CSC 2508 – Final Project

Stephanie Knill
sknill@cs.toronto.edu
1006179181[1]

[1]Department of Computer Science, The University of Toronto

January 4, 2021

Abstract

In this paper I will be implementing the novel Learned Index Structure for Spatial Data (LISA) from Li *et al.* [8]. The motivation for this new index comes from the high storage consumption and IO cost associated with the traditional R-tree for spatial query processing. By replacing the existing R-tree structure with a machine learning model, learned indices [7] have the potential to address these shortcomings. The construction of LISA consists of four parts: 1) partiting the data into grid cells; 2) a partially monotonic mapping function $\mathcal{M}$ that maps spatial data into 1-dimensional space; 3) a learned shard prediction function $\mathcal{SP}$ that partitions these mapped values into shards; and 4) local models $\mathcal{L}$ that manage the inter-shard operations. Rather than fixing the data layout then learning the model to approximate the layout, LISA uses a learned model to generate the data layout (shards), thereby being the first learned index to support data updates and K-NN queries.

1 Introduction

Since most data is multidimensional, the R-tree was proposed in 1984 to be a N-dimensional extension of B^+-trees, thereby able to handle geometrical data including points, line segments, surfaces, volumes, and hypervolumes. By organizing data objects though a hierarchical organization of minimum bounding rectangles (MBR), this structure supports insertions, deletions and enables efficient query processing [4]. Much research has been conducted to improve upon R-trees, including the R^+-trees [16], R^*-trees [1], and Hilbert R-tree [5]. With the expansion of spatial databases and geographical information systems fields, as well as the modern prevalence of multimedia databases of images, voice, music, or video, this has led to the proliferation of R-tree variation research for data storage and retrieval [9]. The next logical step in this research progression is to utilize the latest buzz of machine learning to improve upon the traditional R-tree structures.

A common database operation is a range query, which retrieves all records that satisfy an upper and lower boundary of values. Another common operation is k-Nearest Neighbours: given a dataset and query point $P, q \in \mathbb{R}^d$, we want to find the k closest points in P to the given query point q [13]. In order to construct an efficient Nearest Neighbour search data structure, there are two major approaches:

1. **Indexing**: construct a data structure such that given a query point, it produces a small subset of P (candidate set) that includes the nearest neighbours
2. **Sketching**: compute a compressed representation of points such that computations of approximate distances can be performed quickly

Since we are building an index structure, we will only focus on research in this approach.

1.1 Spatial Indexes

Since database researchers are highly organized, they have impressively refined this approach into 3 categories:

1. **Space Partitioning**: partition the space into regions then index those regions
2. **Data Partitioning**: partition the dataset into subsets then index those subsets
3. **Mapping to 1D Space**: the multi-dimensional space is transformed to a 1-dimensional space. The data objects are then sequentially ordered into regions which are each indexed by a B^+-tree.

Within space partitioning, we have KD-trees [2], quadtrees [3], and octrees [10]. In data partitioning we have our R-tree and its many variants. Examples of our third category include the Microsoft SQL Server [11], MongoDB [12], and UB-tree [14].

1.2 Learned Indexes

A much newer subfield of research, learned indexes seek to replace existing index structures with deep learning models. The premise of this field lies on the idea that indexes are already to a large extent a learned model, hence they are well suited to be replaced by a machine learning (ML) model, including neural networks. A B-Tree can be viewed as a model, specifically a regression tree: it maps a key to a position with a min- and max-error (of 0 and page-size respectively) with a guarantee that the key can be found in the region if it exists. As long

as the error guarantees can be maintained, then the B-Tree index can be replaced by a ML model. For new data, B-Trees need to be re-balanced—or in machine learning terminology, re-trained—so it maintains its error guarantees [7].

This replacement can be achieved by capturing the relationship between search keys and their positions in the database through cumulative density function (CDFs). Thus we can replace the traditional B-trees with a recursive model index (RMI) that places simple models in a hierarchical organization [8].

While a good first step, the RMI still has many shortcomings, including its inability to handle spatial data because of the 1-dimensional data assumption. Although you can theoretically learn multi-dimensional CDFs or transform multi-dimensional data into 1-dimensional data [6], these raise new challenges. While possible to transform multi-dimensional data, the success of this transformation relies heavily on knowledge of the underlying dataset hence it is hard to apply this method to arbitrary new datasets. There have been some progress, however. Since traditional existence filters make no use of the distribution of keys, one group of researchers used a learned Bloom filter variation that can by featuring two layers of Bloom filters surrounding the learned function [7]. As well, a learned Z-order Model (ZM) index has been proposed, which combines the Z-order space filling curve and the staged learning model. Although this learned index significantly reduces the memory cost and performs more efficiently than R-Trees in most scenarios, it unfortunately checks many irrelevant keys for range queries and does not support data updates or K-NN queries [18].

