Waffle: A Workload-Aware and Query-Sensitive Framework for Disk-Based Spatial Indexing

Moin Hussain Moti
HKUST
Hong Kong
mhmoti@connect.ust.hk

Panagiotis Simatis
HKUST
Hong Kong
psimatis@connect.ust.hk

Dimitris Papadias
HKUST
Hong Kong
dimitris@ust.hk

ABSTRACT

Although several spatial indexes achieve fast query processing, they are ineffective for highly dynamic data sets because of costly updates. On the other hand, simple structures that enable efficient updates are slow for spatial queries. In this paper, we propose *Waffle*, a workload-aware, query-sensitive spatial index, that effectively accommodates both update- and query-intensive workloads. Waffle combines concepts of the space and data partitioning frameworks, and constitutes a complete indexing solution. In addition to query processing algorithms, it includes: (i) a novel bulk loading method that guarantees optimal disk page utilization on static data, (ii) algorithms for dynamic updates that guarantee zero overlapping of nodes, and (iii) a maintenance mechanism that adjusts the trade-off between query and update speed, based on the workload and query distribution. An extensive experimental evaluation confirms the superiority of Waffle against state of the art space and data partitioning indexes on update and query efficiency.

PVLDB Reference Format:
Moin Hussain Moti, Panagiotis Simatis, and Dimitris Papadias. Waffle: A Workload-Aware and Query-Sensitive Framework for Disk-Based Spatial Indexing. PVLDB, 16(4): 670 - 683, 2022.
doi:10.14778/3574245.3574253

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://gitlab.com/moinmoti/waffle.

1 INTRODUCTION

Spatial databases manage large data sets of objects with location information (e.g., GPS positions of smartphone users, vessel and aircraft coordinates). Due to the sheer volume of data, and since there is no natural ordering in multi-dimensional space, linear search for objects that satisfy some spatial predicate (e.g., mobile users in the city center, the closest airplanes to the airport) is impractical. Thus, spatial indexes [17, 47, 48] have been developed to efficiently filter and retrieve information. Spatial indexes are usually trees that either partition the space, or the data objects. The deepest tree level comprises *data nodes*, whereas the rest, including the root, consist of *directories*. Each node is associated with a spatial extent (e.g., a minimum bounding rectangle in 2D space) which covers all the nodes and objects in its subtree. The maximum number of children per node is determined by the disk page size.

Among the most common spatial information processing tasks is the *range* query: given a (usually rectangular or circular) region r, output all objects in r. Such a query is processed by accessing the root of the spatial index and recursively retrieving all nodes that intersect r. Instead of a range, a *point* query includes a point l, and outputs the object(s) located at l. Another common query is k *nearest neighbors* (kNNs): given a location l and an integer k, output the k nearest objects to l. Processing starts from the root and visits the nodes containing possible results in a variety of manners [20, 22, 43]. Observe that a range r or kNN query may follow multiple paths from the root to the leaf level because (i) several nodes may intersect r or contain kNNs, or (ii) the indexing scheme allows overlapping nodes.

Although several spatial indexes (e.g., R*-Trees) enable fast query processing, they are ineffective for highly dynamic data sets because updates may be very expensive due to extensive reorganization of the tree structure. Instead, simple grid-based solutions are often preferable to hierarchical indexes for update-intensive applications [35, 58, 59], despite their inferior query performance. Motivated by the query vs. update cost trade-off, we propose Waffle[1], a novel disk-based spatial index, which delivers excellent performance and can accommodate both update- and query-intensive workloads. Waffle constitutes a complete indexing solution that includes efficient algorithms for bulk loading, dynamic updates and query processing. Concretely, our contributions are:

(1) Waffle, a disk-based framework that auto-adapts to all spatial indexing purposes.
(2) A novel bulk loading method that guarantees optimal disk page utilization on static data, and emphasizes square-like nodes.
(3) Algorithms for dynamic updates that create non-overlapping nodes.
(4) A maintenance mechanism that monitors the query workload and distribution, continuously optimizing the index.
(5) An extensive experimental evaluation, which verifies that Waffle outperforms state of the art spatial indexes on update and query performance.

The rest of the paper is organized as follows. Section 2 overviews related work, focusing on the spatial indexes used as benchmarks in our experiments. Section 3 describes Waffle's bulk loading algorithm. Section 4 presents the algorithms for dynamic updates, and Section 5 for query processing. Section 6 contains the experimental evaluation, and Section 7 concludes the paper.

[1]The name Waffle is used to emphasize the square-like nodes in our index.

Proceedings of the VLDB Endowment, Vol. 16, No. 4 ISSN 2150-8097.
doi:10.14778/3574245.3574253

2 RELATED WORK

Spatial indexes are usually tree structures, where (i) each leaf node (also called *data node*) stores points, or minimum bounding rectangles (MBRs) of data objects, and pointers to the corresponding records, and (ii) each internal node (also called *directory*) stores pointers to child nodes that fall within its extent. They are classified depending on the type of storage that they reside. Primary memory indexes (e.g., KD-Trees [5], 2-level [55], CR-Trees [27], BLOCK [37]) are stored entirely in RAM. They are agile and suitable for relatively small scale applications. On the other hand, secondary memory indexes (e.g., KDB-Trees [42], the R-Tree family) reside primarily on the disk, with commonly accessed nodes (i.e., directories) kept in main memory for swift pointer chasing. Disk-based indexes prioritize minimizing the I/O cost (i.e., disk page accesses) over CPU time. Nodes have a maximum capacity C, which is constrained by the disk page size. Often, they also have a minimum capacity, so that there are guarantees on the index size and query performance. Our focus is on disk-based spatial indexes for 2D data points because they are ubiquitous, as they are used to capture mobile users, cars, ship trajectories etc. Moreover, their sheer volume necessitates disk based indexes in most applications.

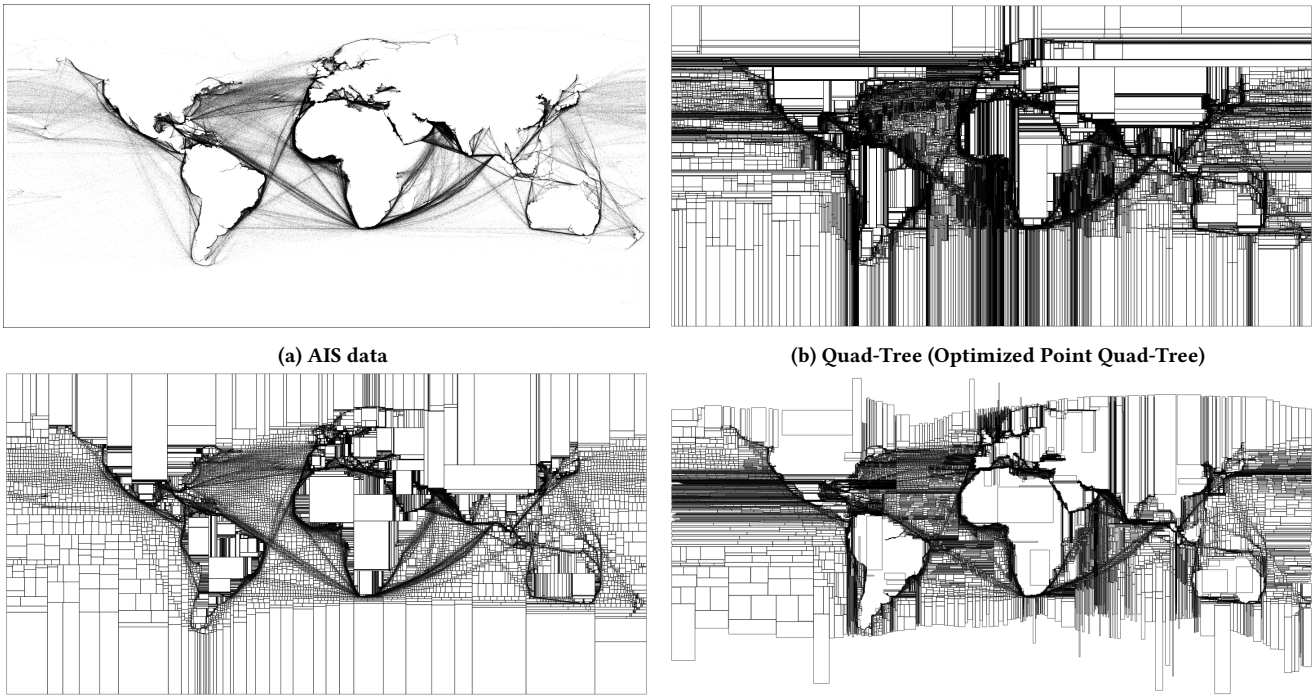
Assuming that the data are given in advance, *bulk loading* packs the index entries into pages, which are then used to build a compact index. Various bulk loading methods have been proposed for spatial data [2, 18, 24, 32, 41, 44]. Dynamic data (e.g., moving objects) necessitate efficient insertion and deletion algorithms. Updates in spatial indexes can be expensive because node overflows may lead to extensive reorganization of the tree. Thus, dynamic indexes are accompanied by construction algorithms that rearrange the tree structure on the fly, with the aim of optimizing the index, while minimizing the update cost. Depending on the bulk loading or update algorithms, spatial indexes are classified as either *space* or *data* partitioning. Space partitioning schemes generate, at each tree level, disjoint rectangular nodes that cover the entire space. The simplest approach is Grid [6], which decomposes the space into equally sized, axis-parallel cells. In practice, more complex hierarchical structures (e.g., KDB-Trees [42], Quad-Trees [15], LSD-Trees [21], hB-Trees [34]) have better query performance. These recursively divide the space until each page has a number of objects between its minimum and maximum capacity. On the other hand, data partitioning indexes (e.g., the R-Tree [19] family) generate, at each tree level, possibly overlapping nodes that aim at minimizing measures such as overlap between nodes, perimeter, and *dead* space (i.e., empty area) inside nodes. Commonly, the node extents are represented by MBRs, but techniques utilizing other geometric shapes have been proposed (e.g., M-Trees [11], SS-Trees [57], SR-Trees [26]). There also exist methods that combine data and space partitioning (e.g., Hybrid-Trees [8]).

