HELDA
HELSINGIN YLIOPISTO
HELSINGFORS UNIVERSITET
UNIVERSITY OF HELSINKI

UNIVERSITY OF HELSINKI

https://helda.helsinki.fi

Towards an Instance-Optimal Z-Index

Pai, Sachith Gopalakrishna

2022-09-05

Pai , S G , Mathioudakis , M & Wang , Y 2022 , ' Towards an Instance-Optimal Z-Index ' , Paper presented at 4th International Workshop on Applied AI for Database Systems and Applications (AIDB@VLDB2022) , Sydney , Australia , 05/09/2022 . <
https://drive.google.com/file/d/1KvvEOzIZ7vt78eyB8k5XdQT5XF2nFngM/view?usp=sharing >

http://hdl.handle.net/10138/351385

cc_by
acceptedVersion

Downloaded from Helda, University of Helsinki institutional repository.

This is an electronic reprint of the original article.

This reprint may differ from the original in pagination and typographic detail.

Please cite the original version.

Towards an Instance-Optimal Z-Index [Extended Abstract]

Sachith Pai
University of Helsinki
Helsinki, Finland
sachith.pai@helsinki.fi

Michael Mathioudakis
University of Helsinki
Helsinki, Finland
michael.mathioudakis@helsinki.fi

Yanhao Wang
East China Normal University
Shanghai, China
yhwang@dase.ecnu.edu.cn

ABSTRACT

We present preliminary results on instance-optimal variants of the *Z-index*, a well-known spatial index that makes use of the Z-order curve. Unlike the base Z-index, the variants we propose aim to adapt to the data and range-query workloads of the given setting. Specifically, we provide an optimal algorithm that builds a Z-index that minimizes the expected number of retrieved data points for the given data and query workload. Moreover, since the optimal algorithm requires supra-linear running time, we additionally propose efficient heuristic algorithms to use in its place. Our experiments evaluate the performance of the resultant Z-indexes.

Reference Format: Sachith Pai, Michael Mathioudakis, and Yanhao Wang. Towards an Instance-Optimal Z-Index [Extended Abstract]. AIDB2022

1 INTRODUCTION

We present preliminary results on instance-optimal variants of the *Z-index*, an intuitively simple spatial index structure with long history in data management [12]. Unlike the base Z-index, the variants we present adapt to the given data and range-query workload, aiming to minimize the number of data points that are redundantly retrieved by the index when answering the range queries.

The Z-index is based on *Z-order curve* (a.k.a. Z-curve), a space-filling curve that projects multi-dimensional data onto a single dimension (1D), with the sort order of the data points in 1D following a predetermined order in which data are visited by the curve in the original data space. An example of the Z-curve and the corresponding Z-index is shown in Figure 1a. As we can see in the example, the Z-curve visits data points according to a hierarchical partitioning of the data space into cells, along with a specific ordering of those cells. For example, at the top level, the space is partitioned into four cells, namely A, B, C, and D; then at the second level each of these cells is partitioned into four sub-cells, and so on, with the partitioning happening at the coordinates corresponding to the median of the data distribution along each axis. Within each cell, the ordering of its sub-cells always follows the same 'Z'-like pattern, i.e., cells are ordered as (ABCD) for the first level, with higher cells having higher importance for the ordering of points. In what follows, we'll be using the term 'data space' to refer to the original 2D space of the data and 'rank' to refer to the Z-order of a point, i.e., its 1D projection.

The Z-index can be directly used for point queries, as the Z-curve provides a mapping from a data point in the data space to its rank. Moreover, the Z-index has the desirable property to be monotonic, i.e., data points that are dominated by a data point P in the data space will have smaller rank than P. Taking advantage of monotonicity, the straightforward query-answering algorithm is to obtain the 1D ranks of the bottom-left and top-right points of the query rectangle, scan the data entries between the two, and filter the data points that satisfy the query. However, the Z-curve does not perfectly preserve locality, which is another desirable property. For example, consider the range query on the rectangle in Figure 1b. In this case, answering the query will lead to retrieving many false-positive data entries, leading to unnecessarily high query cost.

