Spatial Independent Range Sampling

Dong Xie*[2], Jeff M. Phillips[1], Michael Matheny*[3], Feifei Li*[4]
[1]University of Utah, [2]The Pennsylvania State University, [3]Amazon, [4]Alibaba Group
dongx@psu.edu, mmatheny@amazon.com, jeffp@cs.utah.edu, lifeifei@alibaba-inc.com

ABSTRACT

Thanks to the wide adoption of GPS-equipped devices, the volume of collected spatial data is exploding. To achieve interactive exploration and analysis over big spatial data, people are willing to trade off accuracy for performance through approximation. As a foundation in many approximate algorithms, data sampling now requires more flexibility and better performance. In this paper, we study the spatial independent range sampling (SIRS) problem aiming at retrieving random samples with independence over points residing in a query region. Specifically, we have designed concise index structures with careful data layout based on various space decomposition strategies. Moreover, we propose novel algorithms for both uniform and weighted SIRS queries with low theoretical cost and complexity as well as excellent practical performance. Last but not least, we demonstrate how to support data updates and trade-offs between different sampling methods in practice. According to comprehensive evaluations conducted on real-world datasets, our methods achieve orders of magnitude performance improvement against baselines derived by existing works.

CCS CONCEPTS

• **Theory of computation** → **Data structures design and analysis**; • **Information systems** → *Multidimensional range search.*

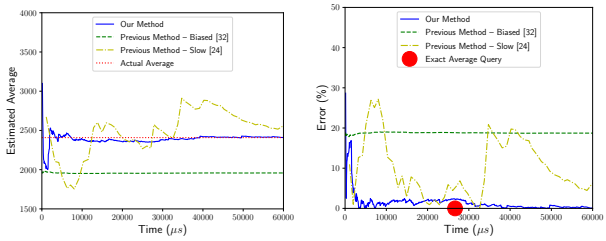
KEYWORDS

SIRS; range sampling; spatial data sampling

ACM Reference Format:
Dong Xie*[2], Jeff M. Phillips[1], Michael Matheny*[3], Feifei Li*[4]. 2021. Spatial Independent Range Sampling. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3448016.3452806

1 INTRODUCTION

The wide presence of smart phones and various sensing devices has created an explosion of spatial data. In particular, such data are critical to location-based services (LBSs) like Uber or Google Maps, and also play important roles in other applications such as site recommendations, traffic optimization, and other IoT applications.



(a) Approximation on AVG (b) Error-Performance Trade-off

Figure 1: Online Aggregation with Sampling.

How to query and analyze such large spatial data in high performance stands as a fundamental challenge. Fortunately, users often do not need exact results for their queries. Instead, they are happy with approximation, especially if it comes with quality guarantees. This opens up the possibility to trade off between query time and accuracy on the fly, hence further enabling *interactive exploration and analysis* [6].

Given the popularity of online map services, spatial data are particularly suited for interactive exploration. For example, a manager in finance may want to get sales aggregation across different regions and time periods with zoom in/out capability on the map so as to understand the data in different granularity. In such case, the user may only need an approximation before moving to other regions. In order to achieve such interactive analysis experience on big spatial data, retrieving *independent random samples* efficiently from arbitrary query regions serves as a fundamental operation.

Such operations not only benefit approximation algorithms as described above, but also facilitate data visualization on a map when there are too many points covered by a requested region [32]. Moreover, spatial statistics analysis (e.g., spatial scan statistics [18], Moran's I [19], or kriging [21]) is also an important application where many spatial regions are checked for anomalies, many pairs of ranges are checked for association, and complex models are built in each spatial zone. In each case, millions of range queries need to be issued either fully accurately or independently sampled so as to ensure accuracy and prevent false correlations. Another prominent application is machine learning, which implicitly assumes independence of samples for use in model bounds, cross-validation, or stochastic gradient descent. Non-independence will break each of the formulations and can lead to incorrect models or conclusions. For instance, the prevalent empirical risk minimization [29] essentially forms a model using a (carefully chosen weighted) average of iid sampled data instances. Thus, to enable interactive and/or spatio-temporally constrained machine learning, *independent* samples are necessary to make it scalable and reliable. When assigning points with weights, weighted random sampling is also paramount for real-time site recommendations, reducing variance in estimates, and spatially-refined machine learning.

Despite its importance, samples are usually given as input in applications rather than generated during the algorithm procedure.

* Work performed partially when authors are affiliated with University of Utah

SIGMOD '21, June 20–25, 2021, Virtual Event, China

ACM ISBN 978-1-4503-8343-1/21/06…$15.00
https://doi.org/10.1145/3448016.3452806

Hence, existing systems typically rely on sampling directly over a full query (which can be very slow), or querying over pre-built samples (which can harbor bias and may be too small). Take online aggregation [15] as an example, where we estimate an aggregated value (e.g., sum, average, etc.) over attributes of points covered by a user query. Previous work [31] does not guarantee reported samples across multiple queries are independent to each other, which can cause serious bias in our aggregation example. Fig. 1 shows the estimated average weight over time of points in a query range produced by our approach, the current state of the art [31] (some pre-built samples and exhibits bias), and the only other existing method [23] that produces independent samples without bias. Our method converges efficiently to the actual average, whereas previous work [31] converges to a significantly biased value (20% off) and the other method [23] converges much more slower to the actual average. There are other challenges in using pre-built [6] or partially pre-built samples [31]. When conducting distribution analysis (like kernel density estimation [27]) over two overlapping regions, samples will always be dependent, thus introduces bias. Further on progressively-improved queries, improvement either stops with unfixable bias due to sample dependence, or cached samples need to be rebuilt, reverting to intractable sampling from a full query.

Independent range sampling (IRS), which formalizes the problem of retrieving independent random samples from a query range, is both an old and a new problem. In the late 1980s, Olken [23] devised a variety of techniques leveraging hierarchies to solve the problem. However, as each sample used the hierarchy independently, it is too slow for modern data set sizes. Using prebuilt samples (c.f., BlinkDB [6]) allowed systems to scale to modern sizes by relaxing independence across samples. Since then theoretical works [4, 5, 16] showed that true independent samples could be generated without traversing a hierarchy for each sample (as did Olken), but were very intricate and only for limited cases in one-dimension or halfspaces in low-dimensions. Inspired by the earlier of these theoretical approaches [16] and Olken's original ideas [23], state-of-the-art practical solutions [31] work for minimum bounding rectangles (MBRs), but still does not provide independence across queries or in progressive queries (as observed in Fig. 1).

In this paper, we study how to support spatial independent random sampling (SIRS) queries over MBRs for low-dimensional data efficiently. Specifically, given an axis-aligned rectangle R and an integer k, a SIRS query over data set P will return k independent samples from points in $R \cap P$. For each of the k samples, the probability for each point in $R \cap P$ to be sampled is equal in uniform SIRS, or proportional to its relative weight among points in the query region for weighted SIRS. We choose MBRs as our query range type since they are the most commonly-used regions (longitude and latitude aligned boxes) in geo-spatial applications. Note that the independence is *required not only for samples returned from the same query, but also hold for samples returned from different queries.*

The central idea is, for each query, to efficiently build a temporary data structure, so that all requested samples can be drawn over the query range completely independent to each other. Specifically, we reduce the general case of SIRS queries to a series of one-dimensional problems where we develop very practical algorithms with guarantees. Several technical challenges are identified and addressed to make our solution rigorously understood as well as practical in terms of space and query efficiency. As a result, we are able to support both uniform and weighted IRS queries with low space cost and query latency, achieving orders of magnitude of empirical performance gain comparing to the state of art.

The key contributions of our work are:

- Our indexes have linear storage cost, guarantee independence of each sample, and perform uniform/weighted SIRS queries orders of magnitude faster than any previous work. The performance of our sampling methods do not depend on the number of data points covered by the query region or its height/width ratio.
- Our sampling indexes are extended to support data updates.
- We carefully study the bottlenecks and trade-offs of different sampling methods.
- We conduct comprehensive evaluations to show the cost of our index structures and sampling methods comparing with baselines derived by existing works.

The remainder of this paper is structured as follows: In Sec. 2, we will formally define SIRS problem and provide essential backgrounds of this paper. Next, we propose a general framework with two instantiations to solve uniform SIRS problem in Sec. 3. Sec. 4 will extend our uniform method to support weighted SIRS queries. Then, we will discuss the bottlenecks of different methods, trade-offs between them and a potential hybrid method in Sec. 5. We will describe how to extend our method to support updates in Sec. 5. To verify the efficiency of our methods, we conduct comprehensive empirical evaluation on real world data sets in Sec. 6. Finally, we provide more detailed connection to related works in Sec. 7 and conclude the paper in Sec. 8.

2 BACKGROUND

In this section, we formalize the uniform and weighted version of spatial independent range sampling (SIRS) problem. Then, we will cover the essential building blocks of our proposed methods. Finally, we will highlight the importance and applications of SIRS queries with several existing baseline solutions.

