were single-threaded and implemented in C++.

3.4.1 Setup

Datasets. We used three real-world datasets and one synthetic dataset, and they contain millions of two-dimensional points. Table 3.3 shows the information on the datasets. The

Table 3.3: Datasets for evaluation

Name	Category	Size (million)
Colorado	Real	35
Paris	Real	28
Shikoku	Real	9
Gaussian	Synthetic	20

three real-world datasets were extracted from OpenStreetMap (OSM)[4]. They contain pairs of latitudes and longitudes. We also prepared a synthetic Gaussian dataset with a mean of 100 and a variance of 500 for each dimension.

Evaluated methods. Our experiments evaluated the following cardinality estimation methods.

- SAFE: Our proposed estimator. We set the cell capability of the quad-tree to 10.
- LHist [62]: is the state-of-the-art data-driven cardinality estimation method for spatial data. It uses polynomial models to approximate the data distribution, and we set the degree of polynomial models to 6 as recommended in the original paper.
- QuickSel [78]: QuickSel is a state-of-the-art supervised (query-driven) cardinality estimation method. It continuously learns from incoming queries to update its Gaussian mixture models. To make it comparable to the above methods, we prepared 1,000 queries as the training set for initializing QuickSel before evaluation.

3.4.2 Result: Static Data Case

For each dataset, we prepared three groups of queries. Each group contains 1,000 randomly generated range queries with a mix of selectivity of about 0.1 and 0.01. The three groups are regarded as disjoint clusters, i.e., queries in the same group (in different groups) have similar (totally different) query regions. In our preliminary experiment, which is shown in Figure 3.6, we found that R-tree runs range counting with a selectivity of 0.01 in less than 10 milliseconds in average. Then, to ensure the effectiveness of estimations, we controlled the estimation time of all the evaluated methods to around 100 microseconds in the following experiments by default. Therefore, m in SAFE was tuned accordingly. Note that this section evaluates the cardinality estimation models in the case where no updates occur.

[4] http://download.geofabrik.de/

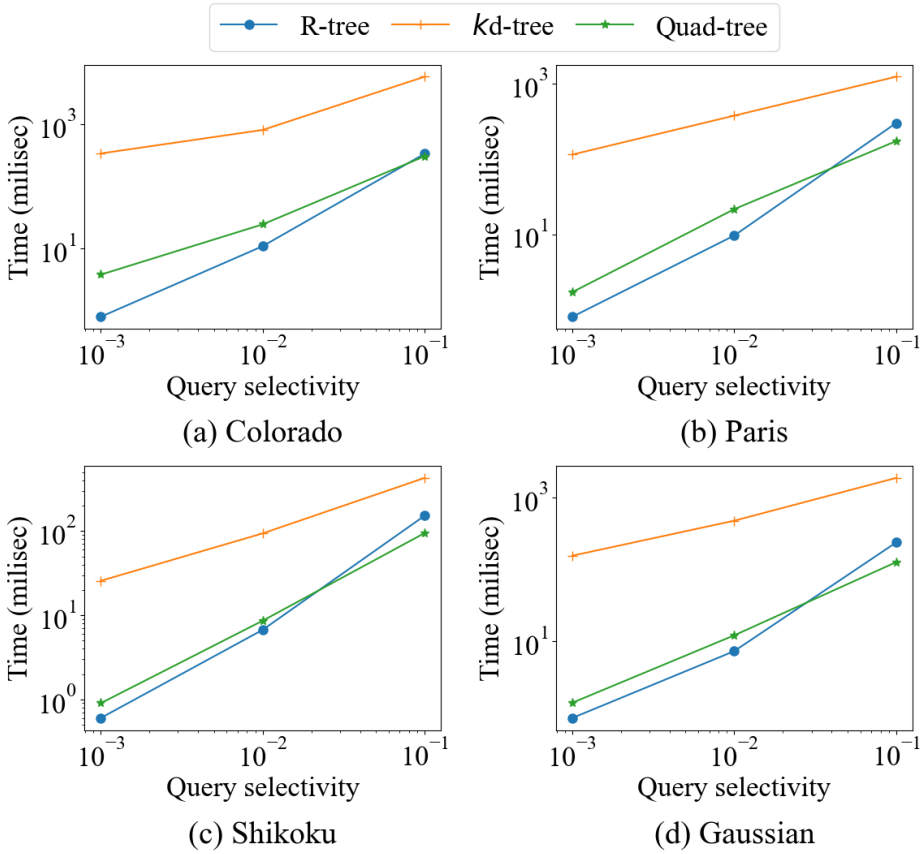


Figure 3.6: Running times for range counting conducted by R-tree, *k*d-tree, and Quad-tree.

Construction time. Figure 3.7 shows the construction time of each method. The construction time of SAFE is not the shortest, but the difference is not large. The construction of SAFE can be divided into three parts: data partitioning based on a quad-tree, sampling, and the training of regression models according to the design in Section 3.3. If we assume that the quad-tree is built in advance (for efficiently running range queries as in Figure 3.6), we can ignore the quad-tree construction time, the main bottleneck of building a SAFE. Then, under this assumption, its construction time can be the fastest.

Accuracy. Figure 3.8 shows the 50th and 95th q-errors of each method. SAFE and LHist show similar q-errors (but SAFE is usually better). It is also seen that QuickSel shows the lowest 95th q-error among the three methods. Actually, this result is expected because QuickSel exploits incoming queries for learning, which is quite advantageous against SAFE and LHist. Nevertheless, (i) all three methods show similar 50th q-errors

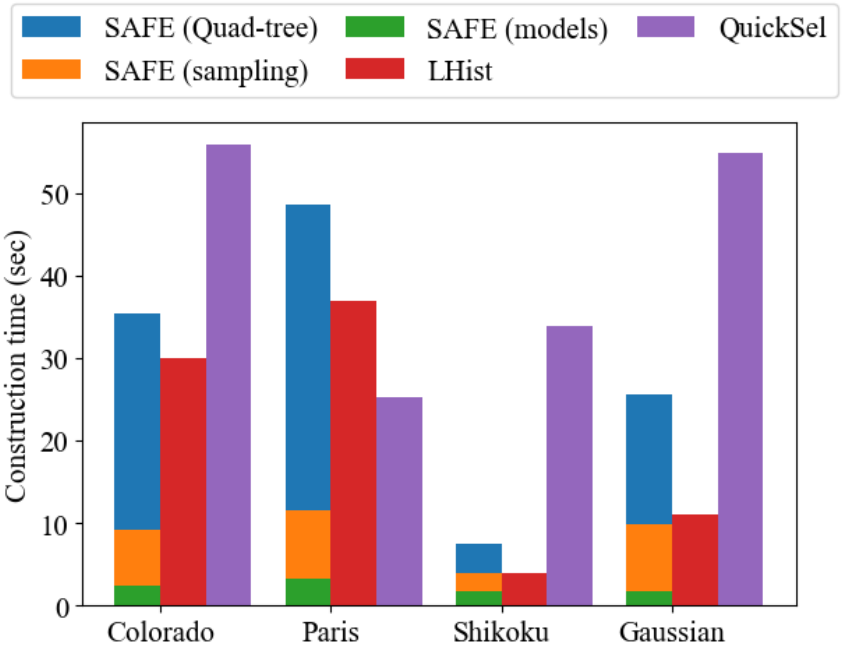


Figure 3.7: Construction time evaluation on different datasets

and (ii) they are close to 1 (nearly no error). We therefore see that SAFE can (often) yield an accurate cardinality for a given range query without any query workloads. (Later, we show the advantage of SAFE against QuickSel w.r.t. time and accuracy trade-off and updatability.)

Accuracy vs. estimation time. We next investigate the trade-off relationship between accuracy and estimation time, and the result is depicted in Figure 3.9. We see that QuickSel has the worst trade-off, whereas SAFE does the best. If applications require faster estimation time, the accuracy of QuickSel degrades quickly. On the other hand, SAFE does not lose its accuracy so much even when a faster estimation time (e.g., less than 10 microseconds) is required. This is a great advantage of SAFE. SAFE can choose the most suitable types of regression models for different settings. This feature makes a good trade-off between accuracy and estimation time possible.

Sampling strategies. We first compare the performances of the three sampling strategies (random point, averaged point, and center point) introduced in Section 3.3.3. We trained SAFE with different sampling strategies. Tables 3.4–3.7 show that these strategies have nearly the same sampling time. However, as for estimation accuracy, the center point

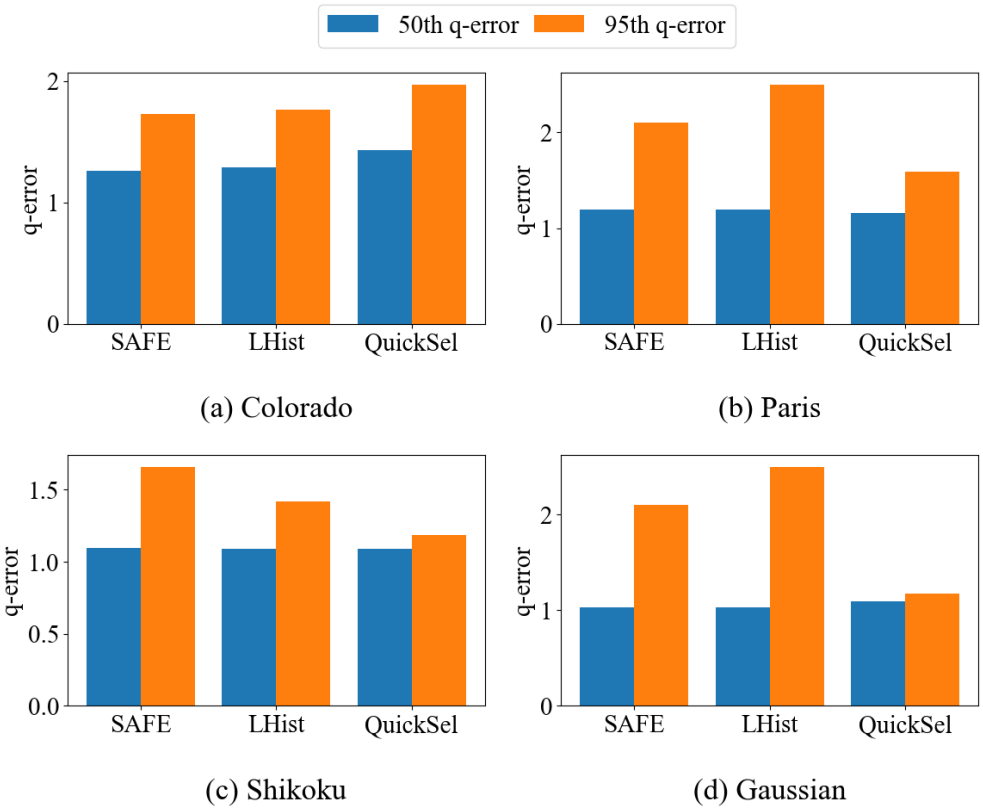


Figure 3.8: Accuracy evaluation on different datasets

strategy fails, whereas the others share similar accuracy. We therefore used the simple random point strategy for SAFE.

Table 3.4: Comparison of sampling strategies on Colorado

Strategy	Time (sec)	50th q-error	95th q-error
Random point	11.1	1.26	1.77
Averaged point	11.2	1.25	1.74
Center point	9.5	1.22	1.80

Sampling efficiency for training. Table 3.8 compares the training time of our models using samples and all points. Clearly, using samples obtains about x10 times faster training time, which confirms our motivation of using only samples.

