Adaptive Indexing of Objects with Spatial Extent

Fatemeh Zardbani
Aarhus University
fatemeh.zardbani@cs.au.dk

Nikos Mamoulis
University of Ioannina
nmamoulis@gmail.com

Stratos Idreos
Harvard University
stratos@seas.harvard.edu

Panagiotis Karras
Aarhus University
piekarras@gmail.com

ABSTRACT

Can we quickly explore large multidimensional data in main memory? *Adaptive indexing* responds to this need by building an index incrementally, in response to queries; in its default form, it indexes a single attribute or, in the presence of several attributes, one attribute per index level. Unfortunately, this approach falters when indexing spatial data objects, encountered in data exploration tasks involving multidimensional range queries. In this paper, we introduce the Adaptive Incremental R-tree (AIR-tree): the first method for the adaptive indexing of non-point spatial objects; the AIR-tree incrementally and progressively constructs an in-memory spatial index over a static array, in response to incoming queries, using a suite of heuristics for creating and splitting nodes. Our thorough experimental study on synthetic and real data and workloads shows that the AIR-tree consistently outperforms prior adaptive indexing methods focusing on multidimensional points and a pre-built static R-tree in cumulative time over at least the first thousand queries.

1 INTRODUCTION

Applications in geographic information systems [31] and 3D scientific data analysis [26] call for exploring data sets of spatial objects at different levels of granularity [22]. The user needs to explore such large amounts of data, yet cannot invest time to build a complete index in advance. For example, consider an application that identifies spatial data objects in satellite images and makes those spatial data available to scientists interested in searching for objects in specific regions; building an index for all objects upfront may not be worthwhile, as *queries are few* and target a limited space. Once the user has finished exploring one spatial area, they may move to a new one, requiring only one part of a disk-resident data set to be indexed in memory at a time. In these circumstances, the need arises to *build and refine* an in-memory index in response to queries indicating user interests [12]; that is, *adaptive indexing* [16, 18, 35] — and, in its most widespread form, *database cracking* [14]. Adaptive indexing allows for straightforward access to the data without the overhead of a priori indexing. By the time a full indexing approach is still preparing the index, an adaptive indexing technique will have already answered thousands of queries [12]. We emphasize that the term "*adaptive*" signifies not only that an index *tunes* itself in response to queries, as in [2], but also, most crucially, that an index *assembles itself in response to queries*.

Existing adaptive indexing methods organize each index around a single dimension or attribute [15], or otherwise dedicate each level of a multilevel index to a single dimension [13, 17, 27]; thus, they accommodate non-point multidimensional objects with spatial extent, as those that occur in scientific applications [26], only extrinsically, via query transformation and post-processing.

In this paper, we introduce the *Adaptive Incremental R-tree* (AIR-tree): an *inherently* multidimensional adaptive hierarchical in-memory index structure that reorganizes and expands itself in response to range queries. Unlike previous art, the AIR-tree keeps track of all spatial dimensions and object extents in each index level, rather than indexing one dimension per level; it progressively constructs a hierarchy of d-dimensional *minimum bounding boxes* (MBBs), splitting such MBBs in response to incoming queries and expanding the index accordingly; it uses a suite of heuristics to maintain the structure compact, minimize indexed empty space, and avoid overlap among nodes. When the number of objects in an MBB falls below a threshold, that MBB is not cracked further. The index eventually becomes comparable to a static in-memory R-tree [26].

We evaluate the AIR-tree with synthetic and real, two- and three-dimensional spatial data objects and workloads. Our results show that the AIR-tree outperforms previous art on multidimensional adaptive indexing [13, 27] in per-query and cumulative response time to a workload by at least one order of magnitude. Moreover, the ensuing in-memory index may even match the steady-state per-query performance of a conventional static in-memory R-tree.

We summarize our main contributions as follows:

- We repurpose criteria designed for spatial index updates to adaptive indexing of objects with spatial extent.
- We transcend prior art [25, 27] by overseeing all object extents in each index level, picking the order by which to crack on query boundaries by a margin-based criterion, and tightly adjusting those boundaries on the crack dimension.
- We show that the AIR-tree also manages point data better than existing methods, as it builds a more compact tree.

- We show that the AIR-tree fares well as an index-building method too, as it effectively disentangles larger objects from smaller ones thanks to its query-driven character.