Problem Formulation. Over a data set of points $P \in \mathbb{R}^d$, the goal of SIRS query is to retrieve independent samples from points of P residing in a query region R. Generally, the query region can be derived from any geometric shape. In this paper, we focus on axis-aligned minimum bounding rectangles (MBRs) for their common presence in real world applications. Formally, we define the uniform SIRS problem as below:

Definition 1 (Uniform SIRS). *Given a data set $P \subset \mathbb{R}^d$, a query MBR R and an integer k, a uniform SIRS query will return k independent random samples from $R \cap P$ with each data point $p \in R \cap P$ having a probability of $\frac{1}{|R \cap P|}$ to represent each of the k samples.*

In more general cases, there could be a function $w : P \to \mathbb{R}$ assigning each point $p \in P$ a weight $w(p)$. Based on such function, we can define weighted SIRS problem as:

Definition 2 (Weighted SIRS). *Given a data set $P \subset \mathbb{R}^d$ with a weight function $w : P \to \mathbb{R}^+$, a query MBR R and an integer k, a weighted SIRS query will return k independent random samples from $R \cap P$ with each data point $p \in R \cap P$ having a probability of $\frac{w(p)}{\sum_{q \in R \cap P} w(q)}$ to represent each of the k samples.*

Without the loss of generality, we fix $d = 2$ in this paper to demonstrate our index structure and sampling algorithm. We expect our method will work with any low dimension cases where $d < 10$; see Sec 3.4 and evaluation up to $d = 7$ in Sec. 6.

Spatial Indexes and Space Filling Curves. Many have studied spatial index structures such as grid file [20], quad-tree [13], KD-Tree [7], R-Tree [14], R+-Tree [28], etc. The central idea of these index structures is to partition the underlying data points based on spatial locality that one could leverage in pruning irrelevant data during the query process. There is another popular approach which maps multi-dimensional data to a single sorted order by via space filling curves like z-order curve and Hilbert curve. Such space filling curves recursively divide a 2- (or higher-) dimensional square into cells, and order those cells, inducing a partial ordering on all points contained in them. This recursion turns the partial order into a single total order, which maintains locality through the recursive cell decomposition. As a result, we can utilize well implemented one-dimensional index structures like B+-Tree in traditional databases to index spatial data. Existing solutions adopting this approach includes linear quad-tree and Hilbert R-tree.

Walker's Alias Method. Given a set of items with weights assigned $\{w_1, w_2, \cdots, w_n\}$, the goal of weighted sampling problem is to build a data structure such that a query extracts an index i with probability $p_i = w_i / \sum_{j=1}^{n} w_j$. The indices returned by different queries should be independent. The classic solution of this problem is Walker's alias method [30] that has $O(1)$ query cost and $O(n)$ preprocessing time. It redistributes the weight of items into n cells, where each cell is partitioned by a pre-calculated weight into at most two items. The weight assigned to each item in the input is maintained as the sum of its weight across all cells. A query selects one cell uniformly at random, then chooses one of the two items in the cell by weight; hence selects items proportional to their weights in $O(1)$ time. The structure is built to support weighted sampling in this method is usually referred as an alias structure.

Baselines. Olken et al. proposed the classic method [23] of sampling on top of a B+-Tree. The main idea is to traverse the tree down a random path in the tree while applying rejection when it is not possible to get a valid sample residing in the query range. Intuitively, this method can be easily generalized into any tree-based index structure, hence we implement it on top of KD-trees as one of our baselines. It is an effective method when the query range is not very selective and the number of requested samples is small. However, it is slow in the general cases due to involving potentially too many random number generations and rejections.

Wang et at. [31] proposed a uniform sampling method on spatial data over MBRs. It leverages pre-built sample buffers on R-tree nodes and maintaining a frontier of tree nodes for each query to accelerate Olken's method. However, it does not guarantee independence across queries, because the sample buffers are fixed. More specifically, reported samples in the sampling buffer is guaranteed to be returned again in the further queries over the same region, thus breaks the independence rule.

Finally, we implement a brute force baseline which retrieves all points in the query range and then samples from that set.

Baseline enhancements. We found a very simple optimization for Olken's method that can *accelerate it up to 12*x, which we call the LCA optimization. Specifically, we find the deepest node that can cover all range query result candidates. In other word, it is the least common ancestor (LCA) of all leaf nodes which intersect the query region. With such a node found, we can start the random traverse from the LCA rather than the tree root. This optimization will reduce rejection rate, and hence the total random number generations. In Sec. 6.3, we will show the effectiveness of this optimization on both uniform and weighted SIRS problem.

To make a fair comparison against our method, we adjust Wang's method [31] by adding a valid sample offset on all tree nodes (so that we can mark samples as invalid after they have been reported) and rebuilding a sample buffer when it is running out of valid samples in order to enforce independence. This is only feasible to implement for uniform SIRS queries but not for weighted SIRS. We implement this method over R-Trees and KD-trees as baselines.

3 UNIFORM SIRS

Our general framework consists of an index structure design and sampling algorithm to support uniform SIRS queries. We will provide details along with essential theoretical analysis.

3.1 Sampling Framework

We start from a simple observation: uniform IRS for intervals over static one-dimensional data sets is simple to solve. On a data set $D = \{x_1, x_2, \cdots, x_m\}$ where $x_i \in \mathbb{R}$, a uniform IRS over interval $[a, b]$ and integer k will return k independent uniform random samples from $\{x \mid x \in [a, b], x \in D\}$. After sorting D to get $(\overline{x_1}, \overline{x_2}, \cdots, \overline{x_m})$, for each query, we do a binary search for a and b to get the index range $[s, t]$ such that $\{x \mid x \in [a, b], x \in D\} = \{\overline{x_s}, \cdots, \overline{x_t}\}$. To get a random sample in range $[a, b]$, we simply generate a random number between $[s, t]$ and return the corresponding element in the sorted sequence. Hence, one-dimensional uniform IRS for intervals can be solved with $O(n)$ space cost, $O(n \log n)$ preprocessing cost, and $O(\log n + k)$ query cost.

Based on this observation, our central focus for the SIRS problem is to **separate the query structure (built in sublinear time) from drawing samples (retrieved in $O(1)$ time each)**. We do this by reducing SIRS queries to a *set* of one-dimensional IRS queries, connected with an alias structure.

First, we layout the data in a sequence $(\overline{p_1}, \overline{p_2}, \cdots, \overline{p_n})$ based on some space decomposition method so spatially close data points are also close to each other in the linear sequences. Then, for arbitrary query MBR, we can decompose it into a set of *continuous* index ranges $\{[s_1, t_1], [s_2, t_2], \cdots, [s_m, t_m]\}$ whose corresponding data cover all possible sample candidates.

Second, we then build an alias structure over these index ranges with $[s_i, t_i]$ having weight $|t_i - s_i + 1|$. Recall that the cost of building the alias structure is $O(n)$ on n weighted items. As a result, it only takes $O(m)$ time for each query to construct its top-level alias structure, which turns out not to be a dominating cost.

Finally, we can retrieve an independent uniform random sample of the query range by leveraging Walker's alias method to get a random index range $[s', t']$ first, and then report a random data point from $\{\overline{p_{s'}}, \cdots, \overline{p_{t'}}\}$.

Lemma 1. *If we can map a query MBR R to $I(n, R)$ continuous index ranges in $M(n, R)$ time, our sampling framework retrieves k independent uniform samples from R in $O(M(n, R) + I(n, R) + k)$ time.*

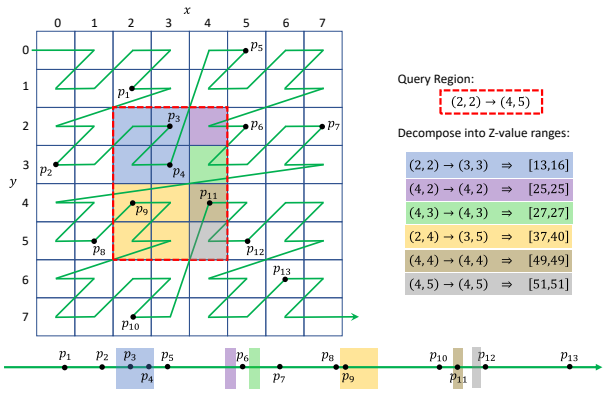


Figure 2: Z-order Curve Based Space Decomposition

Proof. The runtime follows from the above discussion since it takes $O(M(n,R)+I(n,R))$ time to build the top-level alias structure, and $O(1)$ time to generate each query. The independence of each sample also follows directly since each is generated with independent randomness (in two steps: to select a index range through the alias structure, and then to select a point in that index range).

What remains is to argue each point is selected with the correct probability. Let $n_i = t_i - s_i + 1$ be the number of points in interval $[s_i, t_i]$. And let $n_R = \sum_{i=1}^{m} n_i$ be the total number of points in query range R. A point is selected when two events are true: (1) its interval $[s_i, t_i]$ is selected (with probability n_i/n_R), and (2) it is selected from that interval (with probability $1/n_i$). Since these steps are done independently, the overall probability for any point is $\frac{n_i}{n_R} \cdot \frac{1}{n_i} = \frac{1}{n_R}$, as desired. □

There are many ways to map multi-dimensional data to one dimension while preserving spatial locality. In the following subsections, we will show two instances of the sampling framework based on two different space decomposition and indexing strategies.

3.2 Z-Value Sampling Method

Space filling curves naturally fit in our sampling framework – they are commonly adopted to map spatial data to one dimension so that traditional RDBMS indexes like B+-tree can be reused. Z-order curves form a natural hierarchical quad decomposition of underlying space. We use it since it is cheap to calculate and parameter free, but others like Hilbert curves could be used.

To construct z-order curve representation, we first need to discretized the space into unit cells so that each point can assign to a cell represented by integer coordinates. Denoting the coordinate of a cell as (x, y), the z-value of such cell would be interleaving the binary representation of x and y. Fig. 2 show an example of the z-order curve based space decomposition. In particular the z-value of p_3 residing in cell $(3, 2)$ is $(001110)_2 = 14$ by interleaving $(011)_2$ and $(010)_2$. It provides a natural quad decomposition, essentially a quad-tree, and all points lying in a quad-tree cell have consecutive z-values. As a result, each query MBR can be decomposed into a set of continuous z-value ranges derived from its composing quadrants.