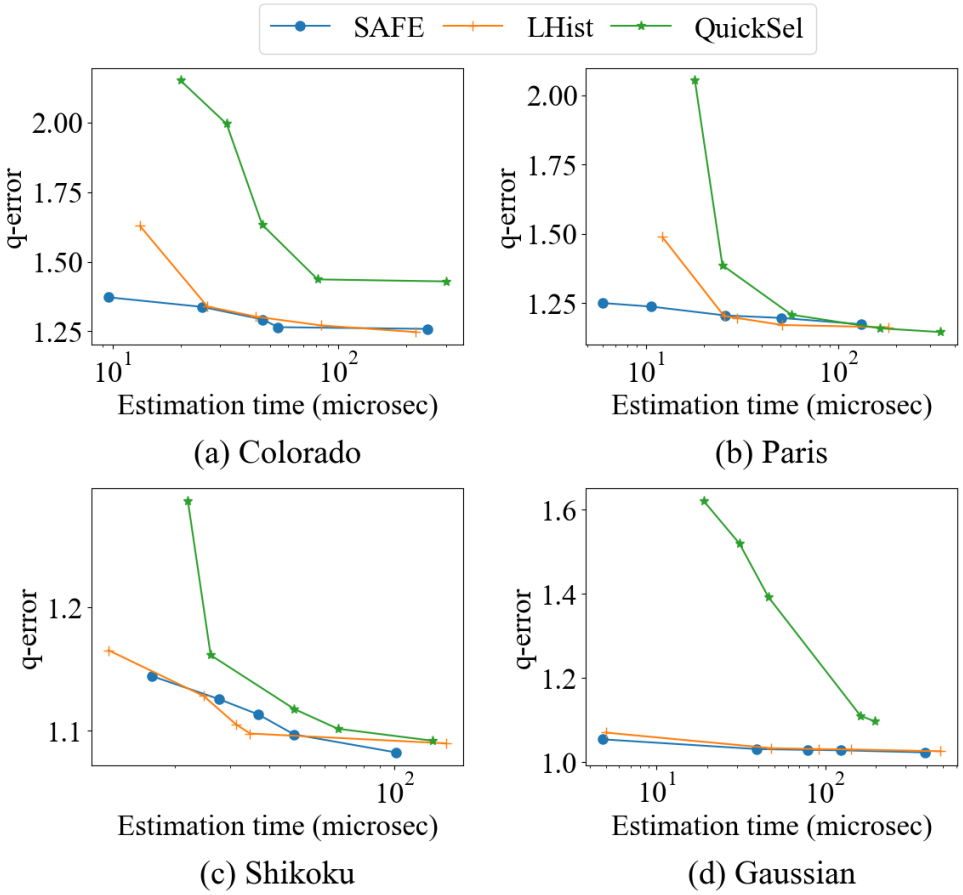


Figure 3.9: Comparison evaluation on estimation time versus accuracy (50th q-error)

Table 3.5: Comparison of sampling strategies on Paris

Strategy	Time (sec)	50th q-error	95th q-error
Random point	7.0	1.12	2.10
Averaged point	6.5	1.19	1.98
Center point	5.8	1.21	1.99

Table 3.6: Comparison of sampling strategies on Shikoku

Strategy	Time (sec)	50th q-error	95th q-error
Random point	1.8	1.09	1.67
Averaged point	2.1	1.09	1.61
Center point	1.4	1.15	2.57

Table 3.7: Comparison of sampling strategies on Gaussian

Strategy	Time (sec)	50th q-error	95th q-error
Random point	4.7	1.044	1.060
Averaged point	4.6	1.050	1.068
Center point	5.0	1.574	22.822

Table 3.8: Training times on all points and samples

Dataset	All points	Samples
Colorado	25.7	2.9
Paris	21.1	2.3
Shikoku	4.9	0.9
Gaussian	20.0	1.7

3.4.3 Result: Dynamic Data Case

We turn our attention to the dynamic data case. This section evaluates accuracy and update time. We here note that estimation time does not change even when we update the models. As competitors, we used QuickSel and SAFE without update[5]. This strategy, denoted by Scaling, scales the estimation results of the *non-updated* model by using the original and updated dataset sizes.

We sequentially updated the datasets by using one million insertions, one million deletions, and one million replacements. The insertions followed a Gaussian distribution within the corresponding data space, and the deletions were randomly selected from the original datasets. Each replacement replaced a randomly selected original point with a synthetic point following Gaussian distribution (i.e., the data distribution gradually changed). We used 300 queries with selectivity of 0.1 and 0.01 for the evaluation.

Figure 3.10 shows the continuous update results on different datasets. SAFE keeps high accuracy as its 50th q-error keeps being almost 1.0 and the 95th q-error is also generally low. On the other hand, Scaling and QuickSel cannot keep accurate estimations, and, as they receive more updates, their 50th q-errors and 95th errors always become larger. Comparing results on different datasets, we can observe the following points: (i) For replacements, the accuracy loss of Scaling and QuickSel is huge. It is attributed

[5]Recall Section 3.2.3, and, to update QuickSel's model for the up-to-date dataset, we need to compute the results of all existing queries, which requires more than a few seconds even for hundreds of queries. This is too slow for dynamic data, thereby we followed the original strategy of QuickSel (i.e., its model was not updated). In addition, LHist does not support data updates and needs reconstruction (incurring at least more than a few seconds for each update), thus we did not use it as a competitor.

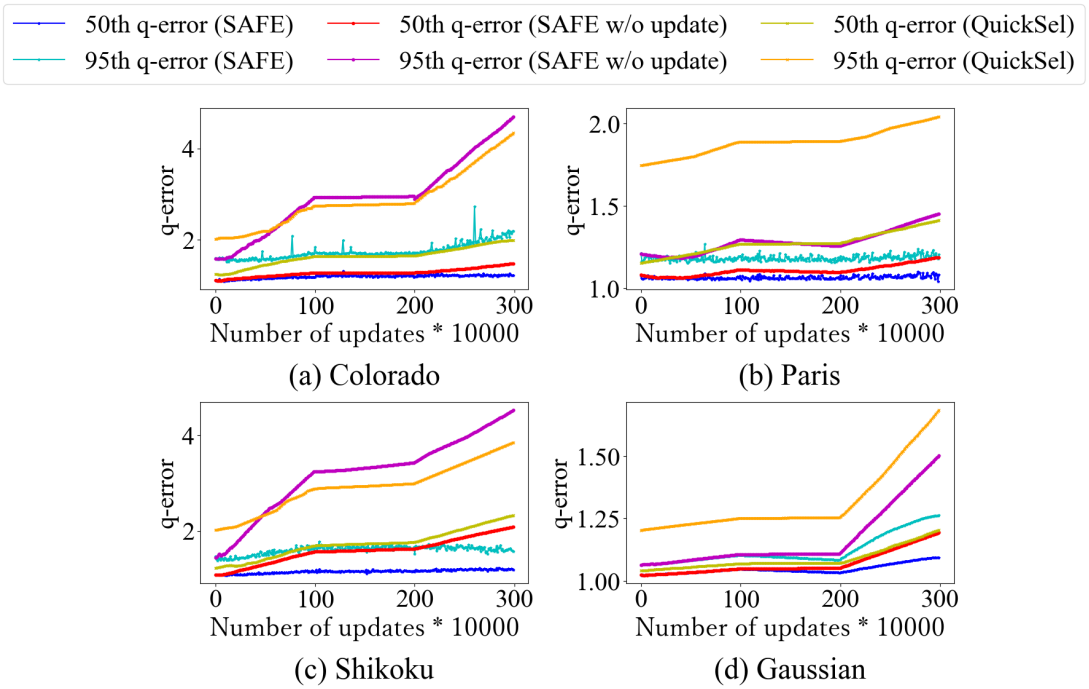


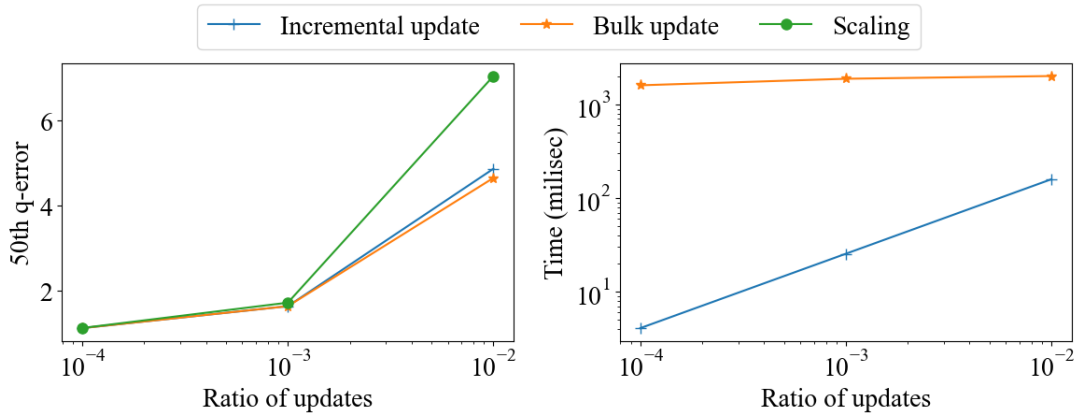
Figure 3.10: Continuous update evaluation

Table 3.9: Average update time of SAFE per an update (microsec)

Datasets	Colorado	Paris	Shikoku	Gaussian
Update time	9.9	19.8	6.2	14.1

to the large variations of data distribution incurred by the replacements. Our update strategy does not suffer from this. (ii) The q-error curves of SAFE have sawtooth-like fluctuations. Some data updates may incur more skewed data distributions, which lead to low-quality approximation. Nevertheless, its average q-error remains low. The credit for SAFE's ability to maintain good accuracy during data updates should be attributed to the incremental strategy proposed in 3.3.5. SAFE can update the necessary regression models when encountering data updates, making it possible to keep the models up-to-date with every data update.

We next investigated the update time of SAFE per an update, which is shown in Table 3.9. We see that the update time is around 10 microseconds and quite fast. It is attributed to the light-weight structure design and sampling-ass This is a desirable result for frequent data updates.



(a) Accuracy (b) Update time

Figure 3.11: Comparison between incremental update and bulk update on a Gaussian dataset.

Is bulk update useful? When data update comes in a batch (like tens of thousands of insertions into a dataset), one may consider that a bulk update is better than dealing with them one-by-one by our update strategy. If a batch contains many updates, it is often the case that insertions/deletions/replacement points are usually distributed all over the data space. Then, nearly every model can benefit from model updates, thereby the sample-based update judgment can be inefficient. We hence consider a bulk update strategy for this case.

Compared with our original update strategy, we always update the model for the x-dimension. This update is needed for large variations of the data distribution. Compared with the construction procedure, the bulk update saves much time in building a quad-tree and generating samples.

Figure 3.11 shows our experimental result on a Gaussian dataset. We generated point insertions from the Gaussian distribution with a variance of 1 for each dimension (notice that this variance is different from the original). We evaluated the accuracy and total time for model updates. We observe that incremental and bulk updates show similar accuracy. As for the update time, bulk update took about two seconds to deal with all updates and is robust to the number of updates. Notice that this update time is much faster than the SAFE construction time. However, our incremental update is much faster than the bulk update strategy even when we have 200,000 updates (i.e., 10^{-2} case) at a time.

Given the experimental results in Sections 3.4.2 and 3.4.3, it is clear that SAFE is better than the state-of-the-art cardinality estimation methods since SAFE is better at estimation-accuracy trade-off and updatability than them.

3.5 Related Work

Data-driven cardinality estimation. Data-driven cardinality estimators learn the data distribution from a given dataset directly. Recall that SAFE also belongs to this category. A representative example is traditional histograms. Latest learning-based methods tend to consider this task as an unsupervised problem. The Sum-Product network [38] recursively assigns points into clusters. It uses a probabilistic graphic model to approximate the joint data distribution. A SUM operator is then used to add estimated results from all the clusters. Those data-driven estimators can well handle two-dimensional range queries. However, they cannot update easily when meeting data updates. Therefore, they are not suitable for dynamic data.

Query-driven cardinality estimation. Query-driven estimators have also been developed. Some traditional histograms are refined based on given query workloads [1, 70]. Many state-of-the-art cardinality methods tend to employ this query-driven style [33], because deep learning models well suit this style. QuickSel [78] is the most representative among non-deep learning technologies. However, its supervised nature, which requires queries to learn the queried data distribution, limits applications. Those query-driven estimators can easily handle range queries as they are the most simple query type. In this chapter, we focus on the dynamic case, which means the training queries will become out-of-date after data updates. It is the main barrier to applying query-driven solutions to range queries on dynamic spatial data.

Spatial data partition. Normally, a given spatial dataset is partitioned and indexed by some classical tree structure to enable fast query processing. Recent works on spatial data partitions provide learning approaches for more efficient data processing and analytics.

The incremental partition for spatial data [96] proposes a framework that makes block-level systems better maintain spatial partitions. It integrates R*-tree and LSM-Tree into its framework and proves its superiority over other partitioning methods. A partition agent is proposed to choose an approximate partitioning technique based on the data distribution [95]. It adopts deep learning models to evaluate the quality of different

partition technologies considering different data distributions. It can automatically decide which partition technique to be used for shifted data distributions. Qd-tree [108] can improve the query performance by optimizing the metric of the number of blocks accessed by a query. It introduces a reinforcement learning algorithm to make partitioned layouts better suit the data distribution.

As we require fast data partitioning techniques to deal with data updates, learning-based partitioning, which typically requires training queries, is not appropriate. Therefore, the above learning-based partitioning techniques are not available for SAFE.

Learned spatial indexes. Learned indexes provide a new way to construct spatial indexes [103]. The range query is the most fundamental function to be supported by learned spatial indexes. Literature [80] extended RMI to 2-dimensional spaces by using z-ordering. LISA [59] is the state-of-the-art learned spatial index. Instead of z-ordering, it proposes a novel shard-based solution to map spatial keys into one-dimensional mapped values. To process range queries, they will map the border values of queries to one-dimensional values in the same way as mapping spatial keys. Waffle [73] is introduced to handle highly dynamic datasets. It can automatically optimize the index by managing the trade-off between the query and update speed. Though Waffle can well handle range queries in a dynamic setting, it requires a query workload in advance and its idea cannot be used in our setting.

3.6 Conclusion

Cardinality estimation plays an important role in optimizing spatial query plans. Considering the real-world spatial applications, spatial datasets are subjective to data updates, i.e., dynamic. In this chapter, we focus on the dynamic spatial data case and propose a new learned cardinality estimator SAFE. However, it exists several challenges to achieve this: i) The performance of the estimator should remain stable on a dataset with frequent data updates. ii) The estimation should be fast, which is hard to achieve by using common neural networks. iii) The estimator needs to support model updates. To solve these challenges, SAFE employs (i) sampling-based learning for fast training, (ii) 2-tier regression models to enable fast and accurate estimation while having an efficient update ability, and (iii) an incremental model update strategy. Combining these three mechanisms, SAFE achieves low-latency and accurate estimation while being

robust to data updates. Our experimental results demonstrate that SAFE outperforms the state-of-the-art estimators while supporting efficient model updates.