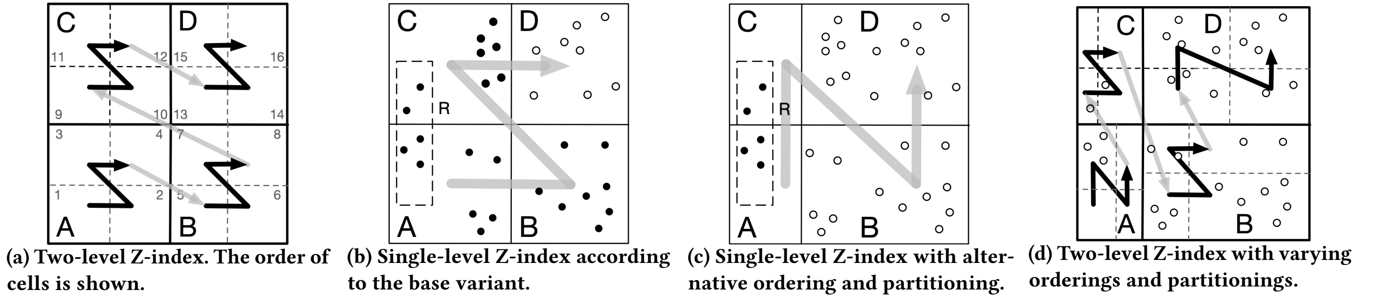
In this extended abstract, we present several variants of the Z-index that, unlike the base Z-index, adapt not only to the data, but also to the anticipated workload of range queries, and thus further mitigate the retrieval of redundant data entries. The variants we propose are flexible in two ways that the base Z-index isn't: the ordering and partitioning of the cells. First, we allow the ordering of the cells to have either the base ABCD pattern or the alternative ACBD, which is the only other ordering of the four sub-cells that preserves monotonicity. Second, rather than partitioning the cells at a fixed location (i.e., at the median), we allow it to vary. Intuitively, it pays off to partition the data space so that many range queries correspond to cells that contain many data points, because this allows the index to avoid redundant retrievals of points in other cells. For example, the Z-index with the alternative ordering and partitioning shown in Figure 1c would retrieve fewer redundant data entries compared to the base Z-index of Figure 1b. Therefore, if similar range queries dominate the anticipated query workload, alternative ordering and partitioning of cells could lead to improved querying performance. The Z-index variants we present allow different ordering and partitioning for each cell across the index hierarchy, as shown in Figure 1d.

Related Work. While the preliminary results we present herein concern only the Z-index and its variants, we include a brief discussion on the related literature, to better locate our work. Our work is related to spatial indexes (see [3] for a survey) and particularly *space-filling curves* [7]. But it is also related to a growing line of work that makes use of the statistical properties of the data and queries to build indexes that are optimized for the given setting. A seminal work in that line is that of Kraska et al. [5], which introduced the idea of *learned indexes*, which utilize machine learning models to enhance or replace traditional indexes. As a follow-up, many different learned indexes have appeared in the last three years. Closest to our work is the recent literature on learned spatial indexes [1, 2, 4, 6, 8, 10, 11, 13, 14].

2 PRELIMINARIES

In general, the Z-index can be used to index a d-dimensional Euclidean space for any $d \in \mathbb{N}$. However, to facilitate presentation, in what follows we restrict ourselves to $d = 2$ and consider a set

This paper is published under the Creative Commons Attribution 4.0 International (CC-BY 4.0) license. Authors reserve their rights to disseminate the work on their personal and corporate Web sites with the appropriate attribution, provided that you attribute the original work to the authors and AIDB 2022. 4th International Workshop on Applied AI for Database Systems and Applications (AIDB '22). September 5th, 2022, Sydney, Australia.



(a) Two-level Z-index. The order of cells is shown.
(b) Single-level Z-index according to the base variant.
(c) Single-level Z-index with alternative ordering and partitioning.
(d) Two-level Z-index with varying orderings and partitionings.

Figure 1: Examples of Z-curve and Z-index.