In the following we elaborate on common spatial indexes, used as benchmarks in our experiments, starting with the R-Tree family. Bulk loading R-Trees to achieve full nodes and high quality MBRs is a well-studied problem. Several techniques sort the data objects on an axis or by a space-filling curve (e.g., Hilbert [25]), and then recursively group them in nodes. Sort-Tile-Recursive (STR) [32], one of the most effective methods, first computes the number of pages required as $P = \lceil N/C \rceil$, where N is the data set's cardinality, then sorts the data set on the x-axis, and "tiles" the points into $\lceil \sqrt{P} \rceil$ vertical slices. If the resulting slices cause an overflow, they are sorted on the y-axis and the algorithm continues recursively. Among the dynamic R-Tree variants (e.g., R+-Trees [49], R*-Trees [3], RR*-Trees [4]), the most popular is the R*-Tree [3], a height-balanced index that uses the query processing algorithms of the original R-Tree [19], but achieves better structure through sophisticated update algorithms. When a node overflows for the first time, instead of splitting the node directly, the R*-Tree performs *reinsertion*. Reinsertion sorts the entries of the full node in increasing order of their distance to the node center, and reinserts a percentage (e.g., 30%) of the farthest entries. Only if the overflow persists, the node splits. Splitting involves sorting the entries on all dimensions in order to select the *split axis*, yielding the minimum perimeter for the new nodes. Among all the groupings of entries on the split axis that satisfy the minimum and maximum capacity constraints, the new nodes adopt the one with the minimum overlap. Caching techniques [51, 52] that perform updates in batches, possibly in multiple versions of R-Trees, have been used to speed up update and query performance. R-trees have also been employed for tracking moving objects [9, 10] on road networks. Although originally proposed for R-Trees, the above techniques are applicable to all spatial indexes based on tree structures.

The Quad-Tree is an umbrella term for a plethora of 2D space partitioning indexes [46], in which directories maintain four children, called *quadrants*. Depending on the partition strategy, different Quad-Trees exist. The PR-Quad-Tree[2] decomposes a node into equally-sized quadrants. The decomposition continues recursively until no data node exceeds the maximum capacity C. The partition occurs at the center of the node, ignoring the distribution of the data. Point-Quad-Trees take into account the data during partitioning. Given a set of points, the Optimized Point Quad-Tree [15] bulk loads them in three steps: (i) the points are sorted on the x-axis, with ties resolved by a secondary sort on the y-axis, (ii) the median point serves as the root for the subtree, and (iii) the remaining points are assigned to the appropriate quadrant. This process continues recursively until all the nodes satisfy the capacity constraints. A similar sort-based approach is used for node splits during dynamic updates. This can lead to unbalanced trees, where leaf nodes in dense areas are at deeper tree levels than those in sparse areas. For the rest of paper, the term Quad-Tree refers to the Optimized Point-Quad-Tree, used as a benchmark in our evaluation.

The KDB-Tree [42] splits nodes on the median entry of the split axis, which alternates among dimensions. For example, in 2D, after the first node overflow, the entries are sorted on the x-axis and the split occurs on the x-coordinate of the median entry. The second split is on the y-axis, the third again on the x-axis and so on. Spread Split [16], is an improved version, which instead computes the distance between the farthest points on each dimension, and splits on the axis with the maximum distance. In case of dynamic updates, the KDB-Tree remains balanced through cascading splits. Specifically, when a directory is partitioned by a split s, its entire subtree is also divided by s, which may lead to extensive reorganization of the tree structure. During bulk loading, cascading splits can be avoided

[2]Generally $C = 1$, while bucket variants have $C > 1$. We drop the term bucket for brevity.



(a) AIS data

(b) Quad-Tree (Optimized Point Quad-Tree)

(c) KDB-Tree (Spread Split)

(d) R-Boost

Figure 1: Bulk loaded data nodes for the AIS data set, cardinality $N = 10^7$, capacity $C = 204$ points (4KB page size)

Table 1: Data node properties

Measure	Quad	KDB (Spread)	R-Boost
#Data Nodes	124342	65536	49020
Avg. #Points/Data Node	80.42	152.59	204.00
Avg. Data Node Perimeter	2.61	1.92	3.00

using a top-down build strategy. The tree starts with a single node covering the entire space and grows by breaking nodes at the bottom level over multiple rounds until no page exceeds the capacity C. Concretely, when a data node n^0 overflows, n^0 is upgraded to a directory n^1 (the superscripts denote the height of a node in the index, e.g., 0 means leaf level). Then, n^1's space is partitioned into two data nodes n_1^0 and n_2^0. If n_1^0 and n_2^0 contain more than C points and splitting them can be accommodated by n^1's fanout, each data node is split, so that their number doubles in each step. When n^1's fanout is reached, this recursive process is repeated for every data node exceeding C. Our KDB-Tree implementation involves Spread Split, and the above bulk loading method.

Figure 1a illustrates 10 million points corresponding to ship locations from the AIS data set, used in our experimental evaluation. Land masses (e.g., Africa) are empty, while popular shipping lanes (around Cape of Hope in South Africa) contain numerous points. Figure 1b to Figure 1d show the data nodes of the bulk loaded Quad-Tree, KDB-Tree, and R-Boost [3]. The disk page size is set to 4KB, yielding a maximum leaf node capacity of $C = 204$ points for all indexes. A good partitioning scheme should adapt to the data distribution. In addition, it should avoid long and thin (i.e., *spaghetti*) nodes because they have a negative impact on query processing. Based on visual inspection, the KDB-Tree and R-Boost outperform the Quad-Tree as they generate large nodes in empty areas, and capture well the high density of shipping lanes with small nodes. Observe that since R-Boost uses data partitioning, it does not have data nodes covering some empty areas.

Table 1 verifies the visual observations of Figure 1 by including the number of data nodes, the average number of points and perimeter per data node for each bulk loading method. The KDB-Tree has 65536 data nodes, compared to 124342 for the Quad-Tree. Moreover, each leaf node of the KDB-Tree has on average 159.59 points, as opposed to only 80.42 for the Quad-Tree. R-Boost has the minimum number (49020) of leaf nodes, most of which are full. However, the data nodes of R-Boost have larger average perimeter 3.00 (i.e., they are more spaghetti-like) compared to those of the KDB-Tree (average perimeter 1.92). Finally, directory nodes of R-Boost may exhibit overlap, while those of the KDB-Tree are disjoint.

Spatial indexes have also been applied to moving objects [23, 45], spatio-temporal aggregation [54], and cloud-based partitioning of spatial data [1]. The recent exponential growth of geo-tagged data by GPS-capable devices (e.g, smartphones, self-driving cars, navigation systems, trackers, etc.) has triggered a renewed interest on

[3] R-Boost library (www.boost.org) contains a state-of-the-art bulk loading algorithm for R-Trees that combines STR [32] and TGS [18].

Table 2: Notation

Symbol	Description
C	Maximum # of points per page
F	Maximum directory fanout
N	Data set cardinality
P	# of pages in the index
n	Node (data or directory)
N_n	# of points in the subtree of node n
P_n	# of pages in the subtree of node n
$\mathcal{S}_n$	Set of splits in node n
n^h	Node at height h

spatial indexing, especially towards *learned* structures that replace internal nodes with machine learning models (e.g., artificial neural networks), or utilize machine learning to enhance the index capabilities [29]. Several multi-dimensional learned indexes (e.g., Flood [36], Tsunami [13]) assume a known data distribution and query workload, or involve approximate querying (e.g., RSMI [40]). Moreover, they focus primarily on in-memory operations and lack in the spatial domain (e.g., not supporting kNN queries [36, 56]). Thus, they do not constitute competitors to the proposed Waffle, which is an exact disk-based index for general spatial queries, without requiring prior knowledge of the data or the queries. LISA [33] is, to the best of our knowledge, the only multi-dimensional learned index focusing on disk applications.

3 WAFFLE: BULK LOADING

Waffle is a disk-based spatial index that combines concepts from space and data partition methods. Specifically, the bulk loading and dynamic update algorithms apply an efficient sort-based scheme that generates non-overlapping nodes, with excellent load balance and without causing top-down propagation of splits. As in the case of data partition indexes, the nodes do not have to cover empty space. The spatial extent of each directory node is an MBR that contains the combined extent of its child nodes. Waffle involves two parameters: data node capacity (C), and directory fanout (F). C is the maximum number of data points per data node. F is the maximum number of child pointers per directory node. In this section we present the bulk loading algorithm of Waffle. Table 2 contains the common symbols used in the rest of the paper.

3.1 Algorithm

Let n^h be a node at level h that overflows ($h = 0$ corresponds to the leaf level). A split in Waffle is perpendicular to the split axis, which corresponds to the *major dimension* of n^h, i.e., the dimension along which its MBR stretches the most. Reducing node perimeter on the major dimension leads to square-like nodes that are easier to pack. Recall that Spread Split [16] also partitions the KDB-Tree nodes on the major dimension for the same reason, but since it always divides on the median, the data nodes are not necessarily packed to their full capacity. Instead, the split point of Waffle corresponds to the entry whose rank is a multiple of C and minimizes the cardinality gap between the nodes, i.e., the entry ranked $C \times \left\lfloor \frac{\lceil N_i/C \rceil}{2} \right\rfloor$ in the sorted list according to the coordinate of the major dimension. This ensures that the number of entries in at least one of the two new data nodes is a multiple of C. Another benefit of using multiples of C is that, when the median is not the last point in its data node, splitting requires dividing the page holding the median, increasing the total number of leaf nodes by one. Waffle, on the other hand, ensures that the number of data nodes remains stable. Furthermore, the two new subspaces after a split differ by at most one page.

Observe, that the major dimension of a node n^h in Waffle is obtained directly by its MBR, whereas Spread Split requires computing the distance between all pairs of points in n^h. Moreover, for intermediate nodes n^h ($h > 0$), median splitting algorithms, including Spread Split, (i) require finding the median of all points in n^h, and (ii) may lead to reorganization of the entire subtree of n^h through cascading splits. Waffle on the other hand, eliminates these problems, by utilizing the first split s_1 in n^h. The first split partitions n^h end to end on the major dimension, so that all subsequent splits and nodes lie entirely on either side of s_1. In addition to median search, this avoids cascading splits by preserving the existing subtree of n^h under the new nodes. To identify the split s_1 required for partitioning n^h, all splits that generate child nodes of n^h are maintained in chronological order in n^h.