To make Z-order SIRS concrete, we first sort all data points based on their z-values and build a quad-tree on top of it. While building the index, we mark the index ranges each quad-tree node covers on the sorted data. As a query MBR is issued, we follow the quad-tree to find the minimum set of disjoint quadrants to cover it. Note that when we found a quadrant fully covered by the query MBR, we can stop traversing its children. Since each quad-tree node has a corresponding index range on the sorted data, all points covered by the query range will be narrowed down to a set of continuous ranges. Finally, we construct an alias structure on top of these ranges so that we can start retrieving samples.

Fig 2 provides a running example of the z-value sampling method. As we can see, all data points are sorted based on their z-value laying out on the axis below. Consider an IRS query on MBR from $(2, 2)$ to $(4, 5)$. According to the z-order curve decomposition, we can break it into 6 continuous z-value ranges. Note that we do not need to further break the quadrants $(2, 2) \rightarrow (3, 3)$ and $(2, 4) \rightarrow (3, 5)$ as they are already fully covered by the query. Then, we check to see how many points are covered by each range and build an alias structure on top of it. In this case, we have an alias structure built on top of distribution $\{2, 1, 1\}$ mapping to continuous ranges on data $p_3 - p_4$, p_9 and p_{11}. Now, for each requested sample, we leverage the alias structure to choose a random range. For instance, we end up with range $p_3 - p_4$. Then, we generate a random integer from $[3, 4]$ to decide either p_3 or p_4 will be returned as a sample.

Let the total number of continuous intervals by decomposing query MBR R is $O(c(R))$ where $c(R)$ represents the number of data points covered by R. Similarly, the time cost for doing such decomposition is also $O(c(R))$. By applying it to Lemma 1, we ended up with a full solution with $O(n)$ space cost and the query cost bounded by the following corollary:

Corollary 1. *For any query range R, the z-value sampling method can retrieve k independent uniform random samples from a query MBR R in $O(c(R) + k)$ expected time.*

Note that $c(R)$ can be $O(n)$ in the worst case. Thus, query cost of this method can be as bad as $O(n+k)$. However, in most realistic cases, $c(R)$ is reasonable, making the z-value sampling method a viable solution. Also note that the query time $O(c(R)+k)$ is expected due to rejection sampling at leaf nodes. We can also achieve worst case $O(c(R) + k)$ query time by scanning all points residing in boundary leaf nodes which is not fully covered by the query range.

3.3 KD-Tree Sampling Method

Although Z-value sampling method provides a decent solution for SIRS problem, there are adversarial cases where its query cost can be as high as $\Omega(n+k)$. To address this issues, we introduce the KD-Tree sampling method which guarantees a theoretical query cost bound of $O(\sqrt{n}+k)$ and achieves a better practical performance.

As the name of our method suggests, the basic tool we are using is KD-tree, which is a space-partitioning data structure for organizing points in a k-dimensional space. In particular, KD-tree is a binary tree in which each leaf node contains a subset of data points and each non-leaf node corresponds with a splitting hyperplane that divides the space into two parts. Each level of a KD-tree splits all children along a specific dimension at the median with a hyperplane perpendicular to the corresponding axis. At the root, all children will be split based on the first dimension. If the first dimension coordinate of a point is less than the median, it will be in the left subtree. Otherwise, it will be in the right subtree. Each level down in the tree divides on the next dimension, returning to the

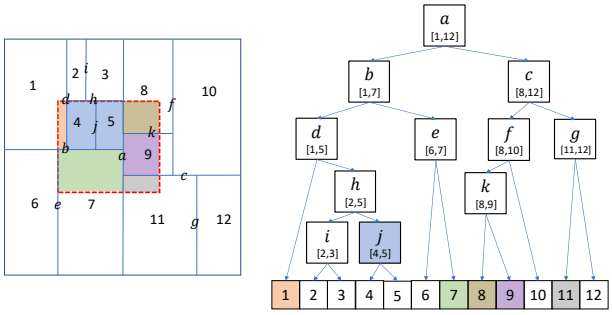


Figure 3: KD-Tree Based Space Decomposition

first dimension once all others have been exhausted. We do this recursively until each node has fewer points than a given threshold. Similar to the Z-value method, we can define a linear ordering of points using the hierarchy defined by the KD-tree such that:

- Each tree node u is corresponded to a continuous interval $[s_u, t_u]$ on data storage.
- If node u is a descendant of node v, the interval of node u is covered by that of node v, i.e. $s_v \leq s_u \leq t_u \leq t_v$.

With these properties, we can combine a KD-Tree and this data layout to fit in our sampling framework. Starting from the full dataset A, we partition it around the splitting hyperplane. We will find how many data points x lay on one side of the splitting hyperplane, while all the data points on that side are located in index range $[0, x]$ on the array and others are located in range $[x+1, |A|]$. Then, we can mark $[0, |A|]$ on the root of our KD-Tree, $[0, x]$ on its left child, and $[x+1, |A|]$ on its right child. We do this recursively until we finish building the tree structure. Fig. 3 shows an example KD-Tree with these index range tags. During tree construction, the storage array will be rearranged to guarantee these properties.

Similar to Z-value sampling method, we can decompose a query MBR into a set of tree nodes in the labeled KD-Tree, which can be further mapped to continuous intervals on the storage. For example, as illustrated in Fig. 3, the query region can be mapped to 6 tree nodes $\{1, j, 7, 8, 9, 11\}$. First each leaf node is mapped to a continuous block on the storage, all sample candidates would be contained in six intervals $\{[s_1, t_1], [s_4, t_5], [s_7, t_7], [s_8, t_8], [s_9, t_9], [s_{11}, t_{11}]\}$. Second, we build an alias structure on the length of all these intervals. Finally, we can draw each sample by retrieving a random interval from the alias structure first and then generating a random integer from the selected index range. Note that there may be some data points residing in the intervals we consider, but not in the query region. In this case, we simply reject the sample and try again.

Theoretically, the maximum number of nodes that any query region is mapped to can be bounded by $O(\sqrt{n})$ [12]. In addition, the labeled KD-Tree itself has space cost of $O(n)$ and construction cost of $O(n \log n)$. Hence, invoking Lemma 1, we can solve uniform SIRS problem with KD-tree sampling method in $O(n)$ space cost, $O(n \log n)$ preprocessing cost, and the query cost as below:

Corollary 2. *For any query range R, the KD-Tree sampling method can retrieve k independent uniform random samples from a query R in $O(\sqrt{n} + k)$ expected time.*

Similar to z-value sampling method, we can get worst-case query time guarantee by scanning all points in the boundary nodes. Note that the query cost is not related to $c(R)$, which promises a reasonable performance even in the worst case.

3.4 Generalization to Other Indexes

Inspired by the KD-Tree sampling method, we can generalize our approach to almost all hierarchical spatial indexes that support range queries such as R-trees [14] and R+-Trees [28]. Furthermore, spatial indexes for higher dimensions or even metric trees for other shaped ranges can be applied to our sampling framework as well. The simplest property sufficient for the framework to be applicable is that each data point is stored in exactly one leaf node. Then at each level of a hierarchy, we can define an ordering among the children and designate that each data point in the earlier child is ordered before that of the data points in the later child. This defines a total ordering (of leaf nodes). If leaf nodes contains single points it is a total order, if not (as we recommend) rejection sampling can be used at the leaf node level. A simple way to layout the data according to such total ordering is to do a depth-first search (DFS) on the tree and concatenate the points of each leaf node to tail of data array when it is visited.

Then on a query R, the space decomposition hierarchy can be used to decompose the range into a set to sequences: in processing each node, if a child is completely contained in R, its entire interval is used; if entirely not in R, it is ignored; and if it is partially in R, then recurse. Given a bound on the preprocessing time, and number of intervals in the decomposition, Lemma 1 can be invoked to bound space and IRS query complexity.

Note that the KD-Tree SIRS has tangible benefits compared to the R-Tree (which dominates in spatial databases) as the binary splits directly provide a logical sub-ordering, whereas R-Tree needs a second pass. Further, most R-Tree spatial decomposition (like STR packing [17]) does not have worst case bounds and thus is not immune to pathological cases. We implemented R-Tree SIRS in our sampling framework and compared it against others in Sec. 6.6.

4 WEIGHTED SIRS

We next extend out approach to support weighted SIRS query. Recall our uniform SIRS method decomposes the query region into a set of continuous ranges on a linear data layout. The weighted SIRS uses the same idea, but more structure is required.

We use the same spatial index and linear data layout $(\overline{x_1}, \overline{x_2}, \cdots, \overline{x_m})$ described in Sec. 3.2 and Sec. 3.3; this can decompose a query region into continuous index ranges $\{[s_1, t_1], [s_2, t_2], \cdots, [s_m, t_m]\}$ on the linear data layout. Second, we build an alias structure on these ranges where each range is assigned to the summed weight of all points it covers. Slightly different from the uniform SIRS solution, we build an alias structure over these index ranges with $[s_i, t_i]$ having weight $\sum_{k=s_i}^{t_i} w(\overline{x_k})$. At this point, we have distributed the total weight properly onto all ranges, thus can leverage Walker's alias method to get a random index range $[s', t']$ and do a weighted sampling on that range to retrieve a weighted SIRS sample.