2 Proposed Methodology

Building upon the existing research of learned indexes in spatial data, we will be implementing Li *et al.*'s novel Learned Index structure for Spatial Data (LISA) [8]. As of publication in June 2020, it is (to the author's knowledge) the first fully-fledged learned index that handles spatial data whilst supporting data updates and K-NN queries.

2.1 Construction of Index

The components of LISA we will be implementing for the construction of the index are

1. **Generation of Grid Cells**: partition the space into a series of grid cells based on the data distribution along a sequence of axes. After generation, we then number the cells along these axes.

2. **Mapping Function $\mathcal{M}$**: based on the cell borders from step (1), we map the data from $\mathbb{R}^d$ into $\mathbb{R}$ such that it is partially monotonic. This means that if keys x_0 and x_1 are in the cells C_i and C_j respectively for $i < j$, then $\mathcal{M}(x_0) < \mathcal{M}(x_1)$.

   For a data key $\boldsymbol{x} = (x_0, \ldots, x_{d-1}) \in \mathbb{R}^d$ that is contained by the cell $C_i = [\theta_{j_0}^{(0)}, \theta_{j_0+1}^{(0)}) \times \cdots \times [\theta_{j_{d-1}}^{(d-1)}, \theta_{j_{d-1}+1}^{(d-1)})$, the mapping function is given by

   $$\mathcal{M} = i + \frac{\mu(H_i)}{\mu(C_i)},$$

   where $H_i = [\theta_{j_0}^{(0)}, x_0) \times \cdots \times [\theta_{j_{d-1}}^{(d-1)}, x_{d-1})$ and μ is the Lebesgue measure on $\mathbb{R}^d$ [15]. This takes the ratio between the volume of the cell containing the point x (the cell C_i) versus the volume of the same cell but whose upper bounds are the x_i's (the cell H_i). Since this

ratio will be from $[0, 1]$ and we increment this by the cell number we are able to achieve an elegant partially monotonic property that yields a tight but fairly even distribution.

3. **Monotonic Shard Prediction Function $\mathcal{SP}$**: we learn a series of learned piecewise linear functions that assign a shard id to each mapped value. By tuning $\mathcal{SP}$'s parameters, we can ensure that each shard contains a similar number of keys. For a point $\boldsymbol{x} \in \mathbb{R}^d$ with mapped value $m \in \mathbb{R}$, our shard prediction function $\mathcal{SP} : \mathbb{R} \to [0, +\infty)$ is defined as

$$\mathcal{SP}(m) = \mathcal{F}_i(x) + i \times D$$

where i is the index of the mapped partition containing m (conducted by a binary search), D is the number of shards in partition i, and $\mathcal{F}_i = \frac{f_i(m)}{\Psi}$, with Ψ being the average number of keys in the initial dataset falling in a shard and f_i the learned piecewise function. For our learned piecewise function f_i, its formation is quite complex and so the exact details of it will be left to [8]. We will, however, go over how it is trained. The piecewise function f_i consists of $\sigma+1$ breakpoints (or pieces) and is tuned by the vectors $\boldsymbol{\alpha} = (\bar{\alpha}, \alpha_0, \alpha_1, \ldots, \alpha_\sigma)$ and $\boldsymbol{\beta} = (\beta_0, \ldots, \beta_\sigma)$, whose values are restricted to ensure f_i is monotonically increasing. The square loss function optimized for is

$$L(\boldsymbol{\alpha}, \boldsymbol{\beta}) = \sum_{i=0}^{V} (f(x_i) - y_i)^2$$

where $V + 1$ is the number of mapped values each $\mathcal{F}_i$ is to process, and y_i is the index corresponding to the mapped value x_i. By fixing $\boldsymbol{\beta}$, we can solve for $\boldsymbol{\alpha}$. This can be achieved by expressing our system as the linear equation $\boldsymbol{A\alpha} = \boldsymbol{y}$ and solving for $\boldsymbol{\alpha}$. Fortunately, the matrix $\boldsymbol{A}^T\boldsymbol{A}$ is symmetric and so we were able to efficiently compute it using the Cholesky algorithm [17].

Using an initialization of $\boldsymbol{\beta}$ and the gradient computation specified in [8], we now perform a gradient descent search, iteratively updating our $\boldsymbol{\beta}$. For the learning rate $lr \geq 0$, we used a brute force grid search to find the learning rate that yielded the update (with its corresponding $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ that are valid) that best minimized the loss function. This continues until either the loss function converges or no learning rates produce a valid $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ assignment.