Based on the above observations, Waffle bulk loads the data points using a two-step process. The first step creates the lowest level with the data nodes, assuming that initially there is a single data node n^0 containing all data points, and a root r^1 pointing to n^0. To partition n^0, we sort its points on the major dimension. After determining the split point s_1, entries up to s_1 in the sorted list are inserted to a new node n_1^0, and the rest to n_2^0. The MBRs of n_1^0 and n_2^0 are computed and inserted to r^1, together with pointers to the new nodes and s_1 (split axis, position), while n^0 is deleted. If either n_1^0 or n_2^0 contains more than C points, it is partitioned again. This recursive process continues until all data nodes hold at most C points and can fit on a disk page. The first step concludes with $\lceil N/C \rceil$ full nodes that contain exactly C points. If the cardinality N is not a multiple of C, the last data node has $N\%C$ points. The (possibly overflowing) root r^1 stores (i) pointers to all data nodes, (ii) their MBRs, and (iii) a vector $\mathcal{S}_{r^1}$ with the splits that generated these nodes in chronological order (i.e., $\mathcal{S}_{r^1}$ starts with s_1). Step 1 is summarized as Procedure *BulkLoad-I* in Algorithm 1.

The second step of bulk loading spans over multiple rounds. Starting bottom-up from level 1, each round splits an overflowing root, to create a new root at the next level. Let r^h denote the root at height h (initially, $h = 1$). A round begins by checking if r^h's fanout exceeds F, in which case r^h is split into two new directories n_1^h and n_2^h using the first split s_1 in $\mathcal{S}_{r^h}$. The contents of r^h (i.e., pointers to child nodes and splits), except s_1, are distributed to n_1^h and n_2^h, depending on which side of s_1 they lie. A new root r^{h+1} stores pointers to n_1^h, n_2^h and their MBRs; s_1 is inserted into the split vector $\mathcal{S}_{r^{h+1}}$ of r^{h+1}. If n_1^h and n_2^h's fanout exceeds F, they are partitioned using their respective first splits, maintained in $\mathcal{S}_{n_1^h}$ and $\mathcal{S}_{n_2^h}$. Split nodes are replaced in r^{h+1} with the new directories, and the splits that created them are kept in $\mathcal{S}_{r^{h+1}}$. The MBRs of the new directories are computed using the MBRs of their child nodes.

Algorithm 1 Procedures for the bulk loading algorithm

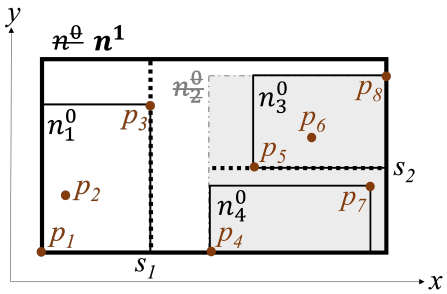
```
1:  procedure BulkLoad-I(DataNode n^0)
2:      Int splitPos ← C × ⌊⌈#Points in n^0 / C⌉ / 2⌋
3:      Split s ← getSplit(n^0, splitPos)
4:      Add s to S_{r^1}
5:      DataNodes {n_1^0, n_2^0} ← Partition(n^0, s)
6:      for all n ∈ {n_1^0, n_2^0} do
7:          if #Points in n > C then
8:              BulkLoad-I(n)
9:          else
10:             Add n to r^1
11:         end if
12:     end for
13: end procedure

14: procedure BulkLoad-II(Directory r^h)
15:     Split s ← S_{r^h}[0]
16:     Remove s from S_{r^h} and add to S_{r^{h+1}}
17:     Directories {n_1^h, n_2^h} ← Partition(r^h, s)
18:     for all n ∈ {n_1^h, n_2^h} do
19:         if #Nodes in n > F then
20:             BulkLoad-II(n)
21:         else
22:             Add n to r^{h+1}
23:         end if
24:     end for
25: end procedure
```

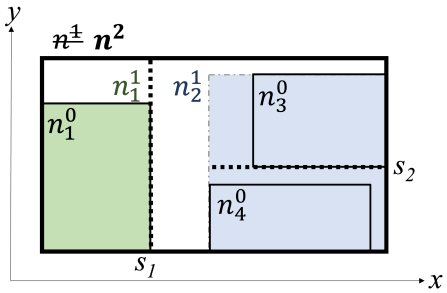
Directory splitting and populating r^{h+1} continues until the fanout of all directories at level h is at most F. If after the end of the round the fanout of r^{h+1} exceeds F, the process is repeated for r^{h+1}. This recursive procedure is summarized as *BulkLoad-II* in Algorithm 1. Note that Algorithm 1 presents a simplified version of the actual implementation. The procedures are implemented so that nodes do not need to know their parent directories.

For a given disk page size, the maximum fanout F is lower than the maximum number C of points per data node, because in addition to pointers, a directory stores the split vector and MBRs, which require two points (the opposite MBR corners) per entry. On the other hand, if the directories are kept in main memory, any value of F, including $F \geq C$, is applicable. For example, if we set $F = \lceil N/C \rceil$, all the data nodes created by step 1 can be accommodated under root r^1, and there is no second step during bulk loading. In this case, Waffle would degenerate to a non-uniform grid.

Figure 2 illustrates a complete example of bulk loading. Starting with Step 1 in Figure 2a, assume $N = 8$, $C = 3$, $F = 2$, a single data node n^0 holding all the points (i.e., p_1 to p_8), and the root node $\boldsymbol{n^1}$ pointing to n^0. $\boldsymbol{n^1}$ and n^0 have identical spatial extents (i.e., the MBR of the entire data set). Since n^0 overflows and is landscape oriented, the points are sorted on the x-axis (their ids refer to their order on x), and the split is vertical. Given $N = 8$ and $C = 3$, the first split point could pass through p_3 or p_6 because they are both multiples of the capacity $C = 3$, and yield the same cardinality difference (5-3) for the resulting nodes. Assuming that p_3 is selected, two new



(a) Step 1



(b) Step 2

Figure 2: Example of bulk loading

data nodes n_1^0 (storing points p_1 to p_3), and n_2^0 (with the rest) are created, while n^0 is discarded. Their spatial extents are tightened to the MBRs covering their data points. Pointers to the new data nodes, and s_1 are stored in $\boldsymbol{n^1}$. The second split occurs in n_2^0 since it exceeds the capacity $C = 3$. Because n_2^0 (visualized in gray) is longer on the y-axis, s_2 is drawn horizontally, and goes through p_5. Thus, n_2^0 is replaced by n_3^0 and n_4^0 each storing the points in their respective side of the split. Finally, n_2^0 is removed, while n_3^0, n_4^0 and s_2 are inserted to $\boldsymbol{n^1}$. Since none of the data nodes exceeds C, step 1 terminates.

Figure 2b presents the second step of bulk loading, where the root $\boldsymbol{n^1}$ overflows (it contains 3 child nodes, whereas the fanout is $F = 2$). $\boldsymbol{n^1}$ is partitioned on its first split s_1 (stored in step 1), creating two new directories n_1^1 and n_2^1. The new directories inherit the nodes and splits of $\boldsymbol{n^1}$ within their spatial extents (i.e, $\mathcal{S}_{n_1^1}=\{\}$, $\mathcal{S}_{n_2^1}=\{s_2\}$), except s_1. Note that to distribute splits to new directories, we also store their center point and not just their position along their split axis. For example, to determine that s_2 lies in n_2^1, we check if the x-coordinate of its center point is to the right of s_1. Similar to Step 1, the directory nodes' spatial extents are tightened to the MBRs covering their children. A new root $\boldsymbol{n^2}$ is created, which points to n_1^1, n_2^1, and stores s_1. Since there are no overflows, the process terminates.

3.2 Analysis

We analyze the I/O cost of bulk loading assuming that all directories reside in RAM and all data nodes on the disk. The data nodes are

formed in the first step. The second step creates directory nodes bottom up, without reading any data point or incurring top down tree reorganization. Thus, we need only consider the first step to estimate the total I/O cost. We have $P = \lceil N/C \rceil$ disk pages to pack. Given M pages of RAM, the I/O cost of *external sorting* P pages is $2P(\log_{M-1} PM^{-1} + 1)$. In each round, a subspace is sorted on its major dimension and divided into two equally paged subspaces (or differing by one if the number of pages is odd). Once each subspace has below M pages, it is further partitioned in main memory until reaching at most C points. Thus, out of a total of $\log_2 P$ sorting rounds, $\log_2 M$ rounds do not incur any I/O cost leaving $R = \log_2 P - \log_2 M = \log_2(PM^{-1})$ rounds to consider for the cost analysis. We begin with a single subspace and double the number of subspaces, each containing half the number of pages every round. After i rounds, there are 2^i subspaces with $P/2^i$ pages each. The I/O cost of the next round is then 2^i times the cost of sorting one such subspace, i.e. $2^i \times [2(P/2^i)(\log_{M-1}((P/2^i)M^{-1}) + 1)]$. When summed over R rounds, the total I/O cost is:

$$\begin{aligned}
&\sum_{i=0}^{R-1} 2^i \left(2(P/2^i)(\log_{M-1}((P/2^i)M^{-1}) + 1)\right) \\
&= 2P \sum_{i=0}^{R-1} \left(\log_{M-1}(P2^{-i}M^{-1}) + 1\right) \\
&= 2P \left(R \log_{M-1}(PM^{-1}) + R - \sum_{i=0}^{R-1} \log_{M-1} 2^i\right) \\
&= 2P \left(R \log_{M-1}(PM^{-1}) + R - \log_{M-1}(2^{\sum_{i=0}^{R-1} i})\right) \\
&= 2P \left(R \log_{M-1}(PM^{-1}) + R - \log_{M-1}(2^{R(R-1)/2})\right) \\
&= 2PR \left(\log_{M-1} \frac{PM^{-1}}{2^{(R-1)/2}} + 1\right)
\end{aligned}$$