In the final stage of our method described above, we have to support retrieving a weighted independent range sample from a given range on top of a sequence, which is essentially the one-dimensional weighted IRS problem. This problem has not yet been thoroughly investigated from a practical perspective. The best solution discussed in theory [5] has $O(n)$ space cost and $O(\log^* n)$ query cost.

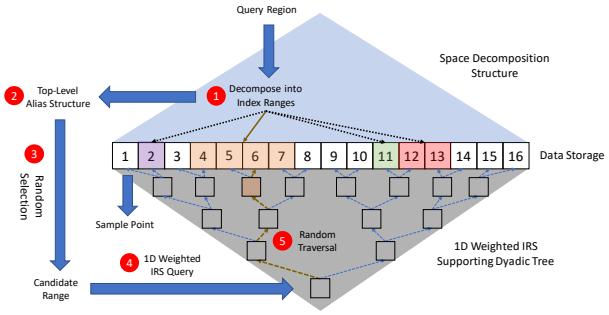


Figure 4: Dual-Tree Solution for Weighted SIRS.

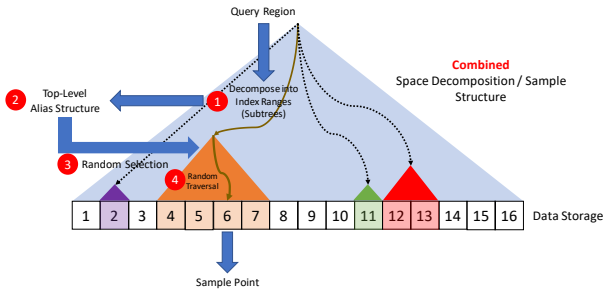


Figure 5: Combined Tree Solution for Weighted SIRS.

This structure involves building a $O(n' \log^2 n')$ sized structure over $n' = n/(\log^2 n)$ grouped points and then recursively applying the structure onto each $O(\log^2 n)$ sized group. This method has a large constant factor in the size of the structure and construction time as it requires building multiple large alias tables at every node and cumulative sums over groups of elements and individual elements. As a result of this constant factor the structure must be recursively applied many times on grouped points to decrease the size, but this recursive nesting in practice is not shallow enough to provide savings in runtime over much simpler methods. Our initial tests indicate that this method is not practical for the data scales we considered.

Another approach to this one-dimensional weighted IRS problem is to build an alias structure on each node of the binary tree over the data. Since each continuous range $[s_i, t_i]$ corresponds with one (or a small number) of such ranges, we can then issues each sample in $O(1)$ time. However, this would require $\Theta(n \log n)$ space, which is too large for massive n.

To address this challenge, we propose a new method without any rejection to answer one-dimensional weighted IRS, which has linear space and is fast enough for practical use. Specifically, we build a dyadic tree attached to the sequence such that any arbitrary query interval $[s, t]$ can be decomposed into $O(\log n)$ sub-intervals which map to tree nodes. Then on a query, we build a top-level alias structure on all returned sub-intervals with their corresponding weights. Finally, we can retrieve samples by picking a random sub-interval from the alias structure and conduct weighted sampling on it. As mentioned, pre-building an alias structure for each of these tree nodes would require $O(n \log n)$ space. Rather, to select a weighted point from each sub-interval we traverse down its corresponding subtree in $O(\log n)$ time. Even though this method gives us a $O(k \log n)$ query cost for retrieving k samples in the query interval, it does not involve any rejection at internal nodes, and hence is not influenced by query selectivity.

Dual-Tree Solution. By combining the space decomposition idea with the solution for one-dimensional IRS described above, we propose a dual-tree solution for solving weighted SIRS problem. Fig. 4 shows its dual-tree structure and corresponding sampling process. We still start from laying out our data while constructing a space decomposition structure. Then, in addition, we build a dyadic tree to support one-dimensional weighted IRS queries on the data layout generated from the last step. Now, we can decompose the query MBR in the first data structure and build the top-level alias structure to pick a continuous data range randomly, then retrieve the weighted sample by issuing an IRS query to the second one. Since both structures have space cost linear to the data size, we have $O(n)$ total space cost. For each query, we can retrieve k samples in $O(\sqrt{n} + k \log n)$ time if we use KD-Tree as our first data structure.

We remark that the solution for one-dimensional IRS problem is not restricted to our method described above. If we take the best theoretical result in the literature [5], we can solve the weighted SIRS problem in $O(\sqrt{n} + k)$ query cost.

Combined Tree Solution. Remarkably, the query ranges we issue to the second structure in the dual-tree solution is limited to the boundaries of tree nodes in the first data structure. Thus, we can merge the two supporting data structure and reduce the constant factor of both space and query cost.

We still construct the top-level alias structure by decomposing the query MBR into continuous ranges with the help of our space decomposition structure. Fig. 5 shows our combined data structure and sampling procedure for weighted SIRS problem. After selecting a random range with Walker's alias method, the sample space is reduced to that subtree's range. Then we can randomly traverse that subtree. On a query, we build a constant-size alias structure over the children of each internal node and data points of each leaf node, and recurse until we reach a leaf.

In this optimization, we not only save the space cost for the dyadic tree but also avoid constructing the second-level alias structure used in answering the one-dimensional weighted IRS query.

5 DISCUSSIONS AND EXTENSIONS

In this section, we will discuss how to extend our methods to support updates, the trade-offs in real world implementation and a potential hybrid method for fitting different circumstances.

Cost of Rejection Sampling. A major lesson we learned from solving the SIRS problem is that rejection sampling can be very expensive in practice and should be avoided as much as possible. According to our empirical evaluation, around 90% of CPU time is wasted because of rejection sampling in Olken's method (even with our LCA optimization) when the query region selects 0.1% of our default data set. This is because Olken's method can reject in any level on the tree and requires the algorithm restarting from the root every time when a rejection occurs. In contrast, our methods effectively reduce the sample rejection rate since the index ranges returned by our space decomposition has a much tighter coverage on query MBRs. More specifically, we only reject samples on the leaf nodes that are not fully covered by the query range. As a result, our methods spend less than 7% of the CPU time for processing query on rejection under the same settings.

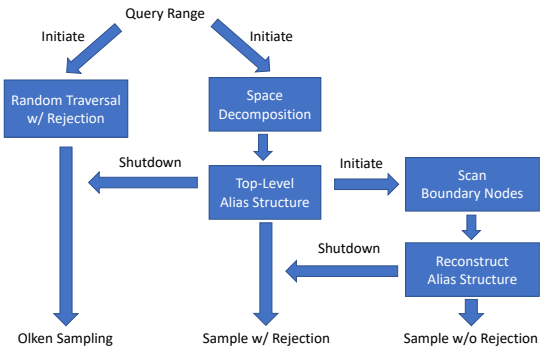


Figure 6: Hybrid Method for SIRS.

We also found that random number generation (RNG) is a very expensive operation. The fastest pseudo-random number generator we used in our implementation (which is `Pcg64Mcg` [26]) can only generate around 13 billion random real numbers between $[0, 1)$ per second. If we want to get better quality random numbers, the default cryptography-safe random number generator in Rust has a much lower throughput at around 61 million operations per second (213x slower than `Pcg64Mcg`). This makes random number generation a major bottleneck of SIRS algorithms. Hence, reducing the number of RNGs and avoiding rejections (which wastes CPU cycles on expensive RNG operations) sits in the center of designing an efficient solution for SIRS.

We also considered completely eliminating rejection. In particular, we can scan all points in leaf nodes which intersect but are not fully covered by the query region so as to find the points in those nodes that are actually covered by the query. Then, instead of regarding such nodes as candidates in the top-level alias structure, we construct a spare alias structure over the points we found from the scanning step. With the help of that, we can directly sample in the spare alias structure to avoid rejection sampling. While this does not affect theoretical asymptotic (leaf nodes are constant size), it is only effective when many points in these boundary regions are not covered, and the number of samples k is large. Note that avoiding rejections in weighted SIRS could be much more important than uniform SIRS as the point weights could heavily bias towards the uncovered points in boundary nodes.

Trade-offs and Hybrid Solution. In some narrow cases, Olken's method is preferred to our solution: when the query region covers a large portion of the data and a small number of samples are requested. To get the best of both worlds, we can build a dynamic trade-off between different methods based on query selectivity and number of samples requested with prior knowledge. However, in most cases, it is hard (and slow) to infer query selectivity in prior.

To make such trade-off easier, we can build a hybrid solution. As illustrated in Fig. 6, we start Olken's method and our SIRS solutions at the same time on two different threads and report samples from Olken's method at the beginning, and switch to our solution when the top-level alias-structure is built on the other side. Our solution can always retrieve samples much faster than Olken's method once the top-level alias structure is constructed since our solution requires less random number generation for each sample and has lower rejection rate. We can even build a third level in our hybrid solution which leverages another thread to scan points in the boundary tree node as described in the last part of this section so that we can switch to a method without rejection to achieve higher throughput in SIRS sample generation.

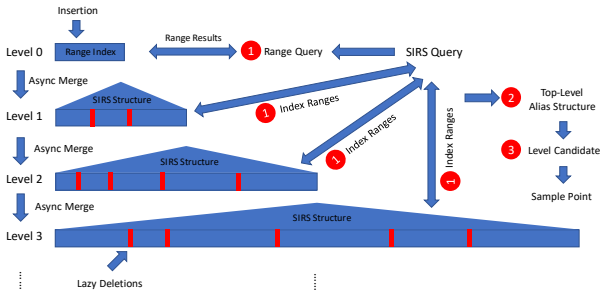


Figure 7: LSM Tree Extension for Supporting Updates

Update Support. When data is constantly changing, we cannot afford to rebuild supporting data structures upon every single update. To solve this problem, we integrate the idea of log structured merge (LSM) tree [24] into our methods.