From the experiment results in Section 3.4, we can observe that the main advantage of SAFE compared with other competitors is the support for frequent updates. It is ascribed to the proposed incremental update strategy. To further achieve better estimation performance and strengthen our advantages, we consider the following two points. First, we want to extend the solution for more general spatial data cases, like three-dimensional data. We live in a three-dimensional real world and coordinates data often come with altitude information. Also, many spatial objects are represented by combinations of data points. To make our solution capable of these different data types, we need to improve the structure to well approximate the complex data distribution, which is a challenge for our current design. Second, we seek update solutions for existing learned cardinality estimators mainly using neural networks. Those existing methods show advantages in scenarios like high-dimensional data and join queries, which also exist in spatial data applications. If we can make these learned estimators update-friendly, we will benefit from them to develop our estimation solutions for spatial data.

Chapter 4

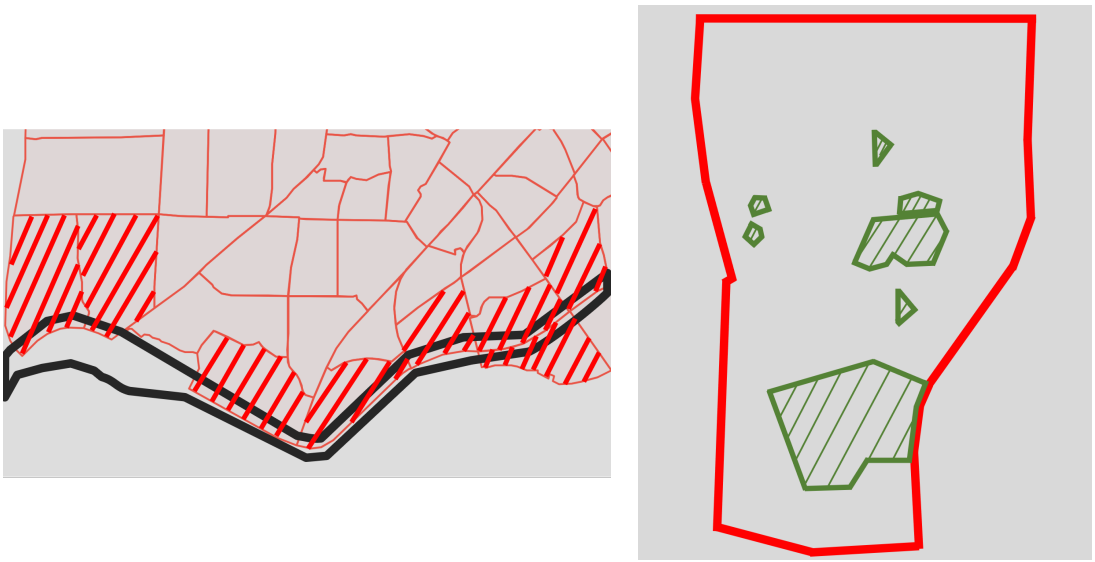
PolyCard: A Learned Cardinality Estimator for Intersection Queries on Spatial Polygons

4.1 Introduction

Spatial applications require many spatial objects like polygons and efficient techniques that process such spatial objects [11, 83]. Many spatial objects in real-world applications are represented as polygons. For example, in map applications, districts, rivers, and parks are represented as spatial polygons (or a set of polygons), and this chapter focuses on spatial polygons. One of the fundamental operations employed in these applications is *intersection query*, which, given a query and a set of spatial polygons, finds all spatial polygons intersecting with the query in the dataset. As shown in Figure 4.1, this query supports (i) finding all districts crossed by a high-risk river during heavy rainfall[1] and (ii) looking for parks located within a specific region using a map application [42, 67]. These examples trivially clarify the importance of intersection queries.

As introduced in Section 1.1, low response time for queries is an essential requirement for application users. Database management systems (DBMS) usually use a cost-based query optimizer to generate the most efficient query plan and to achieve low query latency. The query optimization heavily relies on good estimates of the cardinality of query results [44, 85, 86]. Therefore, to process intersection queries on

[1]These districts are from https://geoshape.ex.nii.ac.jp/.



(a) (b)

Figure 4.1: Examples of intersection queries. (a) Find districts in a city (red polygons) crossed by the bottom river (the black polygon). The results are denoted by the red polygons with slashes. (b) List parks (green polygons with slashes) located in the specified district (the red polygon).

spatial polygons efficiently, we need an estimator that estimates the cardinality of a given intersection query with high accuracy.

Motivation and challenge. Existing cardinality estimation methods for polygons typically rely on histograms built on the minimum bounding rectangles (MBRs) of polygons [55]. They estimate the cardinality of a given query's result based on the statistics derived from the histograms. Unfortunately, these methods have two drawbacks: (1) They approximate polygons by their MBRs, which makes the estimation essentially inaccurate. (2) They cannot maintain stable performance for different datasets and queries. For example, the estimation time of histogram-based methods is highly dependent on the number of buckets accessed. Some methods designed to optimize spatial intersection searches/joins can also be used for cardinality estimation [11]. RI [27] provides efficient approximations for polygons, particularly for the spatial intersection join cases. However, RI is slow to estimate the cardinality of a single intersection query's result.

Learned cardinality estimation methods mentioned in Section 1.1 provide efficient estimations for relational database systems [22, 98, 99, 110]. They have shown great

advantages compared with traditional histogram-based methods. Most of them adopt deep neural networks and train these networks in a supervised manner. However, they cannot be used for spatial polygons. Each polygon is represented by a set of two-dimensional coordinates. The number of coordinates is variable, e.g., from three to hundreds in real-world spatial datasets [24]. The variable length input is not allowed for neural networks. The most common solution to this issue is padding all the input data with zeros [36]. Considering the possible sizes of polygons, the zero padding greatly increases the computation cost for both training and estimation.

Moreover, collecting high-quality training data is always challenging for learned cardinality estimators [51, 54] because the skewed distribution of polygons in real-world polygon datasets makes obtaining sufficient training data difficult. One may come up with randomly generating query (training) polygons to address this issue. This is, however, not appropriate because the shape and the number of vertices of polygons are not fixed. Furthermore, the cardinality distribution of randomly generated queries has a long tail. The distribution with a long tail makes convergence of the training of the network difficult [111].

As can be seen above, existing techniques are not appropriate for cardinality estimation of intersection queries on polygons, despite the importance of efficiently supporting this task. Our data-driven design in Section 3 is incapable of approximating the data distribution of polygons because the distribution is too complex to be approximated by simple regression models. Other data-driven designs using deep neural networks tend to have a large estimation time. Motivated by this fact, we devise an efficient and accurate query-driven cardinality estimator for intersection queries on polygons and overcome the following main challenges.

- Variable input size: Neural networks need to be compatible with input polygons of variable sizes.
- Training data generation: The high-quality and sufficient training data is desired for a learning method.
- Fast estimation: The estimation time should be at least competitive with light-weight methods, e.g., histogram-based ones, and be quite fast (e.g., microsec-order).

Contribution. In this chapter, we propose a learned cardinality estimation method for polygons, providing fast and accurate estimations for spatial intersection queries. We summarize our contributions below.

(i) PolyCard. We propose PolyCard, the first learned cardinality estimator for intersection queries on polygons. The main idea behind PolyCard is to make polygons of variable sizes compatible with the neural network and provide substantial estimation performance derived from learning techniques. We propose an adaptive sampling method to transform polygon data to a fixed size, which considers the spatial distribution of the coordinates belonging to a polygon. The differences between polygons are well preserved even after the transformation. The transformed polygons can be processed efficiently after normalization and sent into a specifically designed neural network. This technique yields the final estimation to be fast and accurate.

(ii) Training data generation. PolyCard is a supervised query-driven estimator, so its performance depends on training data (i.e., training queries). We propose a training data generator to provide training data distributed over the data space and covering different cardinalities. It is difficult to generate polygons serving as training queries directly because polygons have variable number of vertices. We perform data augmentation on real-world polygons and prepare sufficient polygons for the training. The distribution of polygons is naturally skewed. The long tail distribution of true cardinalities corresponding to randomly generated training queries heavily burdens the training process. We guarantee an even cardinality distribution of training data by a sampling strategy.

(iii) Experiments. We conduct experiments on four real-world datasets. Our results demonstrate that PolyCard outperforms state-of-the-art cardinality estimation methods and can maintain stable performances on different datasets. We also analyze the effectiveness of our proposed polygon transformation and training data generator by several comparison experiments. Figure 4.2 shows the overview of our results: PolygCard is very fast and has a better trade-off between speed and accuracy (q-error) than the competitors.

Organization. The rest of this chapter is organized as follows. We present the preliminary information in Section 4.2. We elaborate on PolyCard in Section 4.3. We report our experimental results in Section 4.4, and Section 4.5 concludes this chapter.

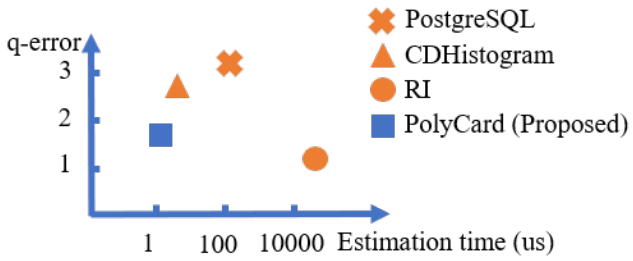


Figure 4.2: Accuracy (represented by the 50th q-error) vs. estimation time on Sports dataset

Table 4.1: Overview of symbols frequently used in this chapter

Symbol	Description
D	Polygon dataset
P	Polygon in a dataset D
p	Vertex in a polygon P
Q	Intersection query
$card(Q)$	True cardinality of Q
$\widehat{card}(Q)$	Estimated cardinality of Q

4.2 Preliminary

We define our problem in this section. Table 4.1 gives the main symbols we use in this chapter. Let D denote a set of polygons. A polygon is a typical spatial object type. We use P to denote a polygon, and it is defined as:

Polygon. P is represented by a list of vertices, i.e., $P = [p_1, p_2, ..., p_m]$. Each of its vertices p is represented by the two-dimensional coordinates, i.e., $p = (x, y)$. There is an edge (or a line segment) between consecutive vertices p_i and p_{i+1} (p_m and p_1 are also consecutive vertices). That is, a polygon is a two-dimensional shape closed by its edges. Notice that each polygon can have a different number of vertices.

This chapter considers intersection queries on polygons, and we define this type of query and its cardinality below. (Recall that Figure 4.1 illustrates some examples of intersection queries.)

Polygon intersection. Polygon A is intersected with polygon B if one of the following cases is met:

- Any pair of edges $\langle e_{ai}, e_{bj} \rangle$ intersects, where e_{ai} is an edge of A and e_{bj} is an edge of B.
- A (B) is fully contained in B (A).

Intersection query. Given a dataset D and a query polygon Q, the intersection query returns all polygons in D that intersect with Q.

Cardinality. The cardinality of an intersection query Q is the output size of Q, i.e., the number of polygons in D that intersect with Q. We use $card(Q)$ to denote the cardinality of Q.

The polygon intersection is first tested using the MBRs of two polygons. If MBRs do not overlap, the two polygons do not intersect. Otherwise, a line segment intersection test is used to check if any pairs of line segments (edges) are intersected [63, 81]. If no pairs of line segments intersect, a point-in-polygon test can determine if one polygon is fully contained within the other [31]. It takes $O(m \log m)$ time to test if two polygons are intersected and $O(Nm \log m)$ time for an intersection query, where m is the number of vertices in a polygon and N is the number of polygons in D. We address the cardinality *estimation* problem to help further accelerate query processing in this chapter. We use $\widehat{card(Q)}$ to denote an estimated cardinality of Q. We aim at fast and accurate estimation.

4.3 PolyCard

4.3.1 Main Idea

PolyCard is a learned cardinality estimator designed for intersection queries on polygons. We first solve the challenge of designing a learned model for polygons with variable-sized vertices. To map variable-sized vertices to fixed-sized ones, zero padding is the most direct solution and can make all polygons the same size as the maximal one. However, it increases the computation cost of NN, resulting in inefficiency. To transform polygons to a fixed size without having this inefficiency issue, we develop an adaptive sampling method. The idea behind the adaptive sampling is twofold. First, sampling can adjust the size freely. Second, adaptive sampling, which considers the spatial distribution of vertices, can well preserve the shapes of polygons even with smaller sizes of vertices.

Standard deep neural networks, such as convolutional neural networks (CNN), multi-layer perceptrons (MLPs), and autoregressive models, have been used for the cardinality

estimation problem on the other data formats [51, 75, 88, 110]. Inspired by these existing works, we build our neural network (NN) based on a combination of MLPs. Its structure helps achieve accurate estimation *within several microseconds* according to our evaluation, see Figure 4.2.

Recall our discussion in Section 4.1: we want to train a NN with query polygons of evenly distributed cardinalities. (Training queries with more evenly distributed cardinalities can help reduce high-percentile errors.) Generating polygons from scratch is challenging, complex, and laborious, due to the variable number of vertices [115] and complex and various polygon shapes. To overcome this challenge, we come up with the idea of using real-world datasets as sources of polygons. Specifically, we generate new (synthetic) query polygons based on real polygons from these datasets, with modifications such as shifting. Additionally, the cardinality distribution of the overall generated training data can be skewed and interfere with the convergence of model training, which makes the training a challenging task. We tackle this challenge by using a down-sampling strategy and achieve a uniform cardinality distribution for well model training. We develop a training data generator based on the above ideas.

4.3.2 Overview

Figure 4.3 shows the framework of PolyCard. For model training, we use our training data generator to obtain sufficient training queries and their true cardinalities. We featurize the training queries into dataset vectors and polygon vectors. Specifically, we represent the dataset information by one-hot vectors. Polygons are also represented as fixed-sized normalized vectors. A NN is trained with these vectors and the true cardinalities. After the training, we use the NN for cardinality estimation. Given a query, we featurize it into two vectors in the same way as the training. These two vectors are entered into the NN, and its output is $\widehat{card}$.