Using the value of $R = \log_2(PM^{-1})$ and rearranging the terms we obtain:

$$\begin{aligned}
&P \log_2 PM^{-1}(\log_{M-1} PM^{-1} + 2) \\
&= \log_2 PM^{-1} \cdot (sort(N) + 2)/2 \implies R \cdot (sort(N) + 2)/2
\end{aligned} \tag{1}$$

where $sort(N)$ is the I/O cost of externally sorting N points. The above analysis concludes that bulk loading on average requires half the I/O cost of sorting all pages per round. Waffle guarantees $P = \lceil N/C \rceil$ data nodes (or pages) for data cardinality N and page capacity C, with zero overlap between nodes at all levels. The height of the index is $H = \log_F P$. The number of directory nodes is approximated as:

$$\begin{aligned}
P \sum_{i=1}^{H} F^{-i} &= P \cdot \frac{1 - F^{-H}}{F - 1} \\
&\approx P \cdot \frac{1 - P^{-1}}{F - 1} = \frac{P - 1}{F - 1}
\end{aligned} \tag{2}$$

Thus, the total number of nodes in the index is approximately $P + \frac{P-1}{F-1}$. Waffle also needs to store the splits. Since bulk loading starts with a single node containing all data points, and each split increases the number of data nodes by 1, the total number of splits

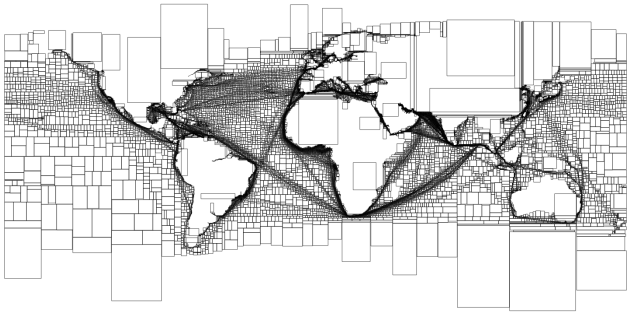


Figure 3: Waffle partition of the AIS data set

equals the number of data nodes minus 1: $P - 1$. A split consumes 9 bytes (2 floats for location, and 1 boolean for dimension), which is less than one-third of a node (1 pointer, 4 floats for MBR corners, and 1 integer for keeping the child node count). Using these statistics, the total size of the index can be estimated as:

$$\begin{aligned}
&= O\left(P + \frac{P}{3} + \frac{P-1}{F-1}\right) \\
&= O\left(\lceil N/C \rceil \cdot \left(\frac{4}{3} + \frac{1}{(F-1)}\right)\right)
\end{aligned} \tag{3}$$

For a fixed value of C, the value of F does not affect the index size significantly.

Figure 3 visualizes the data nodes of Waffle bulk loading. The experimental setup is identical to Figure 1. Waffle outperforms all bulk loading methods, achieving the optimal number of data nodes $\lceil 10^7/204 \rceil = 49020$ (i.e., every data node, except one, is full). Although R-Boost also achieves the same number of data nodes, those of Waffle are highly square-shaped, with an average perimeter of 1.68 comparing to 3.00 by R-Boost.

4 DYNAMIC UPDATES

For disk-based indexes, the CPU-time of common queries is usually negligible compared to the disk I/O cost. Accordingly they prioritize reducing the node accesses over optimizing directory traversal. For instance, during overflows, R*-Trees reinsert a subset of data points to improve the index structure and enhance the efficiency of future queries. However, frequent reinsertions may lead to slow updates on dynamic data sets. Also, a compact structure that maximizes dead space is counter effective to update-heavy workloads as nodes are expanded more often to fit the new data points. This leads to a vicious cycle of contractions (through reinsertions) and expansions especially when performing burst updates. Expansions in R*-Trees may also result in node overlaps, some of which persist after contractions and degrade query performance.

In contrast, Waffle delays index optimizations until required by the query workload. Specifically, nodes that are often accessed by queries are re-organized more frequently than those that are prone to more updates. Consequently, different parts of the tree are maintained according to the workload in the corresponding part of the data space. Section 4.1 introduces update algorithms that guarantee non-overlapping nodes, and Section 4.2 describes the maintenance procedures.

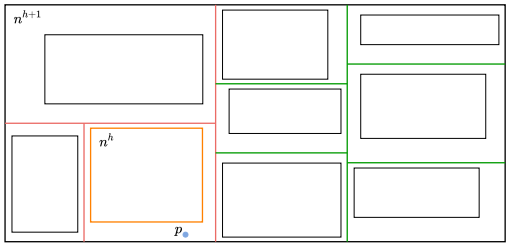


Figure 4: Virtual container

4.1 Insertions and Deletions

Inserting a point p to the index involves three stages.

Stage 1. Starting from the root, perform a depth-first traversal, visiting, at each level, the node that contains p. Consider the case where no such node exists at some level h, i.e. p lies in dead space. Still, p lies on one side of each split stored in n^{h+1} (the directory being visited), so we compute the intersection of all such half-spaces and obtain a virtual container over p, as shown in Figure 4. Next, we find the child node n^h that lies in this container, expand it to fit the point and continue traversing downwards. Note that since all nodes are formed and separated by splits, there always exists a unique node that can be expanded to fit the data point without overlapping any other node. Let n^0 be the data node found at the leaf level. If inserting p to n^0 does not lead to page overflow (i.e., $N_{n^0} \leq C$), finish insertion. Otherwise, proceed to stage 2.

Stage 2: Split n^0 on the median of its major dimension, delete n^0, and add the two new data nodes to n^1. Append the new split to $\mathcal{S}_{n^1}$. Finally, check if n^1 overflows, in which case proceed to Stage 3.

Stage 3: Retrieve the first split s_1 from S_{n^1}, and use it to partition n^1 into two new directories. Distribute n^1's contents, except for s_1, to the new directories depending on which side of the split they lie. Replace n^1 with the new directories and add s_1 to n^2. If n^2 overflows, repeat Stage 3 for n^2.

Procedures in stages 2 and 3 are summarized in Algorithm 2. The first split achieves a partition that is (i) near the median, (ii) cuts the subspace end to end, and (iii) intersects with zero nodes in the subtree, eliminating cascading splits and simplifying subtree partitioning. While R*-Trees involve quadratic updating procedures, Waffle employs a linear approach to find the partition that guarantees zero node overlaps.

For deletions, a point query returns the data node n^0 covering the point p to be deleted. Node n^0 is scanned and p is deleted, potentially shrinking its MBR or deleting the node if empty. Let n^1 be the parent directory of n^0. When n^1 becomes empty, it is deleted, and this action is propagated upwards. During propagation, if p lies on the periphery of the current node's MBR, its area is tightened accordingly. Waffle does not involve an explicit minimum node capacity because underflows are handled by maintenance procedures described next.

4.2 Maintenance

After splits or deletions, nodes may have relatively few children, leading to a large number of half-full nodes. However, for queries with large output (e.g., ranges), the data nodes accessed are proportional to the query selectivity. For instance, assuming uniform distribution, a range that retrieves $x\%$ of the points is expected to access about $x\%$ of the data nodes. Continuing the example, according to the number of data nodes in Table 1 (and as verified in the experiments), the bulk loaded Quad-Tree should be much slower than R-Boost since it has more than 2.5 times the number of data nodes. Although such problems could be solved by some type of node packing, re-organizing the entire index is time consuming, and maybe redundant in the presence of update-heavy workloads. Motivated by these observations, Waffle performs local repacking, aimed at data node minimization, based on the ratio of queries and updates in the corresponding part of the data space.

Algorithm 2 Procedures for the insertion algorithm

```
1:  procedure SplitDataNode(Directory n^1, DataNode n^0)
2:      Sort N_{n^0} along the major dimension
3:      Int splitPos ← N_{n^0}/2
4:      Split s ← getSplit(n^0, splitPos)
5:      Add s to S_{n^1}
6:      DataNodes {n_1^0, n_2^0} ← Partition(n^0, s)
7:      Remove n^0 and add n_1^0, n_2^0 to n^1
8:  end procedure

9:  procedure SplitDirectory(Directory n^{h+1}, Directory n^h)
10:     Split s ← S_{n^h}[0]
11:     Remove s from S_{n^h} and add to S_{n^{h+1}}
12:     Directories {n_1^h, n_2^h} ← Partition(n^h, s)
13:     Remove n^h and add n_1^h, n_2^h to n^{h+1}
14: end procedure
```

For each directory n, we use (i) $fat(n)$ to measure the deterioration of n based on the number of data nodes it contains compared to the optimal number, and (ii) $tolerance(n)$ to define a limit for the deterioration based on the update/query ratio. Let P_n and N_n be the number of data nodes (pages) and points under n, respectively. Given capacity C, the optimal number of pages under n is $\lceil N_n/C \rceil$ ($P_n \geq \lceil N_n/C \rceil$). The *fat* of directory n is computed as:

$$\textbf{fat}(n) = \frac{P_n}{\lceil N_n/C \rceil} - 1 \tag{4}$$

Intuitively, the existence of fat implies that the directory contains more nodes than necessary to accommodate the entries in its subtree. Waffle monitors the fat levels of directories and maintains for each n, a read (R_n) and write counter (W_n), which are updated every time a read or write operation is performed in that directory, respectively. The tolerance is computed as the ratio:

$$\textbf{tolerance}(n) = \frac{W_n}{R_n} \tag{5}$$

When $fat(n)$ exceeds $tolerance(n)$, directory n is repacked. Repacking involves locally bulk loading the subtree of n using the algorithm of Section 3.1 that ensures zero fat at all levels in the subtree, while also restoring the square-like shape of nodes. In this way, frequently queried directories, are timely repacked to preserve the low I/O cost

of queries. On the other hand, nodes that have relatively more updates, compared to queries, provide leeway to postpone repacking, thereby enabling faster update operations.

We also experimented with more complex formulae, taking also into account the timing of the operations (i.e., recent operations are considered more important). However, they do not improve performance significantly, while they impose additional (space/time) complexity. Moreover, although repacking can be performed at all levels, Waffle restricts it at height 1 because maintenance at higher levels neutralizes the fine-grained control of lower levels. As a result, directories at the lowest level in Waffle automatically and independently adjust to any workload in their spatial domain. This enables Waffle to handle a variety of workloads over different regions simultaneously. Finally, note that the maintenance mechanism implicitly distinguishes the query types. For instance, a workload of point and kNN queries, where all results are usually found within 1-2 data nodes, triggers fewer repacking operations than a workload of ranges, even if the query/update ratio is the same in both workloads. This is because each range query is likely to access (and increase the read counters of) more data nodes, decreasing their tolerance.