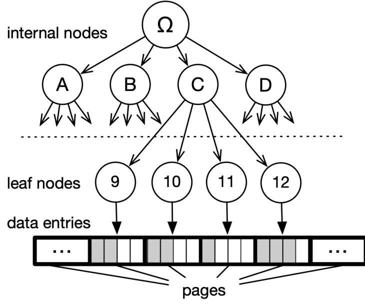


Figure 2: Illustration of the index structure.

$\mathcal{D}$ of N data points in $\mathbb{R}^2$. The Z-index for $\mathcal{D}$ is defined by two elements. The first is a *hierarchical partitioning* of the data space into cells, with each cell partitioned into four sub-cells, down to a predetermined granularity of leaf-cell size L. The hierarchical partitioning is determined by the split points which, for the base Z-index variant, occur at the median of the x- and y-coordinates of all data points. The second is the *ordering* of the cells, and hence of the data points: for the base Z-index variant, for each parent-cell, its four children-cells are ordered according to the ABCD pattern; and higher-level cells have higher ordering importance, as shown in Figure 1a. Notice that the ABCD order guarantees *monotonicity* for points that fall within different leaf-cells: if point a in leaf-cell X is dominated[1] by point b in leaf-cell $Y \neq X$, then a will appear earlier in the ordering than b.

A given partitioning and ordering define the Z-index, with a quaternary structure, as shown in Figure 2. Each *internal node* stores the coordinates of the split-point according to which the corresponding cell is partitioned into its children-cells, as well as the ordering of the children-cells. Each *leaf node*, on the other hand, contains a pointer to a page of size L within the data entries ARRAY. This creates a linked list structure at the leaf layer of the tree which is useful to process range queries. Note that the index is *clustered*, as data points corresponding to consecutive leaf nodes are stored in consecutive pages. Moreover, data points within a page are stored in random order. In all that follows, a cell will be split if and only if it contains more than L points, where L is the number of points that fit within a page (i.e., the retrieval unit for given storage).

Range Queries are defined with a rectangle and request the data points in $\mathcal{D}$ that overlap with the rectangle (see Figure 1). For a given rectangle R, we will write BL_R and TR_R to refer to its bottom-left and top-right vertices. For operations related to rectangles, we'll be using these two vertices, as they are sufficient to define the rectangle. To answer the query defined by a given rectangle R, the index extracts the defining vertices (BL, TR), makes point queries for BL and TR to retrieve the locations (LOW, HIGH) of the first and last page pointed to by the respective leaf nodes; scans the corresponding slice of the data entries ARRAY [LOW:HIGH], while filtering those data points that fall within the query rectangle; and finally returns the filtered points as the result set. For example, consider the one-level Z-index for the data shown Figure 1b and the range query R represented with a rectangle therein. Because the bottom-left and top-right vertices of R are in cells A and C, respectively, and because the order of the cells is ABCD, answering the range query retrieves the data points of cells A, B, and C before it filters and returns the result set.

2.1 Our Approach

We consider generalized variants of the Z-index, for which the partitioning and ordering are allowed to vary for each cell, as shown in the example of Figure 1d. In more detail, for a given dataset $\mathcal{D}$ and set of range queries Q, we are interested in building Z-index with a partitioning and ordering that minimizes the corresponding *retrieval cost*, i.e., the number of data points that are accessed at query time. Essentially, the retrieval cost for a range query is the number of the data entries that are scanned when Z-index is used to answer the query.

Retrieval Cost. We write q_{XY} to denote the number of queries that have their bottom-left vertex in X and top-right vertex in Y. For a set of queries Q and a Z-index with root cell Ω, the total cost of all queries is aggregated to form the full objective function Obj. Specifically, if the ordering of Ω is ABCD, the objective is

$$\begin{aligned}\text{Obj}_\Omega(Q|x,y;\text{ABCD}) &= \sum_{\text{R}\in Q} cost_\Omega(\text{R}|x,y;\text{ABCD}) \\ &= q_{\text{AD}}(n_\text{B}+n_\text{C}) + q_{\text{AC}}n_\text{B} + q_{\text{BD}}n_\text{C} \\ &\quad + \text{Obj}_\text{A}(Q) + \text{Obj}_\text{B}(Q) + \text{Obj}_\text{C}(Q) + \text{Obj}_\text{D}(Q)\end{aligned} \tag{1}$$

A similar formula holds for the ACBD order. If a cell X is no further split, then the recursion terminates with

$$\text{Obj}_X(Q) = q_{XX}n_X$$

Moreover, especially for a **single-level** Z-index, i.e., the index consisting only of the root Ω and four cells (A, B, C, D), we'll be using $\mathcal{H}$ to denote the objective function.