There are two major operations, namely, insertion and deletion, we have to support. Deletion can be supported with rejection sampling: every time when an element is removed, we lazily marked it as invalid and reject it if sampled. For insertions, we can manage it in a LSM-tree manner: new elements are appended on the top level and periodically merged into lower levels. In this case, each level of our LSM-tree except for the top level is an individual data structure that supports SIRS query. The top level of our LSM-Tree is organized as either a general spatial index (like R-Tree) that supports fast insertions and range reporting queries or a simple array. During the compaction, we simply gather data points from all participating levels and build a new SIRS supporting index on top of it. As we have to build a new index in every compaction, the type of index structure hosted in all levels is orthogonal to the compaction procedure. Hence, this approach works for both uniform and weighted SIRS sampling indexes. Note that compaction is much faster for ZV-Tree based sampling indexes than KD-Tree based ones since we can leverage sorted list merging on the data layout of participating levels in building the new SIRS indexes. In contrast, KD-Tree based sampling index requires reordering data points from scratch.

For each incoming SIRS query, we issue the same query region and collect all the candidate ranges at all levels except for the top level of the LSM tree. On the top level, we simply issue a range query to retrieve all points covered by the query region. Then, we build a top level alias structure on the candidate data ranges from lower levels and query results on the top level. Finally, to retrieve each sample, we use the alias structure built above to pick a candidate range (or the top level query results) and draw samples from it with its interval sampling structure.

With a simple level compaction strategy and no bloom filter optimization, we can achieve $O(\log n)$ amortized update cost with $O(c(R)+k)$ query cost when combing ZV-tree and LSM-tree. As for the KD-tree based one, it is slightly slower on update with $O(\log^2 n)$ amortized cost, and better query cost at $O(\sqrt{n}+k)$.

As a general data structure, LSM-tree has a large design space for performance trade-offs [8–11]. In principle, our method is orthogonal to the LSM tree design space and optimizations.

Range Sampling on Disk. By now, we have been discussing how to solve SIRS problem under an in-memory environment. It corresponds with the scenario where all data can be hosted in a server with enough memory so as to support interactive analysis and exploration. When the data size is larger than memory, we can conduct the same solution on disk that still outperforms Olken's method. Note that the cost for each query will be dominated by I/Os rather than CPU. As a result, R-Tree instantiation is preferred over the KD-Tree one since it has high fan-out in internal nodes thus lower tree depth. We can further lower the cost by retrieving samples in batches to remove duplicated random I/Os. If the user only issues one fixed-size query, so does require sample independence (which is essential for many data analysis applications, as stated in Sec. 1), the original RS-Tree with sampling buffers [31] can be used to further reduce I/Os. Nevertheless, if almost all data blocks are touched at least once, then there is a simpler IO-efficient approach. We can retrieve all query-intersected data blocks and stream them into a reservoir sampler.

From another perspective, to support sampling on large data sets that cannot fit in a single machine's memory, we can extend our method into a distributed setting. Essentially, we can either partition our data based on spatial locality and issue corresponding SIRS query to one of the machines, or issue the same query to all machines and retrieve samples across them by constructing another level of alias structure to distribute sample weights.

6 EVALUATION

In this section, we evaluate our sampling methods extensively against the baselines to show its efficiency and effectiveness.

Setup. All experiments are conduct on a Ubuntu 18.04.3 LTS server with an 8-core Intel Xeon E5-2609 2.4GHz and 256GB of DRAM. We implemented all evaluated methods under a unified evaluation platform in Rust stable 1.39.0. Specifically, for uniform SIRS problem, we have the following methods:

- `QTS`: Retrieving all points in the query range by issuing a range query to the KD-Tree and sample on top of range query results.
- `KD-Olken`: Olken's method atop of KD-Tree as described in Sec. 2 *with our LCA optimization*. The version without our LCA optimization is about 12× slower, as discussed in Section 6.3.
- `KD-Buffer`: Sampling buffer method with modification to ensure independence as described in Sec. 2.
- `ZVS`: Z-Value sampling method as described in Sec. 3.2.
- `KDS`: KD-Tree sampling method as described in Sec. 3.3.

The methods below are also implemented for weighted SIRS:

- `QTS`: Retrieving all points in the query range with KD-Tree, build an alias structure on top of range query results, then retrieve samples with it.
- `KD-Olken`: Olken's method atop of KD-Tree as described in Sec. 2 with weighted sampling on each level and our LCA optimization.
- `KD-Tree Dual` and `ZV-Tree Dual`: Dual tree weighted sampling methods atop of KD-Tree and ZV-Tree.
- `KD-Tree Combined` and `ZV-Tree Combined`: Combined tree weighted sampling methods atop of KD-Tree and ZV-Tree.

Datasets and Query Generation. We evaluate all methods on the following three real-world datasets:

- USA: All nodes with coordinates in USA road network from 9th DIMCAS Challenge [1] (around 24 million points in total) where each node is assigned to a weight that sums up the length of its connecting road segments.
- Twitter: Around 240 million tweets with spatial coordinates collected in three months from Twitter streaming API. Each tweet has weight of the follower count of the corresponding user.
- OSM: Points of interest (POIs) collected by OpenStreetMap [3] with semantics tags, which contains around 2.68 billion records. We take the number of tags as the weight of each POI.

Note that we keep only point coordinates and weights in all datasets as they are the only useful information in our evaluation.

Query ranges are generated based on its selectivity over the dataset. Specifically, we generate query ranges that covers a particular percentage (from 0.01% to 1%) of all data points on average with 10% standard deviation. Note that the shape of query range (fatness in our case of rectangles) may influence query performance. We also generate query ranges with different length-width ratios spanning from $1:1$ to $1:64$. For each pair of selectivity and fatness, we generate 1000 query ranges and report the average query latency for that setting.

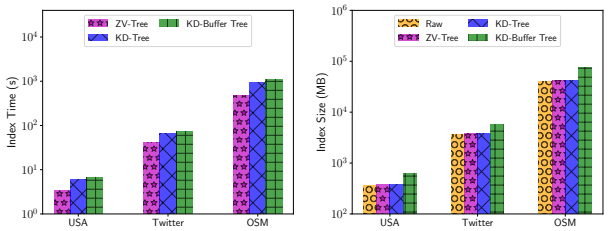
Default Parameters. Without specified, all experiments are conducted on square (i.e., width-length ratio $1:1$) query ranges that covers 0.1% of the data points over full data sets. Default value for the number of samples retrieved k is 1000; the value where most simple statistics begin to exhibit reliable convergence properties. The size of leaf nodes in KD-Trees and ZV-Tree is set to 256. The size of sampling buffers in `KD-Buffer` Tree is set to 128.

6.1 Comparison on Different Datasets

In this section, we compare all methods generally on different data sets under default settings.

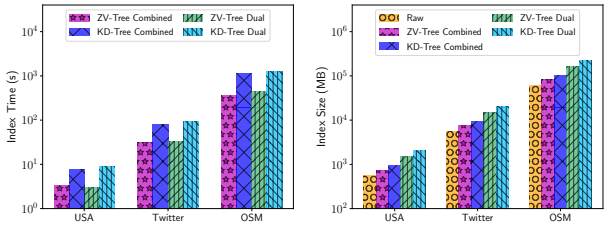
Fig. 8(a) compares index construction time of different methods for uniform SIRS problem. Note that we use the same KD-Tree for `QTS`, `KD-Olken`, and `KDS`. Hence, we only show one KD-Tree indexing cost for these three methods. As we can see, ZV-Tree takes the least time to build since it only involves a single sort over all z-values while KD-Tree requires finding a median at all levels. On the other hand, `KD-Buffer` Tree has the same tree structure as KD-Tree, but each internal node is attached to a sampling buffer. As a result, it takes 9%-17% additional time to build such sampling buffers comparing with vanilla KD-Tree. Fig. 8(b) compares the index size of different methods for uniform SIRS against raw data size (denoted as RAW in the figures). Similarly, ZV-Tree is the most concise index (2% overhead on auxiliary structures) as it has a fanout of 4 on each level while KD-Tree (4% overhead) is binary. `KD-Buffer` Tree introduces a further 50%-73% additional storage overhead comparing with original KD-Tree for attaching sampling buffer to each node.

For weighted SIRS sampling index structures, we compare their construction time in Fig. 9(a). Similar to the results for uniform above, ZV-Tree variants are faster to build compared to the corresponding KD-Tree variants. The combined tree optimization eliminates the cost of building another dyadic tree (which makes up to 20% of the index construction time), accelerating KD-Tree and ZV-Tree based sampling index construction by 8.8%-19%. Combined trees also reduce storage cost significantly as shown in Fig. 9(b). On



(a) Index Time (b) Index Size

Figure 8: Indexing Cost for Uniform SIRS.



(a) Index Time (b) Index Size

Figure 9: Indexing Cost for Weighted SIRS.