5 QUERY PROCESSING

Similar to other spatial indexes, range and point queries are processed using depth-first traversal. Specifically, processing starts from the root, and recursively visits each node that overlaps with the query point or range. When the search hits a data node, the corresponding page is scanned and qualifying points are appended to the output. Depth-first search can also be applied to other spatial queries, by adopting existing MBR-based algorithms, mostly developed for R-Trees. Consider for instance a *spatial join*: given point sets A and B, retrieve all pairs $(a, b), a \in A, b \in B$, such that a and b are within distance d from each other. Assuming that A and B are indexed by two Waffle indexes, the qualifying pairs can be retrieved by traversing the two indexes in parallel, and recursively visiting pairs of nodes whose minimum distance is at most d [7]. The optimizations of [7], originally proposed for R-Trees, are also applicable to Waffle.

Nearest neighbor queries follow best-first traversal [20, 22]. A max-heap stores the k nearest points retrieved so far, while a min-heap maintains the nodes to be visited. In both cases, the key is the distance between the query location and a data point (in the max-heap) or a node (in the min-heap). Processing starts at the root, and all its child nodes are inserted to the min-heap. The top node of the min-heap (i.e., the one with the minimum distance to the query) is visited, and its contents are also inserted in the min-heap. The process is repeated until the first k candidate nearest points are found in some data node. The kth (farthest) candidate resides at the top the max-heap. After this step, only nodes, whose minimum distance is below that of the current kth NN are visited. The process terminates when the key at the top of the min-heap exceeds the distance of the kth NN, because unvisited nodes cannot contain data points that are closer to the query.

Similar best-first traversal techniques are applicable to related queries, such as *k closest pairs* [12]: given point sets A and B, retrieve the k pairs $(a, b), a \in A, b \in B$ with the minimum distance from each other. Assuming that A and B are indexed by two Waffle indexes, they are traversed synchronously top down, and pairs of nodes are inserted in a min-heap according to their minimum distance. Candidate pairs of points are inserted in a max-heap. Search terminates when the top of the max-heap (i.e., distance of the kth pair) is below the top of the min heap (i.e., minimum distance between all unvisited pairs of nodes). Other algorithms based on best-search, originally proposed for R-Trees, that can be easily adapted to Waffle include, *Voronoi-based kNN*[28], *reverse nearest neighbors* [14, 30, 31, 53] and *skylines* [39, 50].

6 EXPERIMENTAL EVALUATION

Our experiments were performed using an AMD Ryzen Threadripper 3960X 3.8GH CPU with 64GB RAM and 64-bit Ubuntu Linux operating system. We compare Waffle with the following indexes:

(1) **R-Boost**: We use the state-of-the-art implementation of the R-Tree from boost.org, which includes (i) a bulk loading algorithm that combines STR [32] and TGS [18], and (ii) the R*-Tree [3] algorithms for dynamic insertions.
(2) **KDB-Tree**: We implemented a highly optimized version of the KDB-Tree, which utilizes (i) Spread Split [16] to select the spit axis (and is vastly superior to the original cyclic split of KDB-Trees), and (ii) the bulk loading algorithm presented in Section 2 that avoids cascading splits.
(3) **Quad-Tree**: We implemented the PR-Quad-Tree and Optimized Point-Quad-Tree. The PR-Quad-Tree has poor performance and is excluded from the diagrams. We use the shorthand Quad-Tree to denote the Optimized Point-Quad-Tree [15].

All implementations are in C++, and executed without multi-threading. We assume that the directories are kept in RAM, and only the data nodes are disk-resident. The data node capacity (C) is set to 204 data points corresponding to 4KB page size. R-Boost requires the fanout (F) to be the same as data node capacity, so we set $F = 204$ for all applicable indexes. Unfortunately, we could not include LISA [33] in our evaluation since the code provided by the authors[4] is in Python and slow compared to the C++ implementations of the other indexes. Moreover, learned indexes require training and their effectiveness to highly dynamic data is questionable. We use the following real data sets in our evaluation:

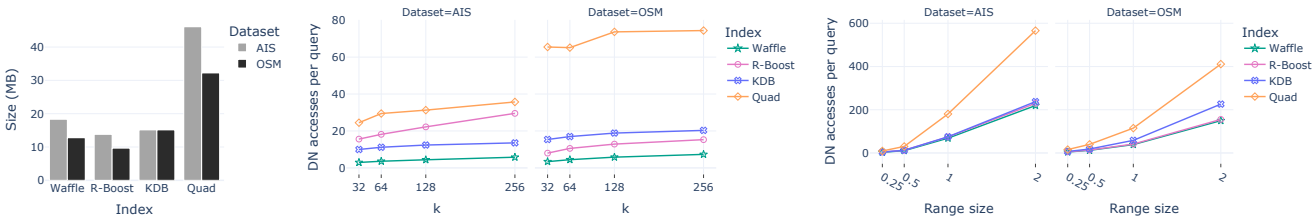
(1) **AIS**: 100 million records of ship coordinates around the planet. It is privately donated by a shipping company.
(2) **OSM** [38]: 70 million geolocations in mainland USA.

In addition to insertion operations, our evaluation focuses on range and kNN queries. A range is a square that covers a percentage r per axis. The values used are 0.25%, 0.5%, 1%, and 2% (e.g., $r = 1\%$ defines a square that covers 10^{-4} of the 2D data space). For kNN queries, k is a power of 2 ranging between 32 and 256.

6.1 Bulk Loading

First, we bulk load AIS and OSM, and measure the index size in Figure 5a. Recall that R-Boost and Waffle guarantee the minimal number of data nodes that are fully filled after bulk loading. For the KDB and Quad-Tree, the number of data nodes depends on the data set size and page capacity. For instance, although AIS and

[4] https://github.com/pfl-cs/LISA



(a) Size **(b) kNN Queries** **(c) Range Queries**

Figure 5: Bulk loading experiments

OSM have different cardinality, $C = 204$ yields the same number of data nodes in the KDB-Tree. Waffle is slightly larger than R-Boost mainly due to the additional split vectors per directory node. Waffle also stores the read and write counters in level 1 directories, but this space overhead is negligible compared to the split vectors.

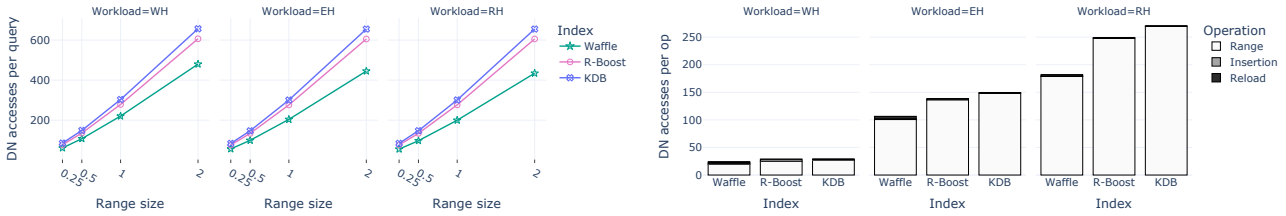
Next we perform 1000 kNN queries uniformly spread across the data set's spatial extent, and report the number of data node accesses per query in Figure 5b. According to Figure 1, AIS is highly skewed towards the water bodies and produces many spaghetti-like nodes for R-Boost, leading to poor kNN performance. On the other hand, OSM is less skewed, and R-Boost outperforms the KDB-Tree for this data set. Waffle, owing to its square-like nodes is at least two times faster on kNN queries than the nearest competitor for both data sets. The Quad-Tree is the slowest index. Figure 5c illustrates the result of a similar experiment with 1000 range queries. As discussed in Section 4.2, a small number of data nodes benefits the performance of queries with large output. Thus, while the KDB-Tree is competitive to Waffle and R-Boost in AIS, it is costlier for OSM. Waffle and R-Boost also benefit by queries that fall into dead space, costing few to zero data node accesses. Because Quad-Trees have the worst performance in all settings, usually by a wide margin, they are omitted from the following diagrams.

6.2 Dynamic Data

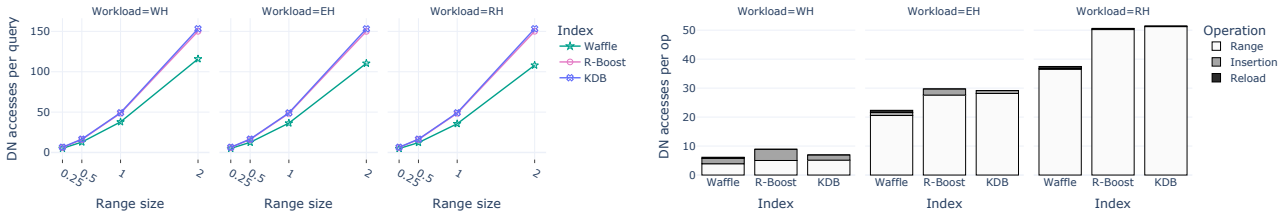
The following experiments are identified by the workload (ratio of insertions over queries), distribution (temporal and spatial distribution of queries), and query type (range or kNN). We consider three types of workloads, namely *write-heavy* (WH, 10 insertions for every query), *equal-heavy* (EH, equal number of insertions and queries) and *read-heavy* (RH, 10 queries for every insertion). We assume three distribution scenarios. (i) Spatial distortion, where the order of insertions and queries is random, and 90% of the queries are concentrated in three regions (30% each), and the remaining 10% in the rest of the data set. For AIS, these regions are the Gulf of Mexico, Mediterranean Sea, and Arabian Sea; for OSM, they are centered around Los Angeles, Chicago, and New York City. (ii) Temporal distortion, where the queries are uniform in the data space, and insertions and queries occur in cycles. In the first and third cycle, 20% of the insertions are followed by 20% of the queries. In the second and fourth cycle, 30% of the insertions are followed by 30% of the queries. (iii) Spatio-temporal distortion, where the queries are both spatially and temporally distorted. For all three cases, insertions are uniformly distributed. The values of r for ranges, and k for kNN queries, are the same as the bulk loading experiments. Each value is used with the same probability. In summary, we consider eighteen combinations of three workloads, three distribution, and two query types.