After the shard prediction function $\mathcal{SP}$ has been trained, we can assign a point $\boldsymbol{x} \in \mathbb{R}^d$ with mapped value m the shard id $\lfloor \mathcal{SP}(m) \rfloor$.

4. **Construction of Local Models $\mathcal{L}$**: build a local model $\mathcal{L}_i$ that manages the keys for each shard $\mathcal{S}_i$

   - The keys that fall in the same shard are stored in one or more consecutive disk pages. Conversely, two keys with different shard ids are stored in two different pages.
   - Local models are responsible for 1) allocating pages to store keys and 2) updating data by splitting or merging pages

   Each local model $\mathcal{L}_i$ contains three attributes (name, PA, and PM) and two methods (lbound and ubound). $\mathcal{L}_i.PA$ is a list of the addresses of the pages that overlap with the corresponding shard S_i, whereas $\mathcal{L}_i.PM$ is a sequence of sorted mapped values that partition the keys in S_i into different pages. Together these allow a user to perform binary searches ($\mathcal{L}_i.lbound$ and $\mathcal{L}_i.ubound$) in order to support efficient range queries.

Unlike RMI which first fixes the data layout and then learns the model to approximate the layout, LISA uses a learned model to generate the data layout (shards). This enables a learned index to now support data updates, including insertion and deletion. So once the index is constructed, the mapping function $\mathcal{M}$ and shard prediction function $\mathcal{S}$ is fixed. The local models, however, are dynamic: $\mathcal{L}_i.PA$ and $\mathcal{L}_i.PM$ may change as insertions and deletions are performed.

2.2 Query Processing

Although able to handle data insertion and deletion, for brevity we will only be examining the performance of LISA for range and k-NN queries

- **Range Queries**: given a query rectangle qr, we perform a range query on teh index to obtain the key that fall in qr. Here we use a novel algorithm that efficiently optimizes the structure of the index that we just built:
  1. Obtain all grid cells that overlap with qr
  2. Decompose qr into a union of cells that each only intersect one grid cell from step (1)
  3. If there are now many small rectangles that are connected, merge them
  4. For each sub-cell of qr:
     (a) Mapping Function $\mathcal{M}$: calculate the mapped value of each vertex of the cell
     (b) Shard Prediction Function $\mathcal{SP}$: calculate the shard id for each of the mapped values
     (c) Local Model $\mathcal{L}$: using the local model that manages the chosen shard, select all the pages that overlap with qr
  5. Now that we have appropriate pages, we go through each page and obtain the keys that fall within qr
- **K-NN Queries**: use a lattice regression model $\mathcal{LR}$ that processes a K-NN query as a series of range queries
  - $\mathcal{LR}$ estimates a range query's rectangle size for a given query point and number of neighbours k
  - A range query with that estimated rectangle size is then executed.
  - If the range query returns less than k nearest neighbours, then the estimated range query's rectangle size is augmented and a new range query is performed. This process continues until a sufficient number of neighbours are found.

Unfortunately due to time constraints, I was unable to finish my project. I have completed the implementation of the index and the range query (but no K-NN query). The code for this can be seen on Github[1].

[1] https://github.com/stephanie-knill/CSC2508_LISA_Implementation

3 Experiments

3.1 Dataset

The dataset used for this project is the imis-3months. This was chosen due to its size, straightforward processing and Li *et al.* also using it. The imis-3months consists of approximately 180 million data points, with each record containing values for {`t, obj_id, lon, lat`}. After removing `t` and `obj_id` from our dataset and eliminating duplicates, we were left with approximately 106 million spatial data points of longitude and latitude.

3.2 Parameter Settings

Table 1 shows the parameter settings used for our experiments. For the maximum number of keys page (Ω), the estimated number of keys per shard (Ψ), and the number of breakpoints (Ω), these were chosen based on the computations in Li *et al.*. The remaining parameters were unfortunately chosen based on the extent my computer protested. For the training of the shard prediction function $\mathcal{SP}$, the learning rates used in the brute force gird search were $1, 0.5, 0.4, 0.3, 0.2, 0.1, 0.01, 0.001, 0.0001]$.

Although the index should theoretically run on the full dataset, the largest size I attempted to index was 1,000,000 data points. While not instantaneous, it was comfortably able to generate the index and perform range queries.

Parameter	**Value**
Max Keys per Page (Ω)	100
Est. Number of Keys per Shard (Ψ)	100
Number of Breakpoints (σ)	100
Number of Grid Cell Partitions (each dimension)	10
Number of Mapped Value Partitions	50

Table 1: Table of parameters used for experiments.