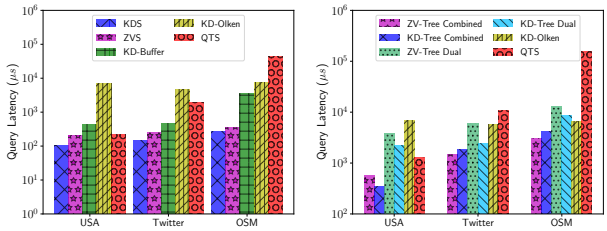
all datasets, the storage cost of combined trees are only 45%-50% of dual trees. Generally, combined trees only add a 35%-69% storage overhead against raw data (which includes around 4% on tree structures and 30%-60% on the alias structures built on leaf nodes), while dual trees are of 2.7x-3.7x raw data size (where the dyadic tree is of 1.5-2x size of raw data size). As debated in Sec. 5, SIRS queries make the most sense for in-memory environment, where combined tree optimization saves significant memory resources.

As to query latency, we compare different methods for uniform SIRS in Fig. 10(a). As we observed in the results, KDS has the best query performance on all three datasets, while ZVS has slightly worse performance. KD-Olken is the slowest method (30x-70x slower than KDS), even with our LCA optimization, on USA and Twitter datasets due to too many rejections. Note that KD-Olken can be even worse than QTS by a factor up to 32x; in these cases (e.g., the USA data set) it would be better to just issue a range query than a SIRS query in practice. Furthermore, we found the possibly counter-intuitive fact that KD-Buffer is consistently much slower (3.25x - 13x) than KDS. This is because we need to keep replenishing sample buffers attached to internal tree nodes to ensure sample independence. Fig. 10(b) shows the query latency for different weighted SIRS methods. According to the results, the combined tree optimization speeds up both KD-Tree and ZV-Tree variant by 2x-6x. ZV-Tree Combined and KD-Tree Combined vary in which has the best latency on different dataset because KD-Tree usually has less candidates in top-level alias structure while ZV-Tree is shallower than KD-Tree. KD-Olken is constantly worse than the combined tree methods due to excessive rejections.

6.2 Effect of Various Parameters

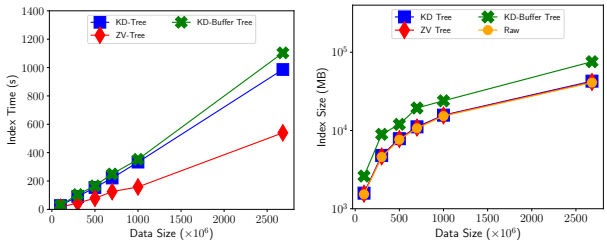
In this section, we will look into how different parameters effects the performance of indexing and SIRS queries.

Scalability. Fig. 11 shows the effect of data size on indexing cost of different uniform SIRS methods. Both indexing time and size grows linearly to the data size. ZVS has both the lowest space and time cost because it is cheaper to construct and has shallower tree structure. KD-Buffer Tree is slower to construct and bigger in size due to



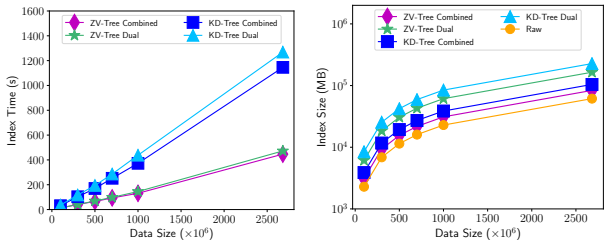
(a) Uniform (b) Weighted

Figure 10: Query Latency on Different Datasets.



(a) Index Time (b) Index Size

Figure 11: Effect of data size on Uniform SIRS Indexing.



(a) Index Time (b) Index Size

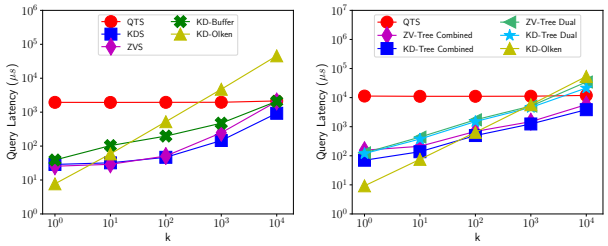
Figure 12: Effect of data size on Weighted SIRS Indexing.

additional complexity introduced by sampling buffers. Except for KD-Buffer Tree, all other sampling structures only introduce a negligible space overhead comparing to raw data.

Fig. 12 shows how data size effects indexing cost on weighted SIRS structures. Similarly, all methods grows linearly on both index construction time and index size. Comparing to KD-Tree variants, ZV-Tree based structures are more concise and faster to build. Furthermore, combined tree optimization constantly reduces memory footprint significantly (more than 50%) and makes index construction slightly faster (around 10%) as well.

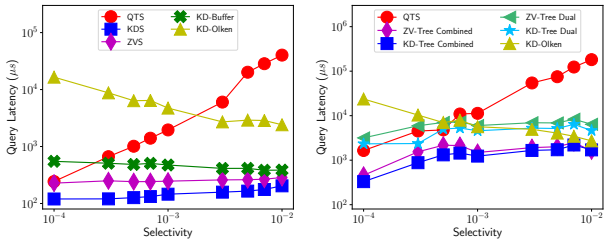
Effect of k. Fig. 13(a) shows how query latency is influenced by number of samples k requested on different uniform SIRS methods. As expected, QTS is not influenced much by k since answering range query is the dominating cost, while all other methods grow linearly to k. With all k values except for 1, KDS is the fastest method. KD-Olken returns the first result faster than any other methods as KDS and ZVS requires constructing top-level alias structure. But, KD-Olken's latency grows much faster than other methods due to excessive rejections, and finally exceeds the cost of QTS. ZVS can be up to 2x slower than KDS as KDS generally has a tighter decomposition. As explained in Sec. 6.1, KD-Buffer is slower than KDS due to the cost of refilling the sampling buffer and the gap grows bigger as k increases.

For weighted SIRS methods Fig. 13(b) shows how k influences their performance. KD-Olken, using our LCA optimization, is able to return the first 10 samples faster than any other methods, yet



(a) Uniform (b) Weighted

Figure 13: Effect of k.



(a) Uniform (b) Weighted

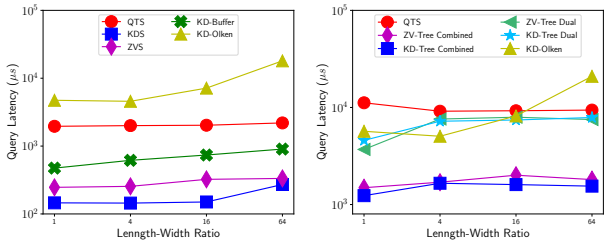
Figure 14: Effect of Selectivity.

its performance deteriorates the fastest and finally worse than QTS. Query latency of the dual tree methods grow slower than KD-Olken but it has worse performance than KD-Olken until $k > 1000$. After $k = 100$, KD-Tree Combined becomes the fastest method (up to 33% faster than ZV-Tree Combined, 9x faster than KD-Olken, and 22x times faster than QTS). Last but not least, we observed the combined tree optimization will always provide 2x-6x performance gain.

Answering a range query and its corresponding SIRS query are orthogonal. When given an SIRS query, users usually do not have any idea how many points the query range will cover. As a result, answering SIRS query can be slower than issuing range query directly when k gets close to actual point count in the range. On uniform SIRS problem, one can make a trade-off based on estimated query range count. However, to estimate it accurately with guarantees, we usually have to rely on methods structurally similar to building the sampling index. For weighted SIRS cases, even simple query count estimation cannot save us since the trade-off between range and IRS queries also relies on the distribution of the weights.

Effect of Selectivity. The selectivity of query range (i.e., how many points are covered by the query range) is an important factor influencing query performance. Even though it is usually hard to estimate, it is worth verifying how different methods are affected. Fig. 14(a) demonstrates such effects on uniform SIRS methods. As expected, QTS grows linearly to query range selectivity as it requires answering the full range query. In contrary, KD-Olken's performance gets better when the query range covers more points as samples have less chance to be rejected. KDS and ZVS are influenced much less by this factor. This is because the performance of constructing top-level alias structure is pretty stable at $O(\sqrt{n})$ or $O(\log c(R))$ level for square ranges, while the actual sampling procedure is not influenced by selectivity at all. KD-Buffer is also not affected much by selectivity as its dominating costs are refilling sample buffers and reconstructing query frontier.

For weighted SIRS methods, we show the effect of selectivity in Fig. 14(b). Again QTS increases linearly with range selectivity, and KD-Olken becomes faster. Both combined tree and dual tree



(a) Uniform (b) Weighted

Figure 15: Effect of Range Fatness.

methods are slightly influenced by query selectivity. Combined tree methods constantly achieve 3x-7x performance gain against dual tree methods. Note that the major performance gain of KD-Tree Combined and ZV-Tree Combined comes only from fewer rejections, but not lower cost on each sample as in the uniform case. This is because these methods will traverse a subtree for each sample in the sampling stage. Hence, the performance gap between KD-Olken and our methods is close when selectivity is high. Since there is a high weight bias on the twitter dataset, the effect of selectivity in weighted SIRS cases is not as stable as that in uniform ones.

Effect of Range Fatness. In this set of evaluation, we look into how query shape effects performance of different methods. In particular, we fix the selectivity of the query range and change the length-width ratio of the query MBR. Fig. 15(a) shows how such a factor influences different uniform SIRS methods. QTS is not influenced at all, the dominating cost of the range query is $O(c(R))$. KDS and ZVS grows slightly by query fatness. The main influence of range fatness to such methods are from rejections in boundary nodes, the total number of which grows slightly as the range becomes fatter. KD-Olken is effected by query fatness more since each rejection in KD-Olken involves more RNG calls (each a factor equal to the depth of the subtree) than the case of KDS (which is 2). The effect of range fatness on KD-Buffer is also mild since query frontier in KD-Buffer is narrow for a limited number of samples (1000 in this case). Thus, it will not involve much more replenishment of the sampling buffer as the length-width ratio grows.