To best simulate a real-life scenario, the experimental procedure is divided into two steps. The first step stabilizes the indexes based on the experiment profile. Since KDB-Trees and R-Trees cannot adjust to workload or distribution, we simply insert 10 million data points as part of their preparation. On the other hand, for Waffle, we insert 10 million data points and execute queries, according to the experiment profile. For example, to prepare Waffle for a write-heavy workload with temporal distortion, 10 million insertions and 1 million ranges are performed in four cycles. In the second step, 1 million further insertions and the corresponding number of queries are performed for all indexes as per the profile. Index performance is evaluated in the second step. For each experiment, we display two plots. The first measures the node accesses per query, while the second plot considers all operations (queries, insertions, and maintenance) executed during the experiment and highlights the difference in overall performance of the indexes. Note that the reload cost is specific to Waffle, and represents the data node accesses for maintaining the index. For R-Boost, the maintenance (re-insertions) is part of the insertion cost.

We first evaluate the spatial distortion case, where most queries are concentrated on three regions of the data sets, and the order of insertions and queries is random. Figure 6 illustrates the query and overall performance for workloads with range queries. Ranges in dense regions require numerous data node accesses that dominate insertion and maintenance cost in all workloads. Thus, the diagrams for overall performance follow those for range query cost. Due to the small number of square-like data nodes Waffle has up to 40% better performance than R-Boost, which in turn slightly outperforms the KDB-Tree. Figure 7 shows the results for kNN queries in the same setting. Waffle again has the best query performance, achieving up to 30% savings with respect to the next competitor, which is the KDB-Tree for AIS, and R-Boost for OSM. Due to the skeweness of AIS, R-Boost suffers from overlapping nodes in dense clusters. Since kNN queries require few data node accesses compared to ranges, insertions and maintenance contribute significantly to the overall cost for the WH and EH workloads. Thus, despite better query performance than the KDB-Tree in OSM, R-Boost requires the most node accesses per operation for these workloads. Because optimizing query performance is not priority for update intensive workloads, Waffle regulates its maintenance procedures accordingly, focusing only on regions with high query rates. Thus,



(a) AIS



(b) OSM

Figure 6: Range queries in spatial distortion

(a) AIS

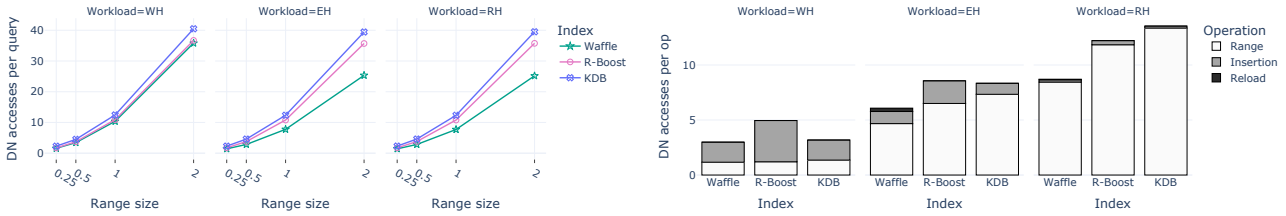
(b) OSM

Figure 7: kNN queries in spatial distortion

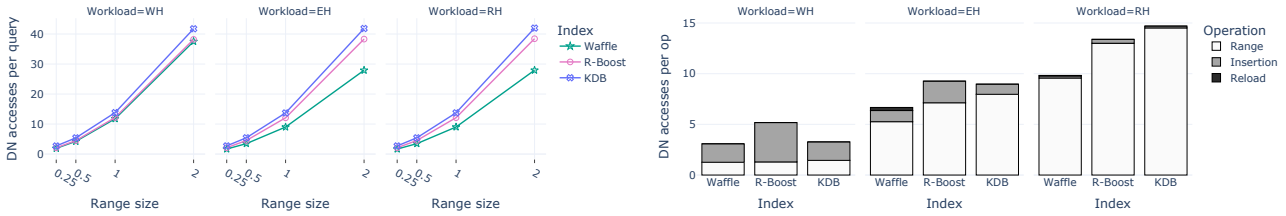
it generates a very compact structure in some parts of the index, while keeping the rest flexible for updates.

Figure 8 evaluates range query and overall performance in temporal distortion, where ranges are uniform, but the insertions and the queries are executed in batches. The relative performance of the indexes is similar to the spatial distortion case. However, compared to Figure 6, since the range queries are no longer concentrated in dense regions, the number of node accesses drops, increasing the relevance of maintenance costs. Therefore, although the KDB-Tree has the worst range performance, it beats R-Boost overall for write- and equal-heavy workloads due to the lower maintenance cost. Figure 9 contains the plots for kNN queries in these settings. Observe that kNN queries are several times more expensive than spatial distortion, since they may fall in sparse regions of the data space. Maintenance procedures in R-Boost and Waffle ensure that sparse regions are covered using the minimal number of nodes, which enables fast retrieval of neighbors. In the absence of such functionality, the KDB-Tree performs poorly for all workloads. However, for the WH workload, R-Boost has about double the insertion cost of Waffle and KDB-Tree in both range and kNN experiments. Thus, while it performs well on queries, it suffers from unnecessary optimizations during insertions. On the other hand, while the KDB-Tree has fast maintenance, its query performance is often very slow. Waffle benefits from on-demand and region specific reloading, achieving the highest efficiency in all cases.

Figure 10 compares index performance under spatio-temporal distortion, where most range queries focus on dense regions, and operations are executed in batches. Similar to the spatial distortion case, ranges dominate the overall cost in RH and EH workloads. Waffle has again the best overall performance, followed by R-Boost,

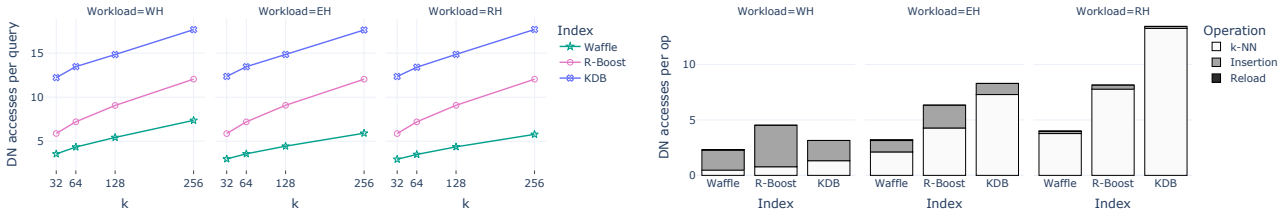


(a) AIS

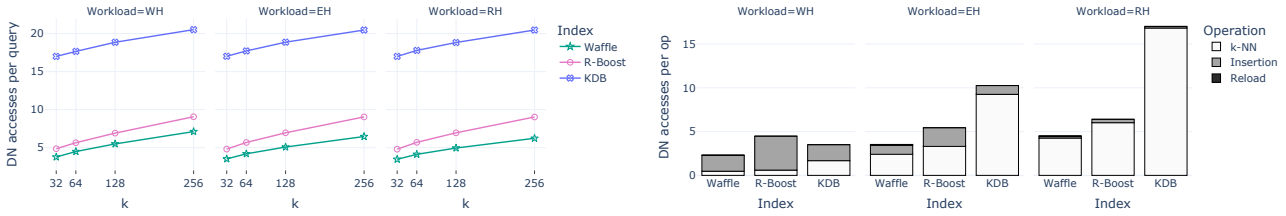


(b) OSM

Figure 8: Range queries in temporal distortion



(a) AIS



(b) OSM

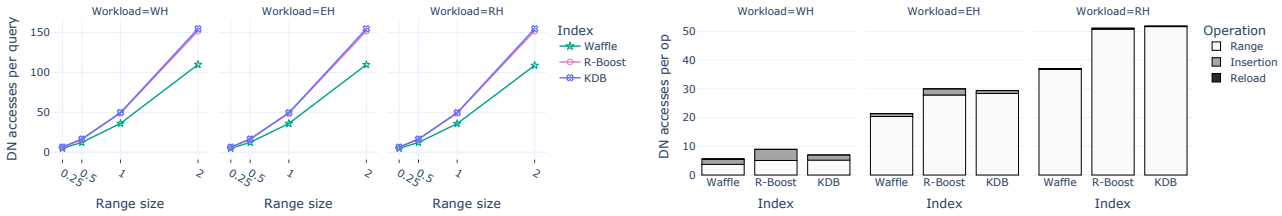
Figure 9: kNN queries in temporal distortion

except for the WH workload, where the KDB-Tree outperforms R-Boost. The diagrams for kNN queries in Figure 11 are also similar to Figure 7, where R-Boost is the slowest index for AIS. Even with the improved kNN performance in OSM, it falls behind the KDB-Tree in WH and EH workloads due to high insertion cost. Observe that contrary to spatial distortion, Waffle incurs negligible maintenance cost in all workloads because reloading occurs only when executing query cycles.

Finally, Figure 12 illustrates the index size for the nine workload/distribution combinations on range queries. Note that R-Boost and the KDB-Tree are insensitive to those parameters, and their structure is fixed. Waffle consumes the least space for RH workloads, which incur frequent reloading. Consequently, it maintains a compact structure that compensates for the additional split vectors. On the other hand, Waffle is the largest index for WH workloads, where repacking is infrequent. However, this is a desirable trade-off as discussed in the previous experiments.

To summarize the evaluation, the Quad-Tree is consistently the worst index for both kNN and range queries. The KDB-Tree does not involve maintenance algorithms and exhibits rather poor query performance for the majority of experiments with dynamic data sets. Unlike data-partitioning indexes such as R-Boost and Waffle, it receives little benefit when range queries are in dead space. For R-Boost, bulk loading and insertion procedures generate spaghetti-like and overlapping nodes in dense regions of skewed data sets, increasing the cost of kNN queries. Moreover, expensive insertions render R-Boost ineffective for write-heavy workloads. On the other hand, square-like and non-overlapping nodes enable the best query performance for Waffle. Coupled with low-cost maintenance, Waffle prevails as the superior index.