2 RELATED WORK

We overview fundamentals on spatial indexes, adaptive indexing, and previous attempts for multidimensional adaptive indexing.
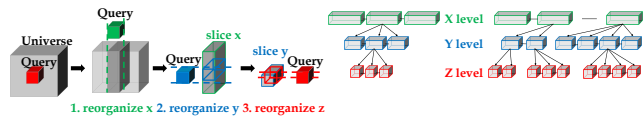
2.1 Spatial indexes

The **KD-tree** [6] is a *binary search tree* that organizes points in space partitions. Each internal node bisects space along one dimension and each level corresponds to one dimension in round-robin fashion. A node at level ℓ represents a data point p, with its left (right) child indexing points preceding (succeeding) p along the dimension associated with ℓ. Other space partitioning methods, the quadtree [19], octree [23], and loose octree [33], create quadrants or octants using predefined boundaries, oblivious to the data.

An **R-tree** [11] organizes spatial data in a hierarchy of multi-dimensional *minimum bounding boxes* (MBBs), each level adding further detail. In effect, an R-tree resembles a multidimensional version of a B-trees. Unlike a KD-tree, an R-tree holds data on leaves only. While worst-case optimal [3] and I/O-optimized variants [5] exist, practical *heuristics* craft MBB boundaries that reduce indexed empty space, margins (targeting square-like MBBs), and overlap among MBBs [4, 10, 26, 30]. Unlike KD-trees and other structures designed for point data, the R-tree is tailored for non-point spatial objects, while also capable to manage points. We develop an *adaptive* R-tree for real-world spatial data with spatial extent.

2.2 Adaptive indexing

Database cracking. *Adaptive indexing* builds an index as a side-effect of query processing [18]; it incurs a small overhead in query response times while progressing to a full index. A prominent adaptive indexing method for column-oriented databases is *database cracking* [14]; cracking uses *range query bounds* as pivots to incrementally *quicksort* numerical values and gradually build an index organized on query bounds. Each query traverses the tree constructed by preceding ones to reach the data values in its range, cracks those index parts further, up to the resolution of a cracking threshold [14], and expands the index accordingly. To achieve robustness, *stochastic cracking* uses carefully selected pivots in addition to those dictated by query ranges [12, 35].

(a) Incremental indexing strategy (b) Index structure after queries

Figure 1: QUASII indexing strategy and data structure [27].

QUASII. The first attempt to apply adaptive indexing to spatial data was the *QUery-Aware Spatial Incremental Index* (QUASII) [27], which applies cracking to one dimension per tree level. Like a KD-tree, QUASII splits the space on a different dimension per level, yet, unlike a KD-tree, uses each dimension only once. For each query, QUASII first cracks along the x-dimension to organize data on the first tree level; it then cracks and indexes the piece corresponding to

the query's x-range, $[x_\ell, x_h]$ along the next dimension, y, on bounds $[y_\ell, y_h]$ on the second tree level; and so on. The ensuing index is a *wide* tree of d levels, each associated with one of d dimensions. Figure 1 illustrates the operation. Nevertheless, QUASII partitions space into rectangles that are *not* MBBs of the data they hold, hence are suboptimal for processing spatial query workloads [4]; it accommodates data objects with spatial extent only by means of *query window extension* [31]: representing all data objects as points, it extends each query window by the size of the largest object in the data, and removes false positives from query results.

Cracking KD-trees. Another attempt to extend adaptive indexing to spatial data sets, concurrent to QUASII, is the *adaptive KD-tree* (AKD) [13, 25]; it also applies cracking to one dimension per tree level. However, it lets the tree height grow while building a KD-tree in response to queries, cyclically returning to the same dimension modulo d. Thus, a range query adds up to $2d$ new tree levels, one for each bound along each dimension. For a uniformly-distributed two-dimensional data set, the ensuing partitioning resembles a grid. Figure 2 shows how the tree evolves in response to a query.
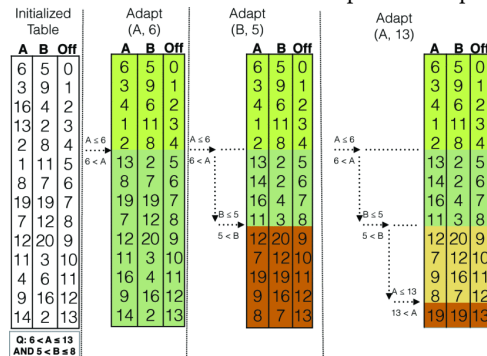

Figure 2: KD-tree indexing strategy and data structure [25].

The AKD was extended to a *progressive* variant [25] which builds an index while responding to queries, yet the queries have no bearing on the process. Each query lets a portion of the data set be indexed, whose size depends on a parameter δ; the index converges to a full index after a sufficient amount of queries. However, a fixed δ results to large query response time variance; to ensure uniform performance, a *greedy progressive* variant adjusts δ by a cost model, hence being more robust than the standard adaptive KD-tree, albeit more costly in overall query response time. All variants are designed for points rather than objects with spatial extent.