Fig. 15(b) shows the weighted SIRS methods. Similarly, QTS is not influenced by query fatness, while KD-Olken's performance degrades when the query range grows fatter. KD-Olken becomes slower when more rejection is involved. All other methods are influenced by this factor mainly because of the combined effect of weight bias and the number of partially covered boundary nodes. But overall, they are fairly stable against the change of range fatness.

6.3 CPU Breakdown

We identify different methods' CPU cost breakdown. This will not only help us understand their bottlenecks, but also guide the trade-off between these methods. We measure CPU time of different component in KD-Tree based sampling methods but omit those for ZV-Tree as they are similar. Fig. 16(a) shows the breakdown of uniform SIRS methods. In QTS, more than 98% of its query time is for answering the range query and all RNG calls are effective since no rejection is occurred. On the other hand, KD-Olken spends most of its CPU cycles in rejecting samples out of the query range. When our LCA optimization is not applied all random paths need to start from tree root. This optimization reduces the query latency by 92% and only incurs a negligible cost – hence all other comparisons

Method	Tot Latency (μs)	CPU Breakdown (μs / %)		
		Effective RNGs	Wasted RNGs	Other Major Components
QTS	1892.64	11.20 (0.60%)	0.00 (0.00%)	Query Time: 1881.44 (99.41%)
KD-Olken w/o LCA	62078.03	642.03 (1.03%)	61435.55 (98.97%)	-
KD-Olken w/ LCA	4981.30	477.31 (9.58%)	4411.35 (88.56%)	LCA Optimization: 2.64 (0.05%);
KD-Buffer	798.56	8.69 (1.09%)	2.97 (0.37%)	Buffer Replenish: 270.53 (57.95%);
KDS w/ Rejection	140.26	99.45 (70.90%)	6.73 (4.80%)	Alias Construction: 23.80 (16.96%);
KDS w/o Rejection	396.30	98.24 (24.79%)	0.00 (0.00%)	Alias Construction: 289.79 (73.12%);

(a) Uniform

Method	Tot Latency (μs)	CPU Breakdown (μs / %)		
		Effective RNGs	Wasted RNGs	Other Major Components
QTS	11128.86	112.41 (1.10%)	0.00 (0.00%)	Query Time: 11006.45 (98.90%)
KD-Olken w/o LCA	70328.76	483.38 (0.69%)	69844.77 (99.32%)	-
KD-Olken w/ LCA	5770.88	355.40 (6.16%)	5412.44 (93.79%)	LCA Optimization: 3.04 (0.05%)
KD-Tree Dual w/ Rej	2491.19	2293.56 (92.07%)	115.31 (4.62%)	Alias Construction: 79.80 (3.20%)
KD-Tree Dual w/o Rej	3143.37	2242.30 (71.33%)	0.00 (0.00%)	Alias Construction: 896.03 (28.51%)
KD-Tree Combined w/ Rej	1245.58	1137.30 (91.31%)	36.29 (2.91%)	Alias Construction: 70.56 (5.66%)
KD-Tree Combined w/o Rej	1356.54	491.69 (36.24%)	0.00 (0.00%)	Alias Construction: 863.08 (63.62%)

(b) Weighted

Figure 16: CPU Breakdown.

use it. It reduces not only the number of rejections, but also cuts the cost of effective RNGs since random paths only need to start from the root of a subtree. KD-Buffer spends less time in acquiring the samples as they already exist in the sampling buffer. However, the major cost comes from maintaining query frontier which takes 40% of query time and replenishing the sample buffer so as to ensure independence. KDS has a moderate cost in building the alias structure and also a fairly low rate of rejection. In particular, only 6.3% of generated samples are rejected (4.8% of CPU time). We can eliminate all rejections by scanning all points that resides in leaf nodes partially covered by the query range. Such an optimization bumps up the cost of the alias structure construction by 10 times but eliminate rejected RNGs. Nevertheless, it slows down the SIRS query asking for 1000 samples by 2.8x.

Fig. 16(b) shows the CPU breakdown for weighted SIRS methods. The pattern is similar for QTS and KD-Olken. Comparing to KD-Tree Combined, KD-Tree Dual requires an additional alias structure construction on the dyadic tree level and also involves more RNGs. Even with the same rejection rate, the overall cost of KD-Tree Dual is 2x higher than KD-Tree Combined. Finally, scanning on all boundary nodes to eliminate rejection is also not a good idea for 1000 samples in the weighted case; in particular, in KD-Tree Combined it is the dominating cost.

6.4 Hybrid Method

We implement a hybrid method (see Section 5) which automatically transitions from KD-Olken to KDS to KDS without rejection as more samples are delivered. We compare it with different methods and report how many samples they can return at key time points, such as the finishing of the construction of top-level alias structure and scanning boundary nodes, to report total number of samples. Fig. 17(a) shows the performance of KD-Olken, KDS, and KDS w/o rejection hybrid. As we can see, our hybrid method is able to start reporting a decent number of samples before KDS finishes constructing the top-level alias structure. Although hybrid method reported less samples from 90 μs to 1000 μs, our hybrid method is able to report samples the fastest and has the highest overall throughput at the end. The main reason why it has not dominated all other methods in all time is that context switches between different method is not free, and scanning boundary nodes while generating samples degrades CPU cache performance.

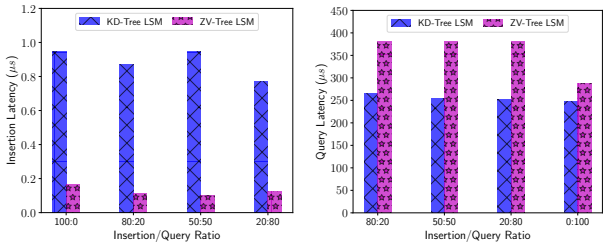
Method	# Samples Retrieved by Timeline (μs)						
	74	91	443	461	1000	3000	5000
Olken	86	106	517	538	1166	3498	5830
KDS w/ Rej	0	287	6237	6541	15651	49454	83257
KDS w/o Rej	0	0	0	364	11263	51706	92149
Hybrid	82	101	882	922	11877	52524	93172

(a) Uniform

Method	# Samples Retrieved by Timeline (μs)						
	109	127	1570	1974	3000	5000	10000
Olken	222	243	2542	3045	4477	7962	14923
Comp w/ Rej	0	72	5855	7474	11585	19600	39636
Comp w/o Rej	0	0	0	1774	6279	15060	37014
Hybrid	216	252	3622	4566	9078	17875	39866

(b) Weighted

Figure 17: Effectiveness of Hybrid Method.



(a) Insertion Latency (b) Query Latency

Figure 18: Update Support with LSM Tree.

Fig. 17(b) shows the case of weighted SIRS hybrid method. Similarly, it is able to report the first few samples fast, a bit less in a short period than KDS, and eventually dominate all other methods.

6.5 Update Support

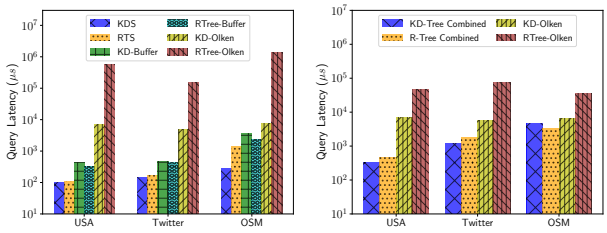
In Sec. 5, we described how to support updates in our sampling structure with LSM trees. To verify such an idea is viable in practice, we implement simple versions of it on top of KD-Tree and ZV-Tree, denoted as KD-Tree LSM and ZV-Tree LSM respectively. Fig. 18 shows how they perform on different workloads. In particular, we first populate 100 million points in each workload and issue 100000 operations with different insertion-query ratio.

As shown in Fig. 18(a), insertions in ZV-Tree LSM is 7x-9x faster than those in KD-Tree LSM. This is because insertions of ZV-Tree LSM only involves merging sorted sequences. In contrast, KD-Tree LSM requires a full KD-Tree construction because its data cannot be described by a simple mergeable full order. On the other hand, ZV-Tree LSM is up 50% slower than KD-Tree LSM, which has better query complexity. Note, these methods provide a proof of concept. In particular, we did not implement deletion operations in these methods due to the complexity of managing rejection rate after lazy deletions. Besides, there is a large design space for LSM trees on aspects like bloom filter optimization and internal passes within levels. We did not incorporate these optimizations, but identify them as good directions of future work.

6.6 Method Generalization

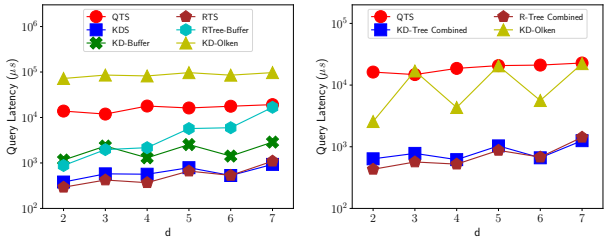
To show the generality of our framework, we extend our methods to other index structures (namely R-Trees) and higher dimensions.

Generalize to R-Tree. We follow Sec. 3.4 on R-Trees (dubbed as RTS). We use 256 points in leaf nodes and a fan-out of 25 in internal nodes. Fig. 19 shows the query latency of R-Tree variants comparing



(a) Uniform (b) Weighted

Figure 19: Comparing R-Tree Instantiations with KD-Tree.