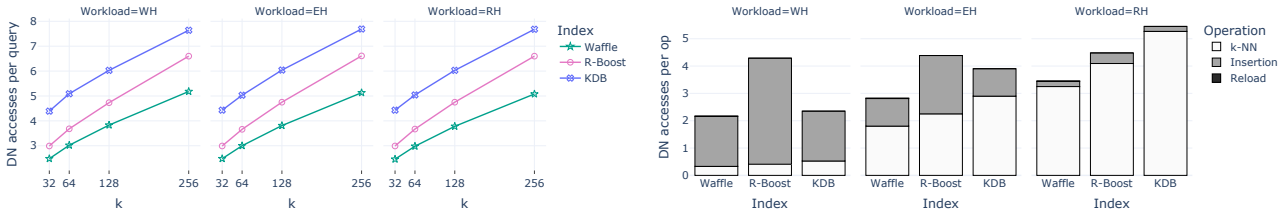
(a) AIS



(b) OSM

Figure 10: Range queries in spatio-temporal distortion

(a) AIS



(b) OSM

Figure 11: kNN queries in spatio-temporal distortion

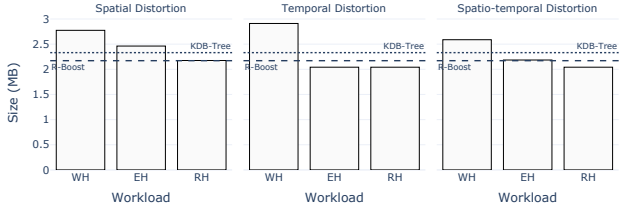


Figure 12: Index sizes for range queries on AIS

7 CONCLUSION

This paper proposes Waffle, a disk-based framework that aims to optimize performance for all spatial indexing purposes. For static data, Waffle achieves the minimum number of data nodes through a fast bulk loading algorithm that avoids cascading splits. The output node extents have a desirable (i.e., square-like) shape that minimizes their average perimeter. For dynamic data sets, it automatically adjusts the balance between node packing and update speed. Depending on the setup, local bulk loading reorganizes subtrees of the index that deteriorate after updates. Whereas competitive spatial indexes, such as the KDB-Tree and the R*-Tree, fail in some settings, our experimental evaluation reveals that Waffle delivers excellent query and update performance in all settings, and for both static and dynamic data.

ACKNOWLEDGMENTS

This work was supported by GRF grants HKUST 16204119 and HKUST 16206220 from Hong Kong RGC.

REFERENCES

[1] Afsin Akdogan, Saratchandra Indrakanti, Ugur Demiryurek, and Cyrus Shahabi. 2015. Cost-efficient partitioning of spatial data on cloud. In *2015 IEEE International Conference on Big Data, Big Data 2015, Santa Clara, CA, USA, October 29 - November 1, 2015*. IEEE Computer Society, 501–506. https://doi.org/10.1109/BigData.2015.7363792

[2] Lars Arge, Mark de Berg, Herman J. Haverkort, and Ke Yi. 2004. The Priority R-Tree: A Practically Efficient and Worst-Case Optimal R-Tree. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004*, Gerhard Weikum, Arnd Christian König, and Stefan Deßloch (Eds.). ACM, 347–358. https://doi.org/10.1145/1007568.1007608

[3] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. 1990. The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, USA, May 23-25, 1990*, Hector Garcia-Molina and H. V. Jagadish (Eds.). ACM Press, 322–331. https://doi.org/10.1145/93597.98741

[4] Norbert Beckmann and Bernhard Seeger. 2009. A revised r*-tree in comparison with related index structures. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*, Ugur Çetintemel, Stanley B. Zdonik, Donald Kossmann, and Nesime Tatbul (Eds.). ACM, 799–812. https://doi.org/10.1145/1559845.1559929

[5] Jon Louis Bentley. 1975. Multidimensional Binary Search Trees Used for Associative Searching. *Commun. ACM* 18, 9 (1975), 509–517. https://doi.org/10.1145/361002.361007

[6] Jon Louis Bentley and Jerome H. Friedman. 1979. Data Structures for Range Searching. *ACM Comput. Surv.* 11, 4 (1979), 397–409. https://doi.org/10.1145/356789.356797

[7] Thomas Brinkhoff, Hans-Peter Kriegel, and Bernhard Seeger. 1993. Efficient Processing of Spatial Joins Using R-Trees. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, DC, USA, May 26-28, 1993*, Peter Buneman and Sushil Jajodia (Eds.). ACM Press, 237–246. https://doi.org/10.1145/170035.170075

[8] Kaushik Chakrabarti and Sharad Mehrotra. 1999. The Hybrid Tree: An Index Structure for High Dimensional Feature Spaces. In *Proceedings of the 15th International Conference on Data Engineering, Sydney, Australia, March 23-26, 1999*, Masaru Kitsuregawa, Michael P. Papazoglou, and Calton Pu (Eds.). IEEE Computer Society, 440–447. https://doi.org/10.1109/ICDE.1999.754960

[9] Jidong Chen and Xiaofeng Meng. 2006. Update-efficient indexing of moving objects in road networks. *GeoInformatica* 13 (2006), 397–424.

[10] Su Chen, Christian S. Jensen, and Dan Lin. 2008. A benchmark for evaluating moving object indexes. *Proc. VLDB Endow.* 1 (2008), 1574–1585.

[11] Paolo Ciaccia, Marco Patella, and Pavel Zezula. 1997. M-tree: An Efficient Access Method for Similarity Search in Metric Spaces. In *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*, Matthias Jarke, Michael J. Carey, Klaus R. Dittrich, Frederick H. Lochovsky, Pericles Loucopoulos, and Manfred A. Jeusfeld (Eds.). Morgan Kaufmann, 426–435. http://www.vldb.org/conf/1997/P426.PDF

[12] Antonio Corral, Yannis Manolopoulos, Yannis Theodoridis, and Michael Vassilakopoulos. 2000. Closest Pair Queries in Spatial Databases. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*, Weidong Chen, Jeffrey F. Naughton, and Philip A. Bernstein (Eds.). ACM, 189–200. https://doi.org/10.1145/342009.335414

[13] 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. https://doi.org/10.14778/3425879.3425880

[14] Tobias Emrich, Hans-Peter Kriegel, Peer Kröger, Johannes Niedermayer, Matthias Renz, and Andreas Züfle. 2013. Reverse-k-Nearest-Neighbor Join Processing. In *Advances in Spatial and Temporal Databases*, David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Mario A. Nascimento, Timos Sellis, Reynold Cheng, Jörg Sander, Yu Zheng, Hans-Peter Kriegel, Matthias Renz, and Christian Sengstock (Eds.). Vol. 8098. Springer Berlin Heidelberg, Berlin, Heidelberg, 277–294. https://doi.org/10.1007/978-3-642-40235-7_16 Series Title: Lecture Notes in Computer Science.

[15] Raphael A. Finkel and Jon Louis Bentley. 1974. Quad Trees: A Data Structure for Retrieval on Composite Keys. *Acta Informatica* 4 (1974), 1–9. https://doi.org/10.1007/BF00288933

[16] Jerome H. Friedman, Jon Louis Bentley, and Raphael A. Finkel. 1977. An Algorithm for Finding Best Matches in Logarithmic Expected Time. *ACM Trans. Math. Softw.* 3, 3 (1977), 209–226. https://doi.org/10.1145/355744.355745

[17] Volker Gaede and Oliver Günther. 1998. Multidimensional Access Methods. *ACM Comput. Surv.* 30, 2 (1998), 170–231. https://doi.org/10.1145/280277.280279

[18] Yván J. García, Mario Alberto López, and Scott T. Leutenegger. 1998. A Greedy Algorithm for Bulk Loading R-Trees. In *ACM-GIS '98, Proceedings of the 6th international symposium on Advances in Geographic Information Systems, November 6-7, 1998, Washington, DC, USA*, Robert Laurini, Kia Makki, and Niki Pissinou (Eds.). ACM, 163–164. https://doi.org/10.1145/288692.288723

[19] Antonin Guttman. 1984. R-Trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, USA, June 18-21, 1984*, Beatrice Yormark (Ed.). ACM Press, 47–57. https://doi.org/10.1145/602259.602266

[20] Andreas Henrich. 1994. A Distance Scan Algorithm for Spatial Access Structures. In *Proceedings of the Second ACM Workshop on Advances in Geographic Information Systems, ACM-GIS 1994, Gaithersburg, MD, USA*, Niki Pissinou and Kia Makki (Eds.). ACM, 136–143.

[21] Andreas Henrich, Hans-Werner Six, and Peter Widmayer. 1989. The LSD tree: Spatial Access to Multidimensional Point and Nonpoint Objects. In *Proceedings of the Fifteenth International Conference on Very Large Data Bases, August 22-25, 1989, Amsterdam, The Netherlands*, Peter M. G. Apers and Gio Wiederhold (Eds.). Morgan Kaufmann, 45–53. http://www.vldb.org/conf/1989/P045.PDF

[22] Gísli R. Hjaltason and Hanan Samet. 1999. Distance Browsing in Spatial Databases. *ACM Trans. Database Syst.* 24, 2 (1999), 265–318. https://doi.org/10.1145/320248.320255

[23] Dmitri V. Kalashnikov, Sunil Prabhakar, and Susanne E. Hambrusch. 2004. Main Memory Evaluation of Monitoring Queries Over Moving Objects. *Distributed Parallel Databases* 15, 2 (2004), 117–135. https://doi.org/10.1023/B:DAPD.0000013068.25976.88

[24] Ibrahim Kamel and Christos Faloutsos. 1993. On Packing R-trees. In *CIKM 93, Proceedings of the Second International Conference on Information and Knowledge Management, Washington, DC, USA, November 1-5, 1993*, Bharat K. Bhargava, Timothy W. Finin, and Yelena Yesha (Eds.). ACM, 490–499. https://doi.org/10.1145/170088.170403

[25] Ibrahim Kamel and Christos Faloutsos. 1994. Hilbert R-tree: An Improved R-tree using Fractals. In *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo (Eds.). Morgan Kaufmann, 500–509. http://www.vldb.org/conf/1994/P500.PDF

[26] Norio Katayama and Shin'ichi Satoh. 1997. The SR-tree: An Index Structure for High-Dimensional Nearest Neighbor Queries. In *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA*, Joan Peckham (Ed.). ACM Press, 369–380. https://doi.org/10.1145/253260.253347

[27] Kihong Kim, Sang Kyun Cha, and Keunjoo Kwon. 2001. Optimizing Multidimensional Index Trees for Main Memory Access. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data, Santa Barbara, CA, USA, May 21-24, 2001*, Sharad Mehrotra and Timos K. Sellis (Eds.). ACM, 139–150. https://doi.org/10.1145/375663.375679

[28] Mohammad Kolahdouzan and Cyrus Shahabi. 2004. Voronoi-Based K Nearest Neighbor Search for Spatial Network Databases. In *Proceedings 2004 VLDB Conference*. Elsevier, 840–851. https://doi.org/10.1016/B978-012088469-8.50074-7

[29] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 489–504. https://doi.org/10.1145/3183713.3196909

[30] Hans-Peter Kriegel, Peer Kröger, Matthias Renz, Andreas Züfle, and Alexander Katzdobler. 2009. Incremental Reverse Nearest Neighbor Ranking. In *2009 IEEE 25th International Conference on Data Engineering*. IEEE, Shanghai, China, 1560–1567. https://doi.org/10.1109/ICDE.2009.144 ISSN: 1084-4627.