[1] Point a is dominated by point b if both coordinates of a are smaller than those of b.

Problem Formulation. The problem we address is to identify a configuration (i.e., a partitioning and ordering of cells) that leads to minimum retrieval cost.

PROBLEM 1 (OPTIMAL Z-INDEX). *Given a set of data points $\mathcal{D}$ and a query workload Q, identify the partitioning and z-ordering that define a Z-index with minimum* OBJ.

In practice, Q will be determined based on historical logs of range queries, or a set of 'representative' or 'important' range queries for the application at hand, or in general a set of anticipated range queries for which the index should be optimized.

3 ALGORITHMS

Let N be the number of data points, $N = |\mathcal{D}|$, and let K be the number of Z-index levels. A *brute-force* approach to solve Problem 1 would be to enumerate all configurations (partitionings and orderings) within a K-way for-loop, with one for-loop per index level enumerating the $O(N^2)$ configurations that are possible given the running configuration for the above levels. Implemented naively, this brute-force approach would compute the objective value $O(N^{2K})$ times before it returned the optimal one. Obviously, its running time would be prohibitive. Nevertheless, two observations allow us to improve upon the running time of the brute-force approach. First, notice that for N data points, there are $O(N^4)$ distinct ranges (rectangles) of interest, and, for the appropriate partitioning of the data space, each of them would be enumerated as a cell at least once by the brute-force approach. Second, Problem 1 has an optimal substructure property, such that finding an optimal configuration for a cell optimally consists in finding the optimal configuration of children-cells. The two observations inspire a dynamic-programming algorithm that solves Problem 1 once for each of the distinct possible ranges/cells, memorizes the individual solutions, and combines them to solve the problem optimally for the entire data space, in $\Theta(KN^4)$ running time. While significantly improved compared to the brute-force approach, such a running time is still prohibitive for large datasets.

3.1 Greedy Heuristics

We seek to further improve the running time with two approximate but efficient heuristics as described below. The heuristics share the following two common features. First, they proceed greedily, determining the partitioning and ordering of the cells within the same level, one level at a time, from top (root) to bottom (leafs). Every time a cell is split and its children ordered, the single-level objective $\mathcal{H}$ is used. Algorithm 1 shows the template of the greedy algorithm followed by such heuristic. Second, the heuristics are based on approximating the exact data distribution $\mathcal{D}$ and range-query distribution Q with approximate 'model' distributions (hence "learned" approaches), which are then used to efficiently approximate various computations. Next, we describe the basic idea for each heuristic and leave the detailed description to the full paper.

The Independence-based Heuristic (INDGREEDY) assumes that the data and queries are the product of independent distributions along the x and y axes. Specifically, the assumption is that the x and y coordinates of the data points, as well as of the BL and TR vertices of the queries, are generated independently for each

```
1 Function Greedy(X, D, Q; SolveCell; Z):
    Input :Node X in Z-index, Query workload Q, Data D;
           Heuristic SolveCell; Z-index Z
2   if n_X < L then
3     return // Do not split when X can fit in page
4   // Split X and order its children-cells
5   Cells A,B,C,D; Ordering o := SolveCell (X,D,Q)
6   Add cells A,B,C,D and ordering o to Z
7   // Apply Greedy to children-cells
8   foreach Cell Y: A, B, C, D do
9     Greedy (X, D, Q; SolveCell; Z)
```

Algorithm 1: Greedy template for heuristic algorithms

axis. This assumption leads to a simplified formula that approximates the single-level objective $\mathcal{H}$. This heuristic, then, chooses the split-point (x, y) for the given cell so as to optimize the simplified approximation of the single-level objective $\mathcal{H}$ and, for the chosen split point, it subsequently chooses the order of the resulting children-cells that optimizes the single-level objective $\mathcal{H}$.