3.3 Evaluation Metrics

For the range queries, we will be evaluating it based on its **IO** cost (the average number of pages to be loaded for a range or KNN query) and its **size** (the disk storage space a method consumes). We generated 1,000 query rectangles, where each side length is a random value in $(0, \frac{1}{4}\Delta_i)$, where Δ_i) is the distance between the min and max values on the i-th axis.

Unfortunately, due to time constraints a benchmark algorithm (for example, an R-tree) was not implemented. Although the index is fully able to handle range queries, there is nothing to compare it to and so we are unable to evaluate our toy implementation's performance metrics.

4 Conclusion and Future Work

In this paper we implemented a novel learned index structure for spatial data (LISA) based on Li *et al.*'s paper. The construction of the index consisted of four parts:

1. Partitioning of data into grid cells

2. A partially monotonic mapping function $\mathcal{M}$ that maps multidimensional spatial data into $\mathbb{R}$

3. Learning a monotonic shard prediction function $\mathcal{SP}$ that assigns a shard id to each mapped value, thereby partitioning the mapped space into shards

4. Construction of local models $\mathcal{L}$ which are assigned to manage each shard. These work in conjunction with a variety of query processes (range queries, KNN queries, insertion and deletion) by handling the management of disk pages with respect to their assigned shard.

Since this was a toy implementation, it would be useful to expand it to the full scope of the paper: full datasets, multiple datasets, sufficient processing power, benchmark algorithms, and its complete scope of query processes (KNN queries, insertions and deletions). Besides these low hanging fruits, it would also be interesting to investigate whether LISA is able to handle other common query types (for example, spatial joins). Additionally, future research should also include evaluating each of these components of LISA and seeing where it can be optimized or replaced (with respect to the evaluation metrics of IO and size). Lastly, since this paper only came out a few months ago it would be useful to evaluate how adoptable LISA is within industry settings.

References

[1] Beckmann, N., Kriegel, H.-P., Schneider, R., and Seeger, B. The r*-tree: an efficient and robust access method for points and rectangles. In *Proceedings of the 1990 ACM SIGMOD international conference on Management of data* (1990), pp. 322–331.

[2] Bentley, J. L. Multidimensional binary search trees used for associative searching. *Communications of the ACM 18*, 9 (1975), 509–517.

[3] Finkel, R. A., and Bentley, J. L. Quad trees a data structure for retrieval on composite keys. *Acta informatica 4*, 1 (1974), 1–9.

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

[5] Kamel, I., and Faloutsos, C. Hilbert r-tree: An improved r-tree using fractals. Tech. rep., 1993.

[6] Kraska, T., Alizadeh, M., Beutel, A., Chi, H., Kristo, A., Leclerc, G., Madden, S., Mao, H., and Nathan, V. Sagedb: A learned database system. In *CIDR* (2019).

[7] Kraska, T., Beutel, A., Chi, E. H., Dean, J., and Polyzotis, N. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data* (2018), pp. 489–504.

[8] Li, P., Lu, H., Zheng, Q., Yang, L., and Pan, G. Lisa: A learned index structure for spatial data. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (2020), pp. 2119–2133.

[9] Manolopoulos, Y., Nanopoulos, A., Papadopoulos, A., and Theodoridis, Y. Series in advanced information and knowledge processing, 2005, 2005.

[10] Meagher, D. Geometric modeling using octree encoding. *Computer graphics and image processing 19*, 2 (1982), 129–147.

[11] Microsoft. Microsoft sql server. https://www.microsoft.com/en-us/sql-server, Accessed December 2020.

[12] MongoDB. Mongodb. https://www.mongodb.com, 2020.

[13] Ramakrishnan, R., and Gehrke, J. *Database management systems*. McGraw-Hill, 2000.

[14] Ramsak, F., Markl, V., Fenk, R., Zirkel, M., Elhardt, K., and Bayer, R. Integrating the ub-tree into a database system kernel. In *VLDB* (2000), vol. 2000, pp. 263–272.

[15] Royden, H. L., and Fitzpatrick, P. *Real Analysis*. Macmillan, 1988.

[16] Sellis, T., Roussopoulos, N., and Faloutsos, C. The r+-tree: A dynamic index for multi-dimensional objects. Tech. rep., 1987.

[17] Stewart, G. W. *Introduction to matrix computations*. Elsevier, 1973.

[18] Wang, H., Fu, X., Xu, J., and Lu, H. Learned index for spatial queries. In *2019 20th IEEE International Conference on Mobile Data Management (MDM)* (2019), IEEE, pp. 569–574.