4.3.3 Training Data Generator

Our training data generator aims at two goals: (i) generating sufficient polygon queries and (ii) obtaining training data with evenly distributed cardinalities. Because polygons have variable sizes and vertices distribution, generating a polygon from scratch is not a good idea. We choose polygons from real-world datasets. Our generator chooses polygons from datasets randomly and performs transformations, i.e., shifting, reflection,

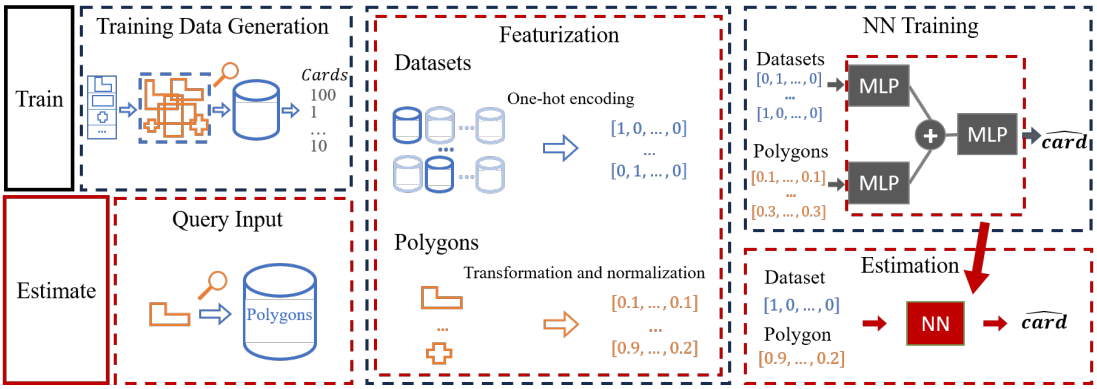


Figure 4.3: Framework of PolyCard. **Training**: Our training data generator is used to obtain sufficient training queries and their true cardinalities (left-top part). We featurize the training queries into dataset vectors and polygon vectors (middle part), that is, we represent the dataset information by one-hot vectors, whereas polygons are repressed as fixed-sized normalized vectors. The NN is trained with these vectors and the true cardinalities (right-top part). **Cardinality estimation**: Given a query input, we featurize it into two vectors in the same way as the training (middle part). We then use the trained NN to obtain $\widehat{card}$.

and perturbation, to generate polygons serving as training queries, as shown in Figure 4.4. To obtain a shifted version of a polygon P, we apply the same modifications to all coordinates of its vertices in each dimension. For the reflection, we randomly select a vertex of P and reflect all the other vertices vertically. To obtain a perturbation edition of a polygon P, we add a random small value to each coordinate of each vertex of P.

The distribution of polygons is naturally skewed [64]. This observation makes the cardinality distribution of randomly generated queries have a long tail, as shown in Figure 4.5(a). As discussed in Section 4.1, this distribution makes convergence of the training hard. Our generator helps obtain training data with evenly distributed cardinalities in two steps. First, generate new query polygons and compute their true cardinalities. Second, our generator guarantees the training data with an even cardinality distribution by down-sampling from the overall generated training data. To achieve this, we generate the cardinality histogram of training data. If the distribution shown in the histogram is skewed, the generator selects small portions of training data from buckets containing more data than the average. With our generator, we can always obtain training data with the cardinality distribution like the one shown in Figure 4.5(b).

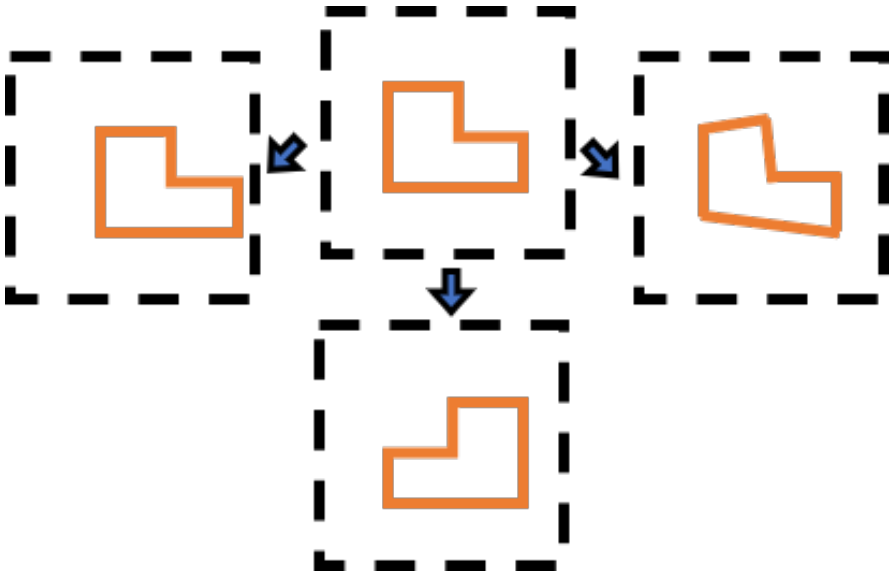


Figure 4.4: Generation of polygons. We perform shifting, reflection, and perturbation on polygons from real-world datasets.

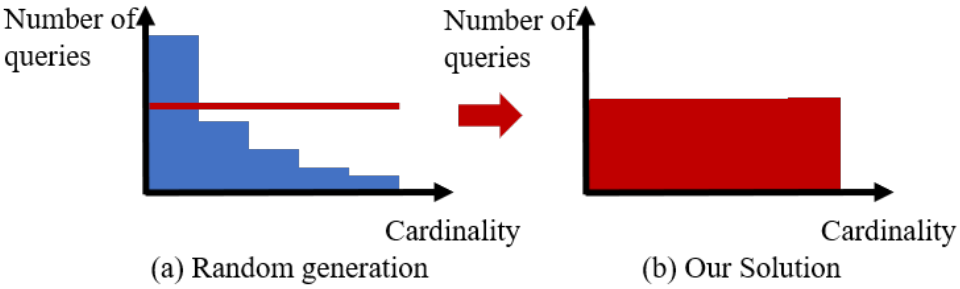


Figure 4.5: Histogram of cardinalities of training queries on Sports dataset.

4.3.4 Polygon Transformation

The polygon transformation aims at transforming polygons with variable-sized vertices to a fixed size m, which makes polygons compatible for feeding into our NN model. Algorithm 3 shows the details of the transformation. Let m be the number of vertices of a polygon P. We want to transform P to a similar polygon with n vertices.

If $n < m$, we run an adaptive sampling that considers the distribution of vertices. We first compute the MBR of the original polygon and divide the MBR into four equal-sized zones according to the midpoints of the four edges of the MBR (line 5). Then, we perform a uniform-like sampling by scanning the vertices sequentially. During the

Algorithm 3: POLYGON-TRANSFORMATION

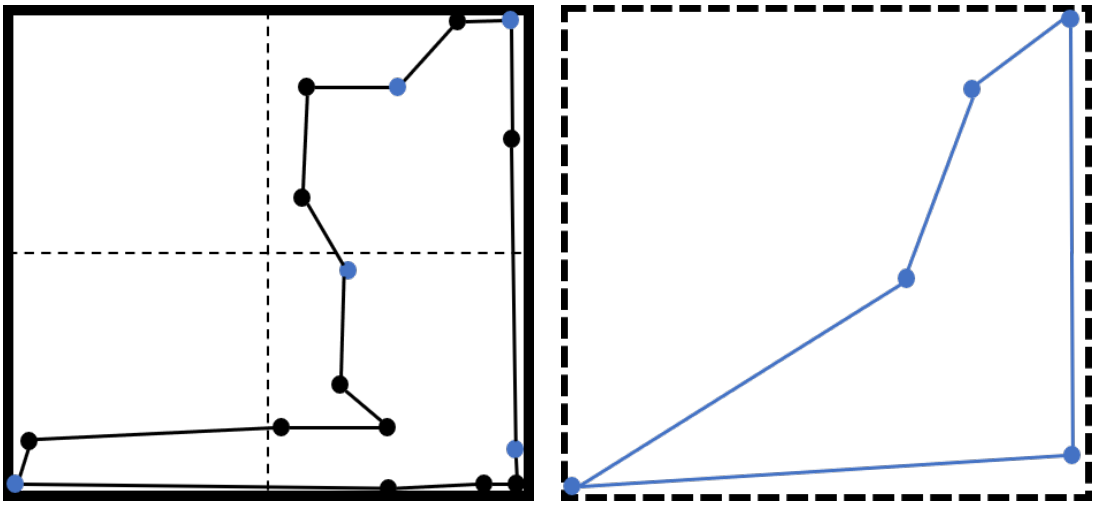
Input: P (a polygon represented by an array of m vertices) and n (the number of vertices of the transformed polygon)
Output: P' (the transformed polygon)

```
1  P' ← ∅, interval ← m/n, count ← 0
2  if n < m then
3      zone_flag ← False
4      foreach i ∈ [1,n] do
5          cur_zone ← GET-ZONE(P[i]) // P[i] is the i-th vertex in P
6          count ← count + 1
7          if zone_flag[cur_zone] = False then
8              P' ← P' ∪ {P[i]}
9              count ← count − interval
10             zone_flag[cur_zone] ← True
11         else
12             if count ≥ interval then
13                 P' ← P' ∪ {P[i]}
14                 count ← count − interval
15 else
16     foreach i ∈ [1,m] do
17         count ← count + 1
18         num_vertices ← FLOOR(count/interval)
19         if num_vertices = 1 then
20             P' ← P' ∪ {P[i]}
21             count ← count − interval
22         else
23             if num_vertices ≥ 2 then
24                 P' ← P' ∪ {INTERPOLATION(P[i−1], P[i], num_vertices)}
25                 count ← count − interval × num_vertices
```

scan, we guarantee that at least one vertex is sampled in each zone intersecting with the polygon. To achieve this, we locate the zone to which each vertex $P[i]$ belongs. If no sample belongs to the zone, we sample $P[i]$ (line 7).

If $m \geq n$, we adopt a uniform interpolation method for the transformation[2]. We sequentially scan the vertices and calculate the number of new vertices to be inserted

[2]Considering the estimation efficiency, n should be smaller than m for most polygons because a large n will increase the size of the NN and the computation cost for estimation.



(a) Original polygon (b) Transformed polygon

Figure 4.6: Our polygon transformation. (a) The polygon is represented by vertices and edges connecting the vertices. The outside rectangle is the MBR of the polygon. The black dashed line separates the MBR into four equal-sized zones referring to the four corners of the MBR. Given the fixed size of five vertices for transformation, we sample the blue vertices. (b) The transformed polygon.

between two vertices by interpolation (line 18). If the number of new vertices equals one, we simply treat the current vertex as a new vertex (lines 19–21). If the number of new vertices is larger than one, we interpolate the coordinates of two consecutive vertices and obtain the required number of new vertices with interpolated coordinates (lines 23–25). The transformation costs $O(\max(m,n))$ time.

In Figure 4.6(a), suppose that we transform the polygon to a fixed size of five vertices. Only two vertices fall into the left-bottom zone, and they are far away from the other vertices. If we run a standard random sampling, they are probably not sampled. Our adaptive sampling guarantees every zone where the vertices exit is covered. Thus, the transformation can preserve the shapes of polygons well. As shown in Figure 4.6(b), the shape of the transformed polygon is very similar to that of the original polygon.

4.3.5 Featurization

As shown in Figure 4.3, the input data fed into the NN consists of two kinds of vectors, respectively representing the dataset information and the polygon. We featurize an input

query into two vectors, a dataset vector and a polygon vector, to better represent the input and help the training converge.

The dataset vector represents which dataset is being queried. We use a one-hot vector of the length M, where M is the total number of datasets used as sources. For this vector, when the i-th dataset is accessed by a query, its corresponding dimension is one and the other dimensions are zero.

To featurize the query polygon, we first perform the transformation discussed in Section 4.3.4. Then we normalize the coordinates of vertices to avoid numerical instability. We construct the polygon vector by concatenating all coordinates of the polygon.

4.3.6 Model

For our NN, we employ MLPs, because they can approximate functions well and yield fast inferences [16, 72]. Each MLP is composed of two fully connected layers, each accompanied by a ReLU activation function:

$$MLP(x) = \max(0, \max(0, xW_1 + b_1)W_2 + b_2), \tag{4.1}$$

where x represents the input vector whereas W_1, W_2, b_1, and b_2 are learned parameters. The ReLU function is represented by $\max(0,x)$. We use one MLP for polygon vectors and another for dataset vectors. The outputs of the two MLPs are concatenated and then fed into the output MLP.

For the output MLP, we replace the final ReLU function with a sigmoid function, which is a common choice as the activation function for the output layer [32]. Compared with the other NNs, the main advantage of our design is the computation speed. For example, the fully connected layer consumes much less computation time compared with the convolutional layer (widely used in various NNs).

Since PolyCard is a supervised estimator, we pre-compute the true cardinality of each training query polygon. Notice that the cardinality has a wide range of possible values. For training, we perform the logarithmization and normalization on the cardinality to compress the range and reduce the impact of extreme values on model training [112]. To get $\widehat{card}$ for a query, we perform the reversion of the logarithmization and the normalization on the output of the NN.