The Sampling-based Heuristic (UNIGREEDY) first samples candidate split points uniformly at random for the given cell, and chooses the split point and ordering that optimize the single-level objective $\mathcal{H}$. Note that, to efficiently evaluate $\mathcal{H}$, UNIGREEDY uses a learned density model that allows it to estimate the number of data points that fall within each of the children-cells resulting from the partitioning, as necessary to compute the objective. For the density model, we use a KD-tree that is fit to the data at an appropriate granularity to obtain good and fast estimates for the number of data points that fall within a given rectangle.

4 EXPERIMENTS

Data. We show results on two synthetic and one real-world datasets. The synthetic datasets contain 2D data points distributed uniformly, and contain 50K and 80K points, respectively. The real-world dataset (HELPOIs) contains 2D data points corresponding to the coordinates of points-of-interests in the city of Helsinki [9].

Queries. We generate several synthetic workloads. Each query workload consists of a collection of *clustered* query rectangles, and we experiment with varying number of clusters, variance of each cluster, and selectivity. The center of a rectangle is generated by a mixture of k Gaussian's with specified variance, with centers randomly sampled from the set of data points. We obtain a query rectangle from the generated center by growing along all four directions such that the query covers a portion of data space equivalent to the required selectivity. Each instance of a query workload consists of M queries (rectangles), generated randomly with the above procedure. We set the value of M to 5% of the N for our experiments. For each dataset, we generate 10 instances of query workloads and the values we report are averages over those instances. We set a page size L of 256 for our experiments.

Experimental Results. The results are shown in Figure 3, where each plot is the heat maps for one dataset. The x-axis corresponds to the selectivity of the query workload and the y-axis to the variance of the query clusters. Each cell in the heat map is associated with one value, which expresses *how many times* the speedup s of the respective Z-index variant (INDGREEDY on the left column,