(a) Uniform (b) Weighted

Figure 20: SIRS Extension to Higher Dimensions.

to KD-Tree ones over uniform and weighted SIRS queries. As mentioned in Sec. 3.4, KD-Tree variants are preferred under in-memory settings as they are immune to pathological cases, and are easier in building indexes since they do not require the second ordering pass. R-Tree variants occasionally provide better performance (see Fig. 19 and 20) due to lower tree height and potential better alignment between the query region and internal R-Tree nodes. The R-Tree implementation for previous state of the art [31] with independence (dubbed as `RTree-Buffer`) is faster than `KD-Buffer` as the tree is shallower in depth thus takes less time to replenished sampling buffers. Olken sampling over R-Tree (dubbed as `RTree-Olken`) is slower than that on KD-Tree because of more rejections. The indexing cost of the R-Tree variants is similar to KD-Tree. It is slightly slower to build R-Trees since they involve an extra pass of reorganizing the data layout, while the index size of R-Tree is slightly smaller due to higher internal node fan-out.

Higher Dimensions. We also extend our method to work on higher-dimensional data sets. Specifically, we run the extended methods over GDELT dataset [2] which has 80 million records with six spatial dimensions and one timestamp. We take pairs of spatial dimensions when dimensionality d is even, and add the timestamp dimension when d is odd. As shown in Fig. 20, our methods (KD-Tree and R-Tree variants) achieve much better performance than baselines by 4x-200x. The indexing cost is growing linearly with d, and shows similar trends as with 2D data, thus omitted.

7 RELATED WORKS

Range sampling is a classic problem in databases rooted from the query sampling goal of returning a random sample set of element in a range. The usefulness of such a sample set has long been recognized even in the non-big-data era [22]. More than three decades ago, Olken et. al published the classic solution to sample data from B+-tree indexes by random tree traversal [23] and was recognized as the correct way to do query sampling.

Recently, the theory community has provided better query complexity bounds. Hu. et al [16] proposed methods for uniform independent range sampling for one-dimensional data over value range intervals. In particular, they described a new structure of $O(n)$ space that answers a query in $O(\log n + k)$ expected time and supports an update in $O(\log n)$ time. Later, Afshani et. al. [5] proposed the solution for weighted independent range sampling for one-dimensional data that uses $O(n)$ space and answers a query in $O(Pred(U, w, n) + k)$ time. Specifically, $Pred(U, w, n)$ is the query time of a predecessor search data structure that use $O(n)$ space on an input of size n from the universe $[U]$ and on a machine with w-bit integers [25]. In the same paper, the authors obtain an optimal data structure for 3D halfspace ranges for uniform independent range sampling problem. Given a query half space h, it can extract k independent uniform random samples from h in $O(\log n + k)$ expected time using a $O(n)$ space structure. This method further implies optimal data structure for two-sided and three-dimensional dominance queries. More recently, Afshani et. al. [4] revisited again the independent range sampling problem on halfspaces and obtain a solution for weighted input for 3D halfspace queries that has $O(n \log n)$ space cost and $O(\log^2 n + k)$ expected query time. When the ratio of weights is $n^{O(1)}$, the space complexity of proposing structure can be reduced to $O(n)$. The authors also provide further tradeoffs if allowing approximation of sample probabilities. Note that all above works do not study the SIRS problem we covered in this paper specifically. Furthermore, no known implementations have been made for these complex methods.

A recent work [31] is published to solve uniform spatial range sampling problem on minimum bounding rectangles(MBRs). The authors started from adapting Olken's method [23] to R+-Trees [28], and then optimized Olken's method by attaching pre-built sampling buffers to each internal tree nodes (we call this the RS-Tree). Despite achieving superior performance against its baseline, RS-Tree does not provide sample independence. When asked the same query twice, RS-Tree will likely return the exact same samples. It is possible to enforce sample independence by invalidating reported samples while constantly replenishing depleted sampling buffers. However, we have shown it expensive in our evaluation results.

8 CONCLUSION

We studied the uniform and weighted spatial independent range sampling (SIRS) problem aiming at retrieving random samples with independence over points residing in minimum bounding rectangles (MBRs). We designed concise index structures with careful data layout based on various space decomposition strategies, and proposed novel algorithms for both uniform and weighted SIRS queries with low theoretical complexity and excellent practical performance, showing orders of magnitude improvement over existing baselines. Overall, we propose using KDS, which better handles dynamic updates and avoids adversarial cases. In future work though, we will explore in more details on various choices of tree structures, space decomposition strategies and fanout, as well as how to adapt the combined sampling LSM trees to different workloads with updates.

Acknowledgment. Dong Xie was supported by a Microsoft Research Ph.D Fellowship. Jeff M. Phillips thanks his support from NSF CCF-1350888, CNS-1514520, IIS-1619287, IIS-1816149, CNS-1564287, CFS-1953350, and an award from Visa Research. Feifei Li thanks his support from NSF CNS-1514520, IIS-1619287, IIS-1801446, and IIS-1816149.

REFERENCES

[1] 9-th dimacs implementation challenge: Shorted paths. http://users.diag.uniroma1.it/challenge9/download.shtml.

[2] The gdelt project. https://www.gdeltproject.org/.

[3] Openstreepmap. https://planet.openstreetmap.org/.

[4] P. Afshani and J. M. Phillips. Independent range sampling, revisited again. In *SoCG*, pages 4:1–4:13, 2019.

[5] P. Afshani and Z. Wei. Independent range sampling, revisited. In *ESA*, pages 3:1–3:14, 2017.

[6] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. Blinkdb: Queries with bounded errors and bounded response times on very large data. In *EuroSys*, 2013.

[7] J. L. Bentley. Multidimensional binary search trees in database applications. *IEEE Trans. Software Eng.*, 5(4):333–340, 1979.

[8] N. Dayan, M. Athanassoulis, and S. Idreos. Monkey: Optimal navigable key-value store. In *SIGMOD*, pages 79–94, 2017.

[9] N. Dayan, M. Athanassoulis, and S. Idreos. Optimal bloom filters and adaptive merging for lsm-trees. *TODS*, 43(4):16:1–16:48, 2018.

[10] N. Dayan and S. Idreos. Dostoevsky: Better space-time trade-offs for lsm-tree based key-value stores via adaptive removal of superfluous merging. In *SIGMOD*, pages 505–520, 2018.

[11] N. Dayan and S. Idreos. The log-structured merge-bush & the wacky continuum. In *SIGMOD*, pages 449–466, 2019.

[12] M. de Berg, M. van Kreveld, M. Overmars, and O. Cheong. *Computational Geometry: Algorithms and Applications*. Springer, 2nd edition, 2000.

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

[14] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, 1984.

[15] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *SIGMOD*, pages 171–182, 1997.

[16] X. Hu, M. Qiao, and Y. Tao. Independent range sampling. In *PODS*, pages 246–255, 2014.

[17] S. T. Leutenegger, M. Lopez, J. Edgington, et al. STR: A simple and efficient algorithm for R-tree packing. In *ICDE*, 1997.

[18] M. Matheny, R. Singh, L. Zhang, K. Wang, and J. M. Phillips. Scalable spatial scan statistics through sampling. In *SIGSPATIAL*, 2016.

[19] P. A. P. Moran. Notes on continuous stochastic phenomena. *Biometrika*, 37(1/2):17–23, 1950.

[20] J. Nievergelt, H. Hinterberger, and K. C. Sevcik. The grid file: An adaptable, symmetric multikey file structure. *TODS*, 9(1):38–71, 1984.

[21] M. A. Oliver and R. Webster. Kriging: a method of interpolation for geographical information systems. *International Journal of Geographical Information Systems*, 4(3):313–332, 1990.

[22] F. Olken. *Random sampling from databases*. PhD thesis, University of California, Berkeley, 1993.

[23] F. Olken and D. Rotem. Random sampling from B+ trees. In *VLDB*, pages 269–277, 1989.

[24] P. E. O'Neil, E. Cheng, D. Gawlick, and E. J. O'Neil. The log-structured merge-tree (lsm-tree). *Acta Inf.*, 33(4):351–385, 1996.

[25] M. Pătraşcu and M. Thorup. Time-space trade-offs for predecessor search. In *STOC*, pages 232–240, 2006.

[26] W. H. Payne, J. R. Rabung, and T. P. Bogyo. Coding the lehmer pseudo-random number generator. *CACM*, 12(2):85–86, 1969.

[27] D. W. Scott. *Multivariate Density Estimation: Theory, Practice, and Visualization*. Wiley Series in Probability and Statistics. Wiley, 1992.

[28] T. K. Sellis, N. Roussopoulos, and C. Faloutsos. The r+-tree: A dynamic index for multi-dimensional objects. In *VLDB*, pages 507–518, 1987.

[29] V. Vapnik. Principles of risk minimization for learning theory. In J. E. Moody, S. J. Hanson, and R. Lippmann, editors, *NIPS*, pages 831–838. Morgan Kaufmann, 1991.

[30] A. J. Walker. New fast method for generating discrete random numbers with arbitrary frequency distributions. *Electronics Letters*, 10(8):127–128, 1974.

[31] L. Wang, R. Christensen, F. Li, and K. Yi. Spatial online sampling and aggregation. *PVLDB*, 9(3):84–95, 2015.

[32] Y. Zheng, Y. Ou, A. Lex, and J. M. Phillips. Visualization of big spatial data using coresets for kernel density estimates. In *Visual Data Science*, 2017.