4.3.7 Training and Estimation

Figure 4.3 shows the training and estimation flow of PolyCard. We take the following steps for the training: (i) generate training data by the generator, which contains the training queries and corresponding true cardinalities and (ii) train the NN with the training data. We use the q-error as our loss function, which is widely used for the cardinality estimation problem [88, 104].

To estimate the cardinality of a given query, we featurize this query into a dataset vector and a polygon vector. After feeding the featurized vectors into the NN, we obtain $\widehat{card}$ as discussed in Section 4.3.6. The estimation time is dependent only on the NN structure. Therefore, the estimation of PolyCard is stably fast for datasets of different sizes and queries of different selectivities.

4.4 Experiments

We report our experiment results and investigate the following questions in this section:

- Is PolyCard accurate, fast, and stable?
- How good is PolyCard compared with competitors?
- Are the challenges mentioned in Section 4.1 severe problems for the cardinality estimation? How much does PolyCard benefit from solving those challenges?
- How much does PolyCard benefit from solving those challenges?

4.4.1 Setup

Datasets. We used four real-world datasets widely used in spatial data processing works[3] [24, 27]. They contain polygons representing the boundaries of various types of objects. Table 4.2 shows the information of the datasets.

Workloads. We prepared one thousand randomly generated queries based on real-world polygons. We guarantee that they have a wide cardinality distribution from one to tens of thousands.

Evaluated methods. We evaluated the following methods in this chapter.

[3] https://spatialhadoop.cs.umn.edu/datasets.html

Table 4.2: Statistics of datasets

Name	Cemetery	Sports	Parks	Buildings
Size (million)	0.2	1.8	10	115
Average #vertices per polygon	10	11	36	7
Maximum #vertices in any polygon	572	5890	127954	1660

- PolyCard: is our proposed method. The size of transformed polygons is set to ten by default. (The impact of different sizes is discussed in Section 4.4.3.) PolyCard is trained on a GPU.
- PostgreSQL[4]: refers to the cardinality estimation method used in PostgreSQL 16. It estimates the cardinality based on the statistics of the datasets including histograms. PostgreSQL does not provide the exact estimation time, so we used the planning time to represent its estimation time instead.
- CDHistogram [49]: is one of the most representative histogram-based methods designed for spatial objects. It originally supports only rectangles. We extended it to support polygons by using MBRs. The bucket size of CDHistogram on each dimension was set to 10,000 by default.
- RI [27]: is a state-of-the-art approximation method for intersection joins on polygons. We employed its upper-bounding technique to estimate the cardinality of a given query's result.

Environment. All experiments were conducted on a server with an Intel Xeon Gold 6254 CPU, a NVIDIA Quadro RTX 8000 GPU, and 768GB RAM. The OS was Ubuntu 22.04.4 LTS. PolyCard was implemented in Python with PyTorch. We implemented CDHistogram in C++. For RI, we used the C++ source code provided by the authors.

4.4.2 Overall Performance

Estimation time. Figure 4.7 shows the estimation time of each method. PolyCard is the fastest method on all datasets except the Cemetery case. Furthermore, PolyCard provides a stable estimation time, i.e., its estimation time is not dependent on dataset size, which is desirable for large datasets. On the other hand, CDHistogram becomes slower as the dataset size grows, and PostgreSQL is usually slower than CDHistogram.

[4] https://www.postgresql.org/docs/current/row-estimation-examples.html

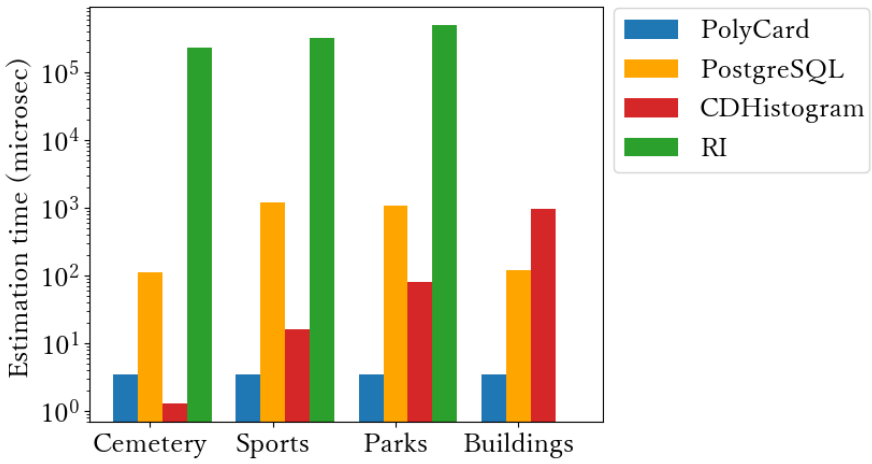


Figure 4.7: Estimation time.

Table 4.3: Accuracy evaluation results. Bold shows the winner.

Datasets	Methods	25th q-error	50th q-error	75th q-error	95th q-error	99th q-error
Cemetery	PolyCard	1.2	**1.45**	**1.91**	**3.22**	**5.3**
	PostgreSQL	**1.0**	1.85	2.33	5.0	7.0
	CDHistogram	1.6	2.25	3.26	6.0	8.0
Sports	PolyCard	**1.32**	**1.85**	**3.26**	**9.34**	**26.78**
	PostgreSQL	1.71	3.15	7.0	23.0	47.08
	CDHistogram	1.54	2.69	5.0	15.5	38.0
Parks	PolyCard	**1.29**	**1.79**	**3.03**	**8.86**	**19.73**
	PostgreSQL	1.46	2.35	4.5	18.69	90.26
	CDHistogram	2.47	3.43	5.31	13.0	30.02
Buildings	PolyCard	**1.72**	**2.99**	7.19	**37.47**	**126.36**
	PostgreSQL	7.74	28.85	75.68	333.05	1192.32
	CDHistogram	2.11	3.48	**6.48**	52.84	367.18

The estimation time of RI is much longer than the other methods and is longer than even the execution time of queries (normally several milliseconds)[5]. It is trivial that RI is not suitable for the estimation task because we can get the true cardinality in a shorter time than RI. We therefore did not evaluate RI in the remaining experiments.

Accuracy. Table 4.3 shows the accuracy evaluation results. Overall, PolyCard achieves smaller errors than PostgreSQL and CDHistogram. When considering high percentile errors (i.e., 95th and 99th q-errors), which are of great concern for the cardinality estimation task, PolyCard beats the competitors on all datasets. PostgreSQL has the

[5]The construction of RI on Buildings took more than several hours, so we omit its result.

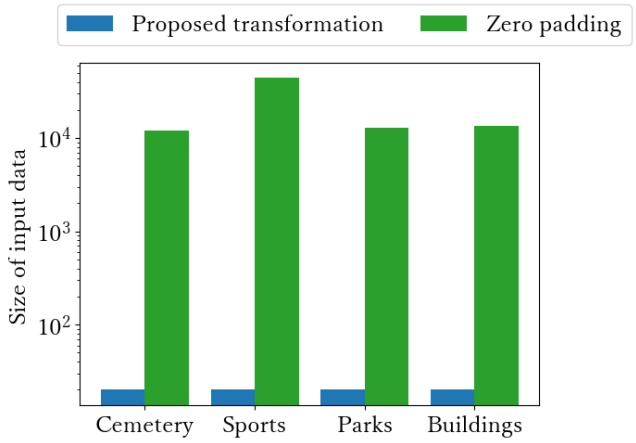


Figure 4.8: Sizes of the input data for the NN achieved by the proposed polygon transformation and zero padding

largest high percentile errors on all datasets except the Cemetery case. These large errors lead to a high risk of generating inefficient query plans.

4.4.3 Detailed Analysis

How good is our polygon transformation? We compare the sizes of input data obtained by the proposed polygon transformation solution (Section 4.3.4) with zero padding to confirm that zero padding is not a feasible approach. (Recall that zero padding is to fix the size of all polygons to the maximum one by adding zero.) Figure 4.8 observes that zero padding incurs too large data (more than 150GB) for NN training (i.e., tens of thousands of dimensions for each polygon). It will greatly enlarge the computation cost and make the training NN much more difficult. Therefore, we can observe that the existing solution to transform polygons to a fixed size is not suitable for our task and our adaptive transformation is a practical solution.

Furthermore, we compare our adaptive sampling transformation with uniform sampling and MBR approximation. Trivially, if method A provides a transformed polygon with a better similarity than method B, method A yields less error. Figure 4.9 shows the advantage of adaptive sampling. This result demonstrates that our polygon transformation is of high quality and helps achieve better accuracy.

Impact of reduced polygon size. Figure 4.10 shows the accuracy results corresponding to different sizes of transformed polygons. The accuracy can be greatly improved

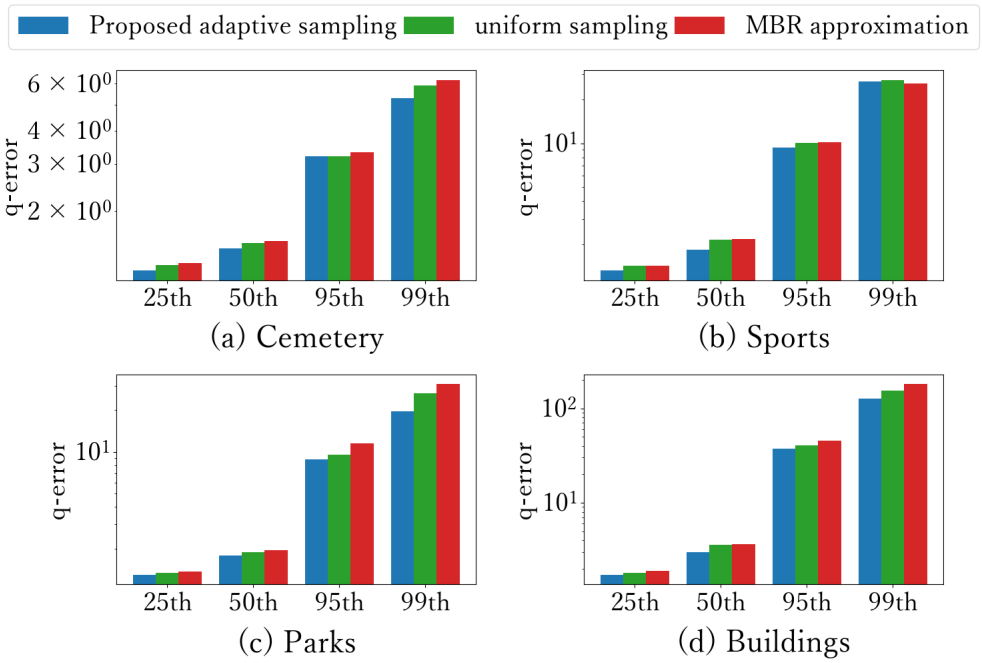


Figure 4.9: Accuracy comparison of the proposed adaptive sampling, uniform sampling, and MBR approximation for polygon transformation

(specifically high percentile errors) when the size increases from 4 to 10. Considering the average number of vertices per polygon shown in 4.2, large sizes like 25 cannot further improve the performance. On the other side, large sizes increase the storage and computation costs. A grid-based search can help find the most suitable size for each dataset. Here, the choice of size = 10 is sufficient for our evaluation.

How good is our training data generator? Figure 4.11 shows two convergence curves of PolyCard; one is generated by our proposed generator and the other is the random one discussed in Section 4.3.3. We can observe that PolyCard meets difficulty in convergence (at a small loss) when trained on randomly generated datasets. As discussed in Section 4.3.3, the cardinality distribution of randomly generated queries has a long tail. It makes convergence of the training hard. In the case of Cemetery, we found that more than sixty percent of queries in the randomly generated training queries have zero cardinality. As a result, the training loss shown in Figure 4.11(a) fails to decrease and becomes a stubborn constant. Furthermore, as shown in Figure 4.12(a), the trained model achieves poor estimation accuracy. Our training data generator solves this problem by generating training data with more evenly distributed cardinalities.

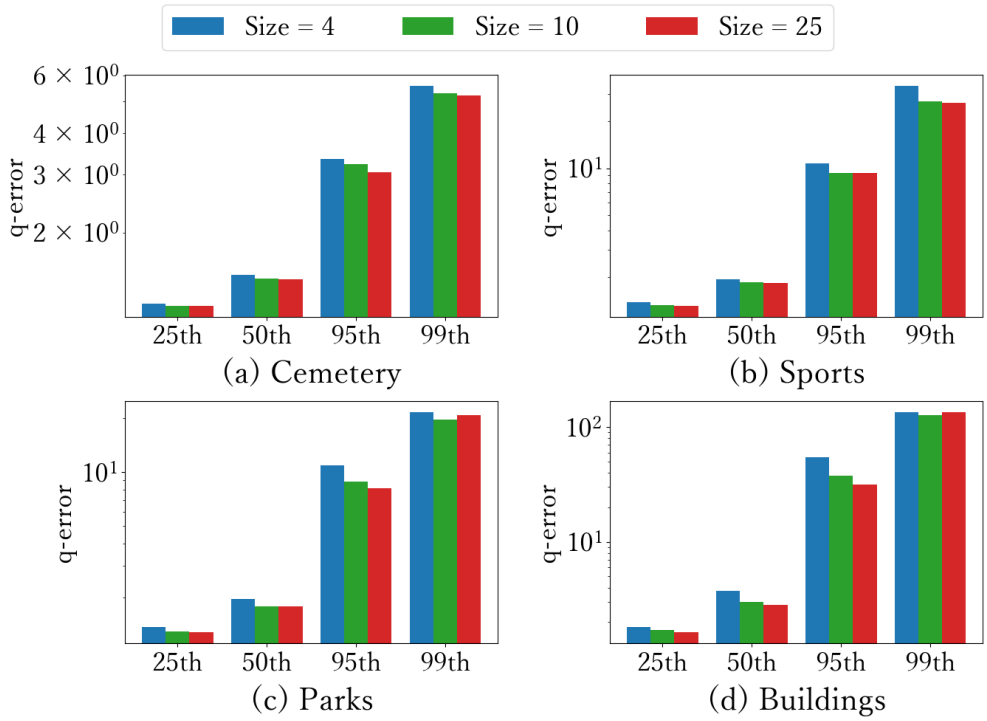


Figure 4.10: Impact of sizes of transformed polygons on the accuracy

Figure 4.12 shows the estimation errors of PolyCard after the above training. The q-errors of PolyCard trained on randomly generated queries are much larger. The above results clearly show that our training data generator helps the training converge and achieve reasonable accuracy.