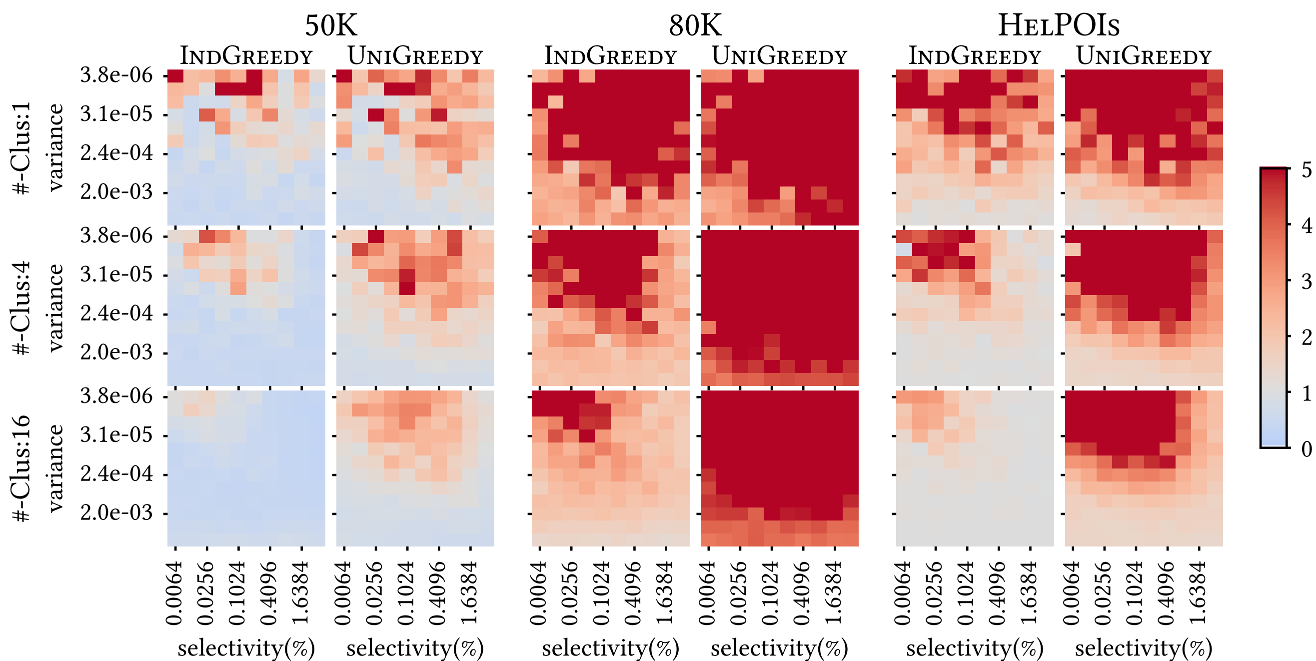


Figure 3: Experimental results.

UniGreedy on the right) over the base Z-index in terms of point retrievals, i.e., $s = \textsc{Obj}(base)/\textsc{Obj}(variant)$. The figure contains three rows of plots corresponding to different number of query clusters in the query-generation procedure.

We make the following observations. First, across all datasets, the benefits of IndGreedy and UniGreedy are more pronounced in some areas of the selectivity×variance space. Intuitively, the instance-optimal variants perform better when queries have small selectivity and small cluster variance. We confirm that such a combination corresponds to having multiple queries all of which are selective and concentrated enough to fall within typically one or otherwise a small number of index leaf cells. In such cases, IndGreedy and UniGreedy have an advantage over the base Z-index because of their flexibility to adapt the alignment of the cells to the location of the queries. Second, in line with the first observation, the benefits of IndGreedy and UniGreedy are larger when queries are generated from a smaller number of clusters. A larger number of clusters leads to query workloads that resemble more and more the uniform distribution of queries, where there is no 'structure' for the variants to exploit. Third, the variants are sometimes inferior to the base Z-index. This is because the variants work with a modified and more easily computable objective function. For those regions that are not favorable to the variants – i.e., in cases of large selectivity or variance, where queries end up spanning multiple index leaves – the variants may end up choosing worse partitions due to the modified objective. This deficiency could be addressed by always making a final check with the full objective against the base Z-index, and using the base variant instead of IndGreedy or UniGreedy if the latter lead to worse objective value. Fourth, UniGreedy typically outperforms IndGreedy, though at a cost of longer build time. This trade-off is as expected as the independence assumption on which IndGreedy is based on, approximates the complex patterns present in data crudely. The longer build time of UniGreedy follows as we build a more accurate density model.

REFERENCES

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

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

[3] Volker Gaede and Oliver Günther. 1998. Multidimensional Access Methods. *ACM Comput. Surv.* 30, 2 (1998), 170–231.

[4] Tu Gu, Kaiyu Feng, Gao Cong, Cheng Long, Zheng Wang, and Sheng Wang. 2021. The RLR-Tree: A Reinforcement Learning Based R-Tree for Spatial Data. arXiv:2103.04541 [cs.DB]

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

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

[7] Mohamed F. Mokbel and Walid G. Aref. 2018. Space-Filling Curves. In *Encyclopedia of Database Systems (2nd ed.)*. Springer.

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

[9] OpenStreetMap contributors. 2017. Planet dump retrieved from https://planet.osm.org . https://www.openstreetmap.org.

[10] Varun Pandey, Alexander van Renen, Andreas Kipf, Jialin Ding, Ibrahim Sabek, and Alfons Kemper. 2020. The Case for Learned Spatial Indexes. arXiv:2008.10349 [cs.DB]

[11] Jianzhong Qi, Guanli Liu, Christian S. Jensen, and Lars Kulik. 2020. Effectively Learning Spatial Indices. *Proc. VLDB Endow.* 13, 11 (2020), 2341–2354.

[12] Raghu Ramakrishnan and Johannes Gehrke. 2003. *Database management systems* (3 ed.). McGraw-Hill New York.

[13] Haixin Wang, Xiaoyi Fu, Jianliang Xu, and Hua Lu. 2019. Learned Index for Spatial Queries. In *MDM*. 569–574.

[14] Songnian Zhang, Suprio Ray, Rongxing Lu, and Yandong Zheng. 2021. SPRIG: A Learned Spatial Index for Range and kNN Queries. In *SSTD*. 96–105.