[31] Hans-Peter Kriegel, Peer Kröger, Matthias Renz, Andreas Züfle, and Alexander Katzdobler. 2009. Reverse k-Nearest Neighbor Search Based on Aggregate Point Access Methods. In *Scientific and Statistical Database Management*, Marianne Winslett (Ed.). Vol. 5566. Springer Berlin Heidelberg, Berlin, Heidelberg, 444–460. https://doi.org/10.1007/978-3-642-02279-1_32 Series Title: Lecture Notes in Computer Science.

[32] Scott T. Leutenegger, J. M. Edgington, and Mario Alberto López. 1997. STR: A Simple and Efficient Algorithm for R-Tree Packing. In *Proceedings of the Thirteenth International Conference on Data Engineering, April 7-11, 1997, Birmingham, UK*, W. A. Gray and Per-Åke Larson (Eds.). IEEE Computer Society, 497–506. https://doi.org/10.1109/ICDE.1997.582015

[33] Pengfei Li, Hua Lu, Qian Zheng, Long Yang, and Gang Pan. 2020. LISA: A Learned Index Structure for Spatial Data. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 2119–2133. https://doi.org/10.1145/3318464.3389703

[34] David B. Lomet and Betty Salzberg. 1990. The hB-Tree: A Multiattribute Indexing Method with Good Guaranteed Performance. *ACM Trans. Database Syst.* 15, 4 (1990), 625–658. https://doi.org/10.1145/99935.99949

[35] Kyriakos Mouratidis, Marios Hadjieleftheriou, and Dimitris Papadias. 2005. Conceptual Partitioning: An Efficient Method for Continuous Nearest Neighbor Monitoring. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005*, Fatma Özcan (Ed.). ACM, 634–645. https://doi.org/10.1145/1066157.1066230

[36] Vikram Nathan, Jialin Ding, Mohammad Alizadeh, and Tim Kraska. 2020. Learning Multi-Dimensional Indexes. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 985–1000. https://doi.org/10.1145/3318464.3380579

[37] Matthaios Olma, Farhan Tauheed, Thomas Heinis, and Anastasia Ailamaki. 2017. BLOCK: Efficient Execution of Spatial Range Queries in Main-Memory. In *Proceedings of the 29th International Conference on Scientific and Statistical Database Management, Chicago, IL, USA, June 27-29, 2017*. ACM, 15:1–15:12. https://doi.org/10.1145/3085504.3085519

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

[39] Dimitris Papadias, Yufei Tao, Greg Fu, and Bernhard Seeger. 2005. Progressive skyline computation in database systems. *ACM Trans. Database Syst.* 30, 1 (2005), 41–82. https://doi.org/10.1145/1061318.1061320

[40] Jianzhong Qi, Guanli Liu, Christian S. Jensen, and Lars Kulik. 2020. Effectively Learning Spatial Indices. *Proc. VLDB Endow.* 13, 11 (2020), 2341–2354. http://www.vldb.org/pvldb/vol13/p2341-qi.pdf

[41] Jianzhong Qi, Yufei Tao, Yanchuan Chang, and Rui Zhang. 2020. Packing R-trees with Space-filling Curves: Theoretical Optimality, Empirical Efficiency, and Bulk-loading Parallelizability. *ACM Trans. Database Syst.* 45, 3 (2020), 14:1–14:47. https://doi.org/10.1145/3397506

[42] John T. Robinson. 1981. The K-D-B-Tree: A Search Structure For Large Multidimensional Dynamic Indexes. In *Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data, Ann Arbor, Michigan, USA, April 29 - May 1, 1981*, Y. Edmund Lien (Ed.). ACM Press, 10–18. https://doi.org/10.1145/582318.582321

[43] Nick Roussopoulos, Stephen Kelley, and Frédéic Vincent. 1995. Nearest Neighbor Queries. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, USA, May 22-25, 1995*, Michael J. Carey and Donovan A. Schneider (Eds.). ACM Press, 71–79. https://doi.org/10.1145/223784.223794

[44] Nick Roussopoulos and Daniel Leifker. 1985. Direct Spatial Search on Pictorial Databases Using Packed R-Trees. In *Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data, Austin, Texas, USA, May 28-31, 1985*, Shamkant B. Navathe (Ed.). ACM Press, 17–31. https://doi.org/10.1145/318898.318900

[45] Simonas Saltenis, Christian S. Jensen, Scott T. Leutenegger, and Mario Alberto López. 2000. Indexing the Positions of Continuously Moving Objects. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*, Weidong Chen, Jeffrey F. Naughton, and Philip A. Bernstein (Eds.). ACM, 331–342. https://doi.org/10.1145/342009.335427

[46] Hanan Samet. 1989. Hierarchical Spatial Data Structures. In *Design and Implementation of Large Spatial Databases, First Symposium SSD'89, Santa Barbara, California, USA, July 17/18, 1989, Proceedings (Lecture Notes in Computer Science)*, Alejandro P. Buchmann, Oliver Günther, Terence R. Smith, and Y.-F. Wang (Eds.), Vol. 409. Springer, 193–212. https://doi.org/10.1007/3-540-52208-5_28

[47] Hanan Samet. 1990. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley.

[48] Hanan Samet. 2006. *Foundations of multidimensional and metric data structures*. Academic Press.

[49] Timos K. Sellis, Nick Roussopoulos, and Christos Faloutsos. 1987. The R+-Tree: A Dynamic Index for Multi-Dimensional Objects. In *VLDB'87, Proceedings of 13th International Conference on Very Large Data Bases, September 1-4, 1987, Brighton, England*, Peter M. Stocker, William Kent, and Peter Hammersley (Eds.). Morgan Kaufmann, 507–518. http://www.vldb.org/conf/1987/P507.PDF

[50] Mehdi Sharifzadeh and Cyrus Shahabi. 2006. The spatial skyline queries. In *Proceedings of the 32nd international conference on Very large data bases (VLDB '06)*. VLDB Endowment, Seoul, Korea, 751–762.

[51] Jaewoo Shin, Jianguo Wang, and Walid G. Aref. 2021. The LSM RUM-Tree: A Log Structured Merge R-Tree for Update-intensive Spatial Workloads. *2021 IEEE 37th International Conference on Data Engineering (ICDE)* (2021), 2285–2290.

[52] Yasin N. Silva, Xiaopeng Xiong, and Walid G. Aref. 2008. The RUM-tree: supporting frequent updates in R-trees using memos. *The VLDB Journal* 18 (2008), 719–738.

[53] Yufei Tao, Dimitris Papadias, and Xiang Lian. 2004. Reverse kNN Search in Arbitrary Dimensionality. In *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, VLDB 2004, Toronto, Canada, August 31 - September 3 2004*, Mario A. Nascimento, M. Tamer Özsu, Donald Kossmann, Renée J. Miller, José A. Blakeley, and K. Bernhard Schiefer (Eds.). Morgan Kaufmann, 744–755. https://doi.org/10.1016/B978-012088469-8.50066-8

[54] Yufei Tao, Dimitris Papadias, and Jimeng Sun. 2003. The TPR*-Tree: An Optimized Spatio-Temporal Access Method for Predictive Queries. In *Proceedings of 29th International Conference on Very Large Data Bases, VLDB 2003, Berlin, Germany, September 9-12, 2003*, Johann Christoph Freytag, Peter C. Lockemann, Serge Abiteboul, Michael J. Carey, Patricia G. Selinger, and Andreas Heuer (Eds.). Morgan Kaufmann, 790–801. https://doi.org/10.1016/B978-012722442-8/50075-6

[55] Dimitrios Tsitsigkos, Konstantinos Lampropoulos, Panagiotis Bouros, Nikos Mamoulis, and Manolis Terrovitis. 2021. A Two-layer Partitioning for Non-point Spatial Data. In *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*. IEEE, 1787–1798. https://doi.org/10.1109/ICDE51399.2021.00157

[56] Haixin Wang, Xiaoyi Fu, Jianliang Xu, and Hua Lu. 2019. Learned Index for Spatial Queries. In *20th IEEE International Conference on Mobile Data Management, MDM 2019, Hong Kong, SAR, China, June 10-13, 2019*. IEEE, 569–574. https://doi.org/10.1109/MDM.2019.00121

[57] David A. White and Ramesh C. Jain. 1996. Similarity Indexing with the SS-tree. In *Proceedings of the Twelfth International Conference on Data Engineering, February 26 - March 1, 1996, New Orleans, Louisiana, USA*, Stanley Y. W. Su (Ed.). IEEE Computer Society, 516–523. https://doi.org/10.1109/ICDE.1996.492202

[58] Xiaopeng Xiong, Mohamed F. Mokbel, and Walid G. Aref. 2005. SEA-CNN: Scalable Processing of Continuous K-Nearest Neighbor Queries in Spatio-temporal Databases. In *Proceedings of the 21st International Conference on Data Engineering, ICDE 2005, 5-8 April 2005, Tokyo, Japan*, Karl Aberer, Michael J. Franklin, and Shojiro Nishio (Eds.). IEEE Computer Society, 643–654. https://doi.org/10.1109/ICDE.2005.128

[59] Xiaohui Yu, Ken Q. Pu, and Nick Koudas. 2005. Monitoring K-Nearest Neighbor Queries Over Moving Objects. In *Proceedings of the 21st International Conference on Data Engineering, ICDE 2005, 5-8 April 2005, Tokyo, Japan*, Karl Aberer, Michael J. Franklin, and Shojiro Nishio (Eds.). IEEE Computer Society, 631–642. https://doi.org/10.1109/ICDE.2005.92