4.5 Related Work

Histogram-based cardinality estimation. Histogram-based cardinality estimation methods are the most widely used in relational database systems like PostgreSQL [55, 102]. These database systems build histograms on relational tables as the statistics. Estimators for rectangle queries have also been developed for spatial cases [8, 20].

CDHistogram addresses this problem by maintaining four sub-histograms corresponding to the four vertices of a rectangle [49]. A bucket in a sub-histogram stores the number of corresponding vertices falling in the bucket. This design can avoid duplicate counts of rectangles spanning several buckets. Histogram-based methods can be used for our task by approximating polygons with their MBRs. Our experimental results

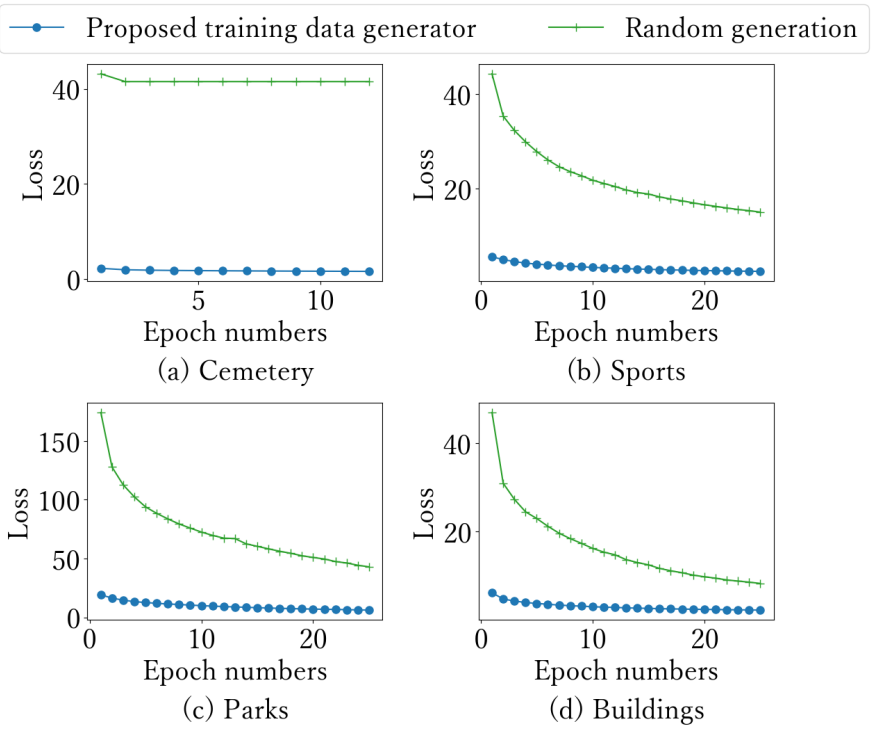


Figure 4.11: Convergence of training

confirm that this approximation hurts estimation accuracy. In addition, the accuracy of histogram-based methods is severely dependent on the bucketing resolution and the data distribution.

Spatial joins and related query optimization have been widely studied [8, 11, 91, 92, 94, 106]. They mainly focus on how to execute complex spatial queries and select the most efficient join algorithms. Note that they have not answered how to estimate the cardinality of an intersection query.

RI [27], which focuses on efficient approximation of polygons for spatial joins, can be used for cardinality estimation. However, our evaluation in Figure 4.2 finds it too slow to be used as a cardinality estimator. RI is designed for precise approximation of polygons and can provide an estimation near the true cardinality. However, it incurs nearly the same time cost of query execution, which is meaningless under the cardinality estimation context. (Note that cardinality estimation is conducted before query execution.)

Spatial intersection query has been widely studied for a long time [9, 28, 84]. The intersection of two polygons is determined by finding a pair of line segments intersecting with each other, where each segment belongs to a different polygon. Recent works [26,

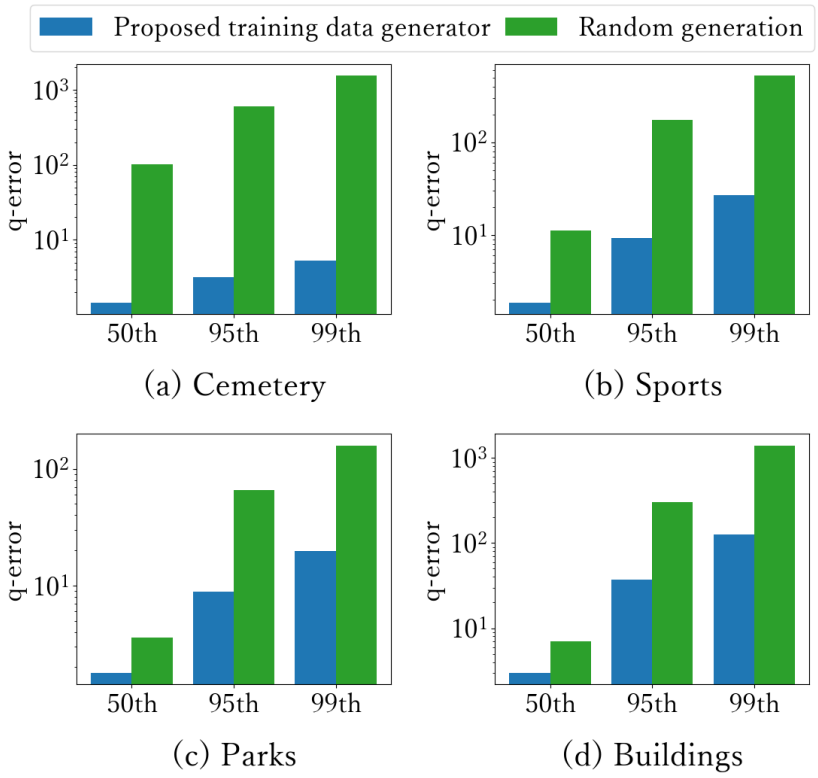


Figure 4.12: The proposed training data generator helps achieve good accuracy: the estimation errors of PolyCard trained on the datasets generated by the generator and the random generation.

63, 79] mainly focus on accelerating intersection query processing by parallelization. These techniques are not available for our problem, because running an intersection query to estimate its result size is meaningless. Our problem is useful for making an efficient execution schedule of many intersection queries.

4.6 Conclusion

Spatial objects like polygons are prevalent in spatial applications. Cardinality estimation is essential for the efficient processing of queries on spatial polygons. In this chapter, we presented PolyCard, which addressed the cardinality estimation problem of intersection queries on polygons. Existing learned estimators meet difficulty for polygons due to the variable sizes of polygons and the generation of new query polygons. Our main idea to solve these challenges is twofold: (i) making learning models compatible with

variable-sized polygons by applying adaptive sampling transformation on polygons and (ii) acquiring sufficient training data with evenly distributed cardinalities by a training data generator. Our experimental results show that our PolyCard has the following advantages compared with other competitors: (i) PolyCard improves 30% accuracy in comparison to the comparison methods, (ii) PolyCard costs only 4 microseconds for a single estimation, and (iii) PolyCard is effective on datasets of different sizes and has low high percentile q-errors.

We observe that PolyCard has the potential to be extended for other settings and tasks, and we want to make PolyCard support most spatial query types other than current intersection queries. This can be achieved by one-hot encoding of different query types. However, how to collect sufficient training queries and corresponding cardinalities remains a challenge because calculating true cardinalities of spatial queries costs a lot. Obtaining training data efficiently is desired if we extend PolyCard for various types of queries. Also, we expect our solution to be capable of most spatial types including line strings and groups of polygons as mentioned in Section 1. It is trivial to extend for line strings because they can be treated as simplified polygons. To process groups of polygons, which means a spatial object is represented by a variable number of polygons, the current transformation of PolyCard is not practical. How to represent groups of polygons for cardinality estimators is a challenging task and has not been studied by existing works. The challenge of obtaining sufficient training queries and corresponding cardinalities also exists for the extension to general spatial objects. We will focus on these two directions and make PolyCard a more general and capable solution for spatial data and objects.

Chapter 5

Conclusion

5.1 Summary

In this thesis, we have discussed learned cardinality estimation for spatial data. In Chapter 1, we introduced the importance of cardinality estimation to spatial applications. Fast and accurate cardinality estimation can accelerate query processing, and furthermore, achieve good user experiences for spatial applications. Recently proposed learned cardinality estimators have shown greatly improved performance compared with traditional methods. However, they cannot address challenges brought by spatial data: i) complex data distribution, ii) frequent data updates, and iii) variable data sizes. Aiming at fast and accurate cardinality estimation for spatial data we presented our research challenges.

Existing learned methods focus on high-dimensional relational data and complex query types by using complex neural networks. They overlook the one-dimensional case, which is prevalent in many query processing tasks. Whether learned methods are suitable for one-dimensional cardinality estimation remains a question. In Chapter 2, we proposed a learned estimator for the one-dimensional case to answer this question. We noted that the light-weight regression models adopted by recently proposed learned indexes are well-suited for the task of cardinality estimation. The light-weight regression models can approximate data distribution well in the one-dimensional case, and their structures can ensure fast estimation speeds. Compared with other solutions in evaluation, the proposed learned method is much faster and achieves relatively good accuracy, demonstrating that learned methods are promising for one-dimensional cardinality estimation and have the potential to be used for spatial (multi-dimensional) data case.

In Chapter 3, we proposed SAFE, a fast and accurate learned cardinality estimator for dynamic spatial data. Frequent data updates in the dynamic case require an update-friendly solution. We found that the light-weight regression models used in Chapter 2 can be extended for approximating multi-dimensional data. We organized light-weight models hierarchically and made them updatable by employing (i) a sampling-assisted learning for fast training, and (ii) an incremental model update strategy with the idea of "update only necessary models". Specifically, we build a quad-tree to partition the data space into cells. We can easily obtain a sample set derived from the cells. Our regression models approximate the data distribution of the sample set, instead of the original dataset, significantly speeding up the training process. Additionally, the regression models organized hierarchically are intrinsically divisible. Therefore, for a data update, we usually only need to update one regression model (if the data update leads to a sample update), which is extremely fast. We can incrementally update our regression models for data updates. Evaluations on both real-world and synthetic datasets demonstrate that SAFE outperforms the state-of-the-art estimators and achieves low-latency and accurate estimation for dynamic spatial data.

In Chapter 4, we focused on complex spatial polygons and proposed PolyCard, a learned cardinality estimator for intersection queries on polygons. Applying learning techniques to polygons represented by a variable number of vertices is challenging. We have two main ideas to make our proposed learned solution practical: (i) making learning models compatible with variable-sized polygons by performing adaptive sampling transformation on polygons and (ii) generating sufficient training data with evenly distributed cardinalities by a training data generator. Our experimental results demonstrate that PolyCard outperforms competitors with a great advantage. PolyCard is fast, accurate, and can keep stable performances on datasets of different sizes.

In summary, aiming at fast and accurate cardinality estimation for spatial data, we studied the case of one-dimensional data as a basis and proposed SAFE for dynamic spatial data, and PolyCard for intersection queries on spatial polygons. Our experiment results show that these methods can provide fast and accurate estimation. These advantages can further accelerate query processing and improve user experiences for spatial applications.

5.2 Future Work

We plan to investigate the following issues related to the current works.

5.2.1 Cardinality Estimation for Various Spatial Query Types

Currently, we have investigated cardinality estimation for range queries (Chapter 3) and intersection queries (Chapter 4) on spatial data. There exist various spatial query types in spatial applications, especially complex ones like distance-based queries and spatial joins across multiple tables. There are nearly no existing learned cardinality estimation methods designed for these complex spatial queries. Developing learned cardinality estimators for various query types is valuable for the practical use. Furthermore, we plan to develop a learned cardinality estimator compatible with most query types, achieving fast and accurate estimation.

5.2.2 Cardinality Estimation for Dynamic Spatial Objects

We have studied cardinality estimation for dynamic spatial points in Chapter 3, and for spatial polygons in Chapter 4. However, the case of dynamic complex spatial objects like polygons remains unsolved. Updates of spatial objects are frequent in applications like mobile maps. Developing an updatable learned solution for spatial objects is challenging. For example, a data-driven learned estimator needs to approximate the data distribution of frequently updated polygons. Additionally, updates of vertices belonging to polygons further complicate the approximation. We plan to find a way to capture the frequently updated data distribution and develop an update-friendly cardinality estimator for dynamic spatial objects.