2.3 Learned indexes

Some works [7, 24] use grid-based *learned* multidimensional indexes. A learned model determines the size of each grid cell and models the data distribution therein, based on a sample query workload. These methods learn a data model using a sample workload, yet aim to reduce index space rather than to build an index in real time; besides, they are designed for point data. Thus, they constitute a research direction with a different aim from ours.

2.4 Other forms of workload-awareness

Other works build a workload-aware index *offline* [1, 8, 21, 34] or first create an index and then refine it *online* in response to queries [2], yet do not *assemble* the index in response to queries.

3 THE ADAPTIVE R-TREE

This section presents the AIR-tree and its construction. First, we describe its desirable properties and initialization. Next, we outline the cracking operations in response to queries and the memory allocation scheme. Delving deeper, we explain how we craft a data-oriented partitioning out of the ensuing space partitioning and introduce a *lazy boundary maintenance* strategy. Lastly, we outline extensions to stochastic actions and to higher dimensions.

3.1 Core properties

When building an AIR-tree in response to queries, we aim to preserve the following core properties of a well-designed R-tree [4, 11]:

- **Root.** If not a leaf, the root has *at least two* children.
- **Internal node bounds.** A non-root internal node holds a number of children *bounded* in an interval $[m_f, M_f]$.
- **Leaf bounds.** A leaf holds a *bounded* number of objects.
- **Balance.** All leaves are at the *same* level.
- **Overlap.** Overlap among MBBs is kept *small*.
- **Margins.** MBBs are square-like [11]; when allocating objects we prefer to create MBBs with *small* sum of margins.
- **Dead space.** The indexed space devoid of data is kept *small*.

Past research has used heuristics [4, 11] to achieve the last three properties, which are quantitative. We preserve the goals of such heuristics as we describe in Section 3.3. Next, we outline how we partially relax the upper bound M_ℓ on data objects per leaf.

3.2 Leaf types

Index construction begins with a single root node that is also a leaf node comprising all data objects brought in main memory from external storage. The tree grows by processing queries. As we cannot always adhere to the bound M_ℓ on the number of data objects a leaf holds, we introduce two types of leaves:

- **irregular**, which hew to no restriction on held data objects;
- **regular**, which adhere to the bound M_ℓ on held data objects.

The initial root node is an irregular leaf holding all data objects. Any newly created leaf node may also start out in an *irregular* state; however, as a leaf gets cracked, it spawns leaves of fewer contents; eventually, a leaf's count falls below M_ℓ, whereupon it becomes *regular*. We aim for the AIR-tree to progress towards a state similar to that of a regular R-tree, rendering *all leaves* regular; once an irregular leaf turns regular, it never returns to an irregular state. Ideally, M_ℓ must be set to a value large enough to let the AIR-tree quickly converge to a regular state and avoid accessing multiple small leaves for query answering, yet small enough to facilitate quick result retrieval; we find a proper value experimentally in Section 4.4, along with a value for the internal node fanout M_f. We emphasize that *regularity* only applies to leaves; internal nodes never exceed the threshold M_f on the number of their entries.

3.3 Spatial partitioning

The AIR-tree responds to a spatial range query as a regular R-tree, yet also performs indexing as a side-effect of query processing. With each query, we traverse the tree built so far along one or more branches, until we reach one or more leaves that match the query range. In case a leaf λ is *fully* contained within the query range, we append its contents to query results. On the other hand, in case a leaf λ *partially intersects* the query range, its treatment depends on its regular or irregular status; regular leaves are already well-refined, while irregular ones need further cracking. Therefore, we treat each leaf λ partially intersecting a query as follows:

- if λ is regular, we scan it and collect query results;
- if λ is irregular, we crack it and collect query results.

To crack an irregular leaf, we crack on one query bound at a time and create an MBB for the contents of the resulting piece on the *non-query* side. Figure 3 presents a query range overlapping a leaf in two dimensions. The solid-lined outer rectangle is the MBB of an irregular leaf and the solid-lined inner one indicates the query bounds. We create a piece that includes the query results and new pieces for the data *fully outside* the query bounds. A naïve option is to create 8 new pieces, a to h in the figure, in addition to the middle piece. However, this method incurs a significant overhead, which does not necessarily benefit subsequent queries; for instance, based on the given query, dividing the space on the left side of the query into 3 pieces, a, d, and f, may not be useful for consequent queries. Another option is to create pieces b and g, along with one piece consisting of the union of a, d, and f, and another piece being the union of c, e, and h. Still, this solution creates elongated MBBs on the two sides of the query, whereas we prefer square-like MBBs.