5.2.3 Benchmark in Spatial DBMSs

We have demonstrated that SAFE (Chapter 3) and PolyCard (Chapter 4) outperform the state-of-the-art methods on real-world datasets. However, we also want to understand to what extent these cardinality estimators aid in query processing. While there are benchmark works on relational databases that evaluate cardinality estimation solutions integrated into DBMSs (e.g., PostgreSQL), benchmarking for spatial data has not been studied. We plan to integrate our solutions into some DBMSs and design a benchmark

framework for cardinality estimation on spatial data. We will evaluate the estimators by end-to-end query times to show their value for spatial query processing. We believe that this will be greatly helpful to other researchers working on spatial queries.

Acknowledgements

First and foremost, I would like to express my sincere gratitude to my direct advisor, Assistant Professor Daichi Amagata, for guiding me in scientific research and providing continuous mentorship throughout my life at Osaka University. Under his guidance, I learned how to identify, analyze, and solve problems logically. This thesis could not have been completed without his valuable guidance. I look up to him as a role model, and his exemplary image as a researcher will continue to inspire me in the future.

I am deeply thankful to my official advisor Professor Takahiro Hara, who gave me the invaluable opportunity to start my database research from scratch at Osaka University. He supported my freedom in scientific exploration. His insightful feedback and encouragement have been invaluable throughout my studies at Osaka University, and have also given me greater confidence to face future challenges.

I am grateful to Associate Professor Yuya Sasaki for his care and guidance regarding my research and future development. His support in both my academic and personal life is also invaluable. Besides, he substantially helped improve my thesis quality as a committee member of the thesis.

I want to express my gratitude to Associate Professor Keichi Takahashi for serving as a committee member of my thesis. His insightful and constructive comments significantly improved the quality of the thesis.

I would also like to acknowledge Professor Takuya Maekawa for his invaluable assistance in both my research and laboratory life. His advice on my research projects has significantly contributed to the progress of my work.

I would like to acknowledge my laboratory buddies, Dr. Yiming Tian, Dr. Qingxin Xia, Dr. Naoya Yoshimura, Dr. Thilina Dissanayake, Dr. Zhi Li, Dr. Heng Zhou, Mr. Jaime Morales, Mr. Guanyu Cao, Mr. Zesheng Cai, Ms. Yue Zhao, Mr. Yusuke Arai, Mr. Yuki Nishino, Mr. Kei Tanigaki, Mr. Ryo Shirai, Mr. Kazuyoshi Aoyama, Mr. Aoran Chen, Mr. Shengzhe Jiao, Mr. Yuqiao Wang, Mr. Yiqing Zhang, Mr. Yang Wang, Ms. Hongyin Qiao, and Mrs. Makiko Higashinaka. It is my pleasure to work with all

members of Hara laboratory. They have been kind and cooperative throughout my life in Hara laboratory.

Finally, I owe my deepest gratitude to my parents, whose unwavering support and encouragement have been the foundation of my academic journey. Their belief in my abilities has been a constant source of strength and motivation, inspiring me to persevere through challenges. Their sacrifices and love have supported my journey at Osaka University, and for that, I am eternally grateful.

References

[1] Ashraf Aboulnaga and Surajit Chaudhuri. 1999. Self-tuning Histograms: Building Histograms without Looking at Data. *SIGMOD Record* 28, 2 (1999), 181–192.

[2] Ildar Absalyamov, Michael J Carey, and Vassilis J Tsotras. 2018. Lightweight Cardinality Estimation in LSM-based Systems. In *SIGMOD*. 841–855.

[3] Ablimit Aji, Fusheng Wang, and Joel H Saltz. 2012. Towards Building a High Performance Spatial Query System for Large Scale Medical Imaging Data. In *SIGSPATIAL*. 309–318.

[4] Daichi Amagata, Yusuke Arai, Sumio Fujita, and Takahiro Hara. 2022. Learned k-NN Distance Estimation. In *SIGSPATIAL*. 1–4.

[5] Daichi Amagata and Takahiro Hara. 2016. Monitoring MaxRS in Spatial Data Streams. In *EDBT*. 317–328.

[6] Daichi Amagata and Takahiro Hara. 2017. A General Framework for MaxRS and MaxCRS Monitoring in Spatial Data Streams. *ACM Transactions on Spatial Algorithms and Systems* 3, 1 (2017), 1–34.

[7] Daichi Amagata and Takahiro Hara. 2019. Identifying the Most Interactive Object in Spatial Databases. In *ICDE*. 1286–1297.

[8] Alberto Belussi, Elisa Bertino, and Andrea Nucita. 2004. Grid Based Methods for Estimating Spatial Join Selectivity. In *SIGSPATIAL*. 92–100.

[9] Bentley and Ottmann. 1979. Algorithms for Reporting and Counting Geometric Intersections. *IEEE Transactions on computers* 100, 9 (1979), 643–647.

[10] Jon Louis Bentley. 1975. Multidimensional Binary Search Trees Used for Associative Searching. *Commun. ACM* 18, 9 (1975), 509–517.

[11] Panagiotis Bouros and Nikos Mamoulis. 2019. Spatial Joins: What's Next? *SIGSPATIAL Special* 11, 1 (2019), 13–21.

[12] Surajit Chaudhuri. 1998. An Overview of Query Optimization in Relational Systems. In *PODS*. 34–43.

[13] Jeremy Chen, Yuqing Huang, Mushi Wang, Semih Salihoglu, and Ken Salem. 2022. Accurate Summary-Based Cardinality Estimation Through the Lens of Cardinality Estimation Graphs. *PVLDB* 15, 8 (2022), 1533–1545.

[14] Octav Chipara, Chenyang Lu, and Gruia-Catalin Roman. 2012. Real-Time Query Scheduling for Wireless Sensor Networks. *IEEE transactions on computers* 62, 9 (2012), 1850–1865.

[15] Zhaole Chu, Zhou Zhang, Peiquan Jin, Xiaoliang Wang, Yongping Luo, and Xujian Zhao. 2024. LIVAK: A High-Performance In-Memory Learned Index for Variable-Length Keys. In *DAC*. 1–6.

[16] Ronan Collobert and Samy Bengio. 2004. Links between Perceptrons, MLPs and SVMs. In *ICML*. 23.

[17] Douglas Comer. 1979. Ubiquitous B-Tree. *Comput. Surveys* 11, 2 (1979), 121–137.

[18] Graham Cormode, Minos Garofalakis, Peter J. Haas, and Chris Jermaine. 2012. Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches. *Found. Trends Databases* 4, 1–3 (2012), 1–294.

[19] Andrew Crotty. 2021. Hist-Tree: Those Who Ignore It Are Doomed to Learn. In *CIDR*.

[20] Mark Derthick, James Harrison, Andrew Moore, and Steven F Roth. 1999. Efficient Multi-Object Dynamic Query Histograms. In *IEEE Symposium on Information Visualization*. 84–91.

[21] Jialin Ding, Vikram Nathan, Mohammad Alizadeh, and Tim Kraska. 2020. Tsunami: A Learned Multi-Dimensional Index for Correlated Data and Skewed Workloads. *PVLDB* 14, 2 (2020), 74–86.

[22] Anshuman Dutt, Chi Wang, Azade Nazi, Srikanth Kandula, Vivek Narasayya, and Surajit Chaudhuri. 2019. Selectivity Estimation for Range Predicates Using Lightweight Models. *PVLDB* 12, 9 (2019), 1044–1057.

[23] Ahmed Eldawy and Mohamed F Mokbel. 2015. The Era of Big Spatial Data. In *ICDE Workshops*. 42–49.

[24] Ahmed Eldawy and Mohamed F Mokbel. 2015. Spatialhadoop: A Mapreduce Framework for Spatial Data. In *ICDE*. 1352–1363.

[25] Raphael A Finkel and Jon Louis Bentley. 1974. Quad Trees: A Data Structure for Retrieval on Composite Keys. *Acta informatica* 4 (1974), 1–9.

[26] Roger Frye and Mark McKenney. 2022. Per Segment Plane Sweep Line Segment Intersection on the GPU. In *SIGSPATIAL*. 1–4.

[27] Thanasis Georgiadis and Nikos Mamoulis. 2023. Raster Intervals: An Approximation Technique for Polygon Intersection Joins. *SIGMOD* 1, 1 (2023), 1–18.

[28] Günther Greiner and Kai Hormann. 1998. Efficient Clipping of Arbitrary Polygons. *ACM Transactions on Graphics* 17, 2 (1998), 71–83.

[29] Andrey Gubichev and Thomas Neumann. 2014. Exploiting the Query Structure for Efficient Join Ordering in SPARQL Queries. In *EDBT*. 439–450.

[30] Amit Gupta, S Sudarshan, and Sundar Vishwanathan. 2001. Query Scheduling in Multi Query Optimization. In *International Database Engineering and Applications Symposium*. 11–19.

[31] Eric Haines. 1994. Point in Polygon Strategies. *Graphics Gems* 4, 1 (1994), 24–46.

[32] Jun Han and Claudio Moraga. 1995. The Influence of the Sigmoid Function Parameters on the Speed of Backpropagation Learning. In *International Workshop on Artificial Neural Networks*. 195–201.

[33] Yuxing Han, Ziniu Wu, Peizhi Wu, Rong Zhu, Jingyi Yang, Liang Wei Tan, Kai Zeng, Gao Cong, Yanzhao Qin, Andreas Pfadler, Zhengping Qian, Jingren Zhou, Jiangneng Li, and Bin Cui. 2021. Cardinality Estimation in DBMS: A Comprehensive Benchmark Evaluation. *PVLDB* 15, 4 (2021), 752–765.

[34] Hazar Harmouch and Felix Naumann. 2017. Cardinality Estimation: An Experimental Survey. *PVLDB* 11, 4 (2017), 499–512.

[35] Shohedul Hasan, Saravanan Thirumuruganathan, Jees Augustine, Nick Koudas, and Gautam Das. 2020. Deep Learning Models for Selectivity Estimation of Multi-Attribute Queries. In *SIGMOD*. 1035–1050.

[36] Mahdi Hashemi. 2019. Enlarging Smaller Images before Inputting into Convolutional Neural Network: Zero-Padding vs. Interpolation. *Journal of Big Data* 6, 1 (2019), 1–13.

[37] Shuhei Hayashida, Daichi Amagata, Takahiro Hara, and Xing Xie. 2018. Dummy Generation Based on User-Movement Estimation for Location Privacy Protection. *IEEE Access* 6 (2018), 22958–22969.

[38] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. 2020. DeepDB: Learn from Data, not from Queries! *PVLDB* 13, 7 (2020), 992–1005.

[39] Keizo Hori, Yuya Sasaki, Daichi Amagata, Yuki Murosaki, and Makoto Onizuka. 2023. Learned Spatial Data Partitioning. In *International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*. 1–8.

[40] Xiao Hu, Yuxi Liu, Haibo Xiu, Pankaj K Agarwal, Debmalya Panigrahi, Sudeepa Roy, and Jun Yang. 2022. Selectivity Functions of Range Queries are Learnable. In *SIGMOD*. 959–972.

[41] Yannis Ioannidis. 2003. The History of Histograms (Abridged). In *VLDB*. 19–30.

[42] Edwin H Jacox and Hanan Samet. 2007. Spatial Join Techniques. *ACM Transactions on Database Systems* 32, 1 (2007), 7–es.

[43] Ramesh Janipella, Vikash Gupta, and Rucha V Moharir. 2019. Application of Geographic Information System in Energy Utilization. In *Current Developments in Biotechnology and Bioengineering*. 143–161.

[44] Yuchen Ji, Daichi Amagata, Yuya Sasaki, and Takahiro Hara. 2022. A Performance Study of One-Dimensional Learned Cardinality Estimation. In *DOLAP*. 86–90.

[45] Yuchen Ji, Daichi Amagata, Yuya Sasaki, and Takahiro Hara. 2024. Fast Learned Cardinality Estimation for Dynamic Spatial Data. In *Proc. of DEIM Forum*.

[46] Yuchen Ji, Daichi Amagata, Yuya Sasaki, and Takahiro Hara. 2024. Learned Cardinality Estimator for Intersection Queries on Two-dimensional Polygon Data. In *IPSJ SIG-DBS / IPSJ SIG-IFAT / IEICE DE*.

[47] Yuchen Ji, Daichi Amagata, Yuya Sasaki, and Takahiro Hara. 2024. SAFE: Sampling-Assisted Fast Learned Cardinality Estimation for Dynamic Spatial Data. In *DEXA*. 201–216.

[48] Yuchen Ji, Daichi Amagata, Yuya Sasaki, and Takahiro Hara. 2025. Efficient Learned Cardinality Estimation for Dynamic Spatial Data. *IEICE Transactions on Information and Systems* (2025).

[49] Ji Jin, Ning An, and Anand Sivasubramaniam. 2000. Analyzing Range Queries on Spatial Data. In *ICDE*. 525–534.

[50] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *ICLR*.

[51] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter A. Boncz, and Alfons Kemper. 2019. Learned Cardinalities: Estimating Correlated Joins with Deep Learning. In *CIDR*.

[52] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. 2020. RadixSpline: A Single-Pass Learned Index. In *aiDM*. Article 5.

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

[54] Suyong Kwon, Woohwan Jung, and Kyuseok Shim. 2022. Cardinality Estimation of Approximate Substring Queries Using Deep Learning. *PVLDB* 15, 11 (2022), 3145–3157.

[55] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *PVLDB* 9, 3 (2015), 204–215.

[56] Fred H Lesh. 1959. Multi-Dimensional Least-Squares Polynomial Curve Fitting. *Commun. ACM* 2, 9 (1959), 29–30.

[57] Beibin Li, Yao Lu, and Srikanth Kandula. 2022. Warper: Efficiently Adapting Learned Cardinality Estimators to Data and Workload Drifts. In *SIGMOD*. 1920–1933.

[58] Guoliang Li, Xuanhe Zhou, and Lei Cao. 2021. AI Meets Database: AI4DB and DB4AI. In *SIGMOD*. 2859–2866.

[59] Pengfei Li, Hua Lu, Qian Zheng, Long Yang, and Gang Pan. 2020. LISA: A Learned Index Structure for Spatial Data. In *SIGMOD*. 2119–2133.

[60] Henry Liu, Mingbin Xu, Ziting Yu, Vincent Corvinelli, and Calisto Zuzarte. 2015. Cardinality Estimation Using Neural Networks. In *International Conference on Computer Science and Software Engineering*. 53–59.

[61] Jie Liu, Wenqian Dong, Qingqing Zhou, and Dong Li. 2021. Fauce: Fast and Accurate Deep Ensembles with Uncertainty for Cardinality Estimation. *PVLDB* 14, 11 (2021), 1950–1963.

[62] Qiyu Liu, Yanyan Shen, and Lei Chen. 2021. LHist: Towards Learning Multi-Dimensional Histogram for Massive Spatial Data. In *ICDE*. 1188–1199.

[63] Yiming Liu and Satish Puri. 2020. Efficient Filters for Geometric Intersection Computations Using GPU. In *SIGSPATIAL*. 487–496.

[64] Stephen Macke, Alex Beutel, Tim Kraska, Maheswaran Sathiamoorthy, Derek Zhiyuan Cheng, and Ed H Chi. 2018. Lifting the Curse of Multidimensional Data with Learned Existence Indexes. In *Workshop on ML for Systems*. 6.

[65] Marcel Maltry and Jens Dittrich. 2022. A Critical Analysis of Recursive Model Indexes. *PVLDB* 15, 5 (2022), 1079–1091.

[66] Marcel Maltry and Jens Dittrich. 2022. A Critical Analysis of Recursive Model Indexes. *PVLDB* 15, 5 (2022), 1079–1091.

[67] Nikos Mamoulis, Dimitris Papadias, and Dinos Arkoumanis. 2004. Complex Spatial Query Processing. *GeoInformatica* 8 (2004), 311–346.

[68] Ryan Marcus, Andreas Kipf, Alexander van Renen, Mihail Stoia, Sanchit Misra, Alfons Kemper, Thomas Neumann, and Tim Kraska. 2021. Flow-Loss: Learning Cardinality Estimates That Matter. *PVLDB* 14, 11 (2021), 2019–2032.

[69] Ryan Marcus, Andreas Kipf, Alexander van Renen, Mihail Stoian, Sanchit Misra, Alfons Kemper, Thomas Neumann, and Tim Kraska. 2020. Benchmarking Learned Indexes. *PVLDB* 14, 1 (2020), 1–13.

[70] Volker Markl, Peter J Haas, Marcel Kutsch, Nimrod Megiddo, Utkarsh Srivastava, and Tam Minh Tran. 2007. Consistent Selectivity Estimation via Maximum Entropy. *The VLDB journal* 16 (2007), 55–76.

[71] Zizhong Meng, Peizhi Wu, Gao Cong, Rong Zhu, and Shuai Ma. 2022. Unsupervised Selectivity Estimation by Integrating Gaussian Mixture Models and an Autoregressive Model. In *EDBT*. 2–247.

[72] Songsong Mo, Yile Chen, Hao Wang, Gao Cong, and Zhifeng Bao. 2023. Lemo: A Cache-Enhanced Learned Optimizer for Concurrent Queries. In *SIGMOD*. 1–26.

[73] Moin Hussain Moti, Panagiotis Simatis, and Dimitris Papadias. 2022. Waffle: A Workload-Aware and Query-Sensitive Framework for Disk-Based Spatial Indexing. *PVLDB* 16, 4 (2022), 670–683.

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

[75] Parimarjan Negi, Ziniu Wu, Andreas Kipf, Nesime Tatbul, Ryan Marcus, Sam Madden, Tim Kraska, and Mohammad Alizadeh. 2023. Robust Query Driven Cardinality Estimation under Changing Workloads. *PVLDB* 16, 6 (2023), 1520–1533.

[76] Shunya Nishio, Daichi Amagata, and Takahiro Hara. 2022. Lamps: Location-Aware Moving Top-k Pub/Sub. *IEEE Transactions on Knowledge and Data Engineering* 34, 1 (2022), 352–364.

[77] Varun Pandey, Alexander van Renen, Andreas Kipf, and Alfons Kemper. 2021. How Good are Modern Spatial Libraries? *Data Science and Engineering* 6, 2 (2021), 192–208.

[78] Yongjoo Park, Shucheng Zhong, and Barzan Mozafari. 2020. QuickSel: Quick Selectivity Learning with Mixture Models. In *SIGMOD*. 1017–1033.

[79] Satish Puri, Dinesh Agarwal, Xi He, and Sushil K Prasad. 2013. MapReduce Algorithms for GIS Polygonal Overlay Processing. In *IEEE International Symposium on Parallel & Distributed Processing*. 1009–1016.

[80] Jianzhong Qi, Guanli Liu, Christian S Jensen, and Lars Kulik. 2020. Effectively Learning Spatial Indices. *PVLDB* 13, 12 (2020), 2341–2354.

[81] Philippe Rigaux, Michel Scholl, and Agnes Voisard. 2001. *Spatial Databases: with Application to GIS*.

[82] Ibrahim Sabek, Tenzin Samten Ukyab, and Tim Kraska. 2022. Lsched: A Workload-Aware Learned Query Scheduler for Analytical Database Systems. In *SIGMOD*. 1228–1242.

[83] Yuya Sasaki. 2021. A Survey on IoT Big Data Analytic Systems: Current and Future. *IEEE Internet of Things Journal* 9, 2 (2021), 1024–1036.

[84] Michael Ian Shamos and Dan Hoey. 1976. Geometric Intersection Problems. In *Annual Symposium on Foundations of Computer Science*. 208–215.

[85] Michael Shekelyan, Anton Dignös, Johann Gamper, and Minos Garofalakis. 2021. Approximating Multidimensional Range Counts with Maximum Error Guarantees. In *ICDE*. 1595–1606.

[86] Yexuan Shi, Yongxin Tong, Yuxiang Zeng, Zimu Zhou, Bolin Ding, and Lei Chen. 2023. Efficient Approximate Range Aggregation over Large-Scale Spatial Data Federation. *IEEE Transactions on Knowledge and Data Engineering* 35, 1 (2023), 418–430.

[87] Mihail Stoian, Andreas Kipf, Ryan Marcus, and Tim Kraska. 2021. PLEX: Towards Practical Learned Indexing. In *AIDB*.

[88] Ji Sun, Guoliang Li, and Nan Tang. 2021. Learned Cardinality Estimation for Similarity Queries. In *SIGMOD*. 1745–1757.

[89] Luming Sun, Cuiping Li, Tao Ji, and Hong Chen. 2023. MOSE: A Monotonic Selectivity Estimator Using Learned CDF. *IEEE Transactions on Knowledge and Data Engineering* 35, 3 (2023), 2823–2836.

[90] Mingjie Tang, Yongyang Yu, Qutaibah M. Malluhi, Mourad Ouzzani, and Walid G Aref. 2016. LocationSpark: A Distributed In-Memory Data Management System for Big Spatial Data. *PVLDB* 9, 13 (2016), 1565–1568.

[91] Ryosuke Taniguchi, Daichi Amagata, and Takahiro Hara. 2022. Efficient Retrieval of Top-k Weighted Spatial Triangles. In *DASFAA*. 224–231.

[92] Ryosuke Taniguchi, Daichi Amagata, and Takahiro Hara. 2022. Efficient Retrieval of Top-k Weighted Triangles on Static and Dynamic Spatial Data. *IEEE Access* 10 (2022), 55298–55307.

[93] Cees J Van Westen, Enrique Castellanos, and Sekhar L Kuriakose. 2008. Spatial Data for Landslide Susceptibility, Hazard, and Vulnerability Assessment: An Overview. *Engineering Geology* 102, 3-4 (2008), 112–131.

[94] Tin Vu, Alberto Belussi, Sara Migliorini, and Ahmed Eldawy. 2021. A Learned Query Optimizer for Spatial Join. In *SIGSPATIAL*. 458–467.

[95] Tin Vu, Alberto Belussi, Sara Migliorini, and Ahmed Eldway. 2020. Using Deep Learning for Big Spatial Data Partitioning. *ACM Transactions on Spatial Algorithms and Systems* 7, 1 (2020), 1–37.

[96] Tin Vu, Ahmed Eldawy, Vagelis Hristidis, and Vassilis Tsotras. 2021. Incremental Partitioning for Efficient Spatial Data Analytics. *PVLDB* 15, 3 (2021), 713–726.

[97] Jiayi Wang, Chengliang Chai, Jiabin Liu, and Guoliang Li. 2021. FACE: A Normalizing Flow Based Cardinality Estimator. *PVLDB* 15, 1 (2021), 72–84.

[98] Xiaoying Wang, Changbo Qu, Weiyuan Wu, Jiannan Wang, and Qingqing Zhou. 2021. Are We Ready for Learned Cardinality Estimation? *PVLDB* 14, 9 (2021), 1640–1654.

[99] Yaoshu Wang, Chuan Xiao, Jianbin Qin, Xin Cao, Yifang Sun, Wei Wang, and Makoto Onizuka. 2020. Monotonic Cardinality Estimation of Similarity Selection: A Deep Learning Approach. In *SIGMOD*. 1197–1212.

[100] Yaoshu Wang, Chuan Xiao, Jianbin Qin, Rui Mao, Makoto Onizuka, Wei Wang, Rui Zhang, and Yoshiharu Ishikawa. 2021. Consistent and Flexible Selectivity Estimation for High-Dimensional Data. In *SIGMOD*. 2319–2327.

[101] Edward J Wegman. 1969. *Nonparametric Probability Density Estimation*. Technical Report. North Carolina State University. Dept. of Statistics.

[102] Lucas Woltmann, Dominik Olwig, Claudio Hartmann, Dirk Habich, and Wolfgang Lehner. 2021. PostCENN: Postgresql with Machine Learning Models for Cardinality Estimation. *PVLDB* 14, 12 (2021), 2715–2718.

[103] Jiacheng Wu, Yong Zhang, Shimin Chen, Jin Wang, Yu Chen, and Chunxiao Xing. 2021. Updatable Learned Index with Precise Positions. *PVLDB* 14, 8 (2021), 1276–1288.

[104] Peizhi Wu and Gao Cong. 2021. A Unified Deep Model of Learning from both Data and Queries for Cardinality Estimation. In *SIGMOD*. 2009–2022.

[105] Wentao Wu, Yun Chi, Shenghuo Zhu, Junichi Tatemura, Hakan Hacigümüs, and Jeffrey F Naughton. 2013. Predicting Query Execution Time: Are Optimizer Cost Models Really Unusable?. In *ICDE*. 1081–1092.

[106] Dong Xie, Feifei Li, Bin Yao, Gefei Li, Liang Zhou, and Minyi Guo. 2016. Simba: Efficient In-Memory Spatial Analytics. In *SIGMOD*. 1071–1085.

[107] Yifan Yang and Shimin Chen. 2024. LITS: An Optimized Learned Index for Strings. *PVLDB* 17, 11 (2024), 3415–3427.

[108] Zongheng Yang, Badrish Chandramouli, Chi Wang, Johannes Gehrke, Yinan Li, Umar Farooq Minhas, Per-Åke Larson, Donald Kossmann, and Rajeev Acharya. 2020. Qd-Tree: Learning Data Layouts for Big Data Analytics. In *SIGMOD*. 193–208.

[109] Zongheng Yang, Amog Kamsetty, Sifei Luan, Eric Liang, Yan Duan, Xi Chen, and Ion Stoica. 2020. NeuroCard: One Cardinality Estimator for All Tables. *PVLDB* 14, 1 (2020), 61–73.

[110] Zongheng Yang, Eric Liang, Amog Kamsetty, Chenggang Wu, Yan Duan, Xi Chen, Pieter Abbeel, Joseph M. Hellerstein, Sanjay Krishnan, and Ion Stoica. 2019. Deep Unsupervised Cardinality Estimation. *PVLDB* 13, 3 (2019), 279–292.

[111] Jason Yosinski, Jeff Clune, Anh Nguyen, Thomas Fuchs, and Hod Lipson. 2015. Understanding Neural Networks Through Deep Visualization. In *Deep Learning Workshop on ICML*. 448–456.

[112] Jiahui Yu and Konstantinos Spiliopoulos. 2023. Normalization Effects on Deep Neural Networks. *Foundations of Data Science* 5, 3 (2023), 389–465.

[113] Xiaohui Yu and Ada Fu. 2001. Piecewise Linear Histograms for Selectivity Estimation. In *International Symposium on Information Systems and Engineering*. 319–326.

[114] Yongluan Zhou, Ji Wu, and Ahmed Khan Leghari. 2013. Multi-Query Scheduling for Time-Critical Data Stream Applications. In *SSDBM*. 1–12.

[115] Chong Zhu, Gopalakrishnan Sundaram, Jack Snoeyink, and Joseph SB Mitchell. 1996. Generating Random Polygons with Given Vertices. *Computational Geometry* 6, 5 (1996), 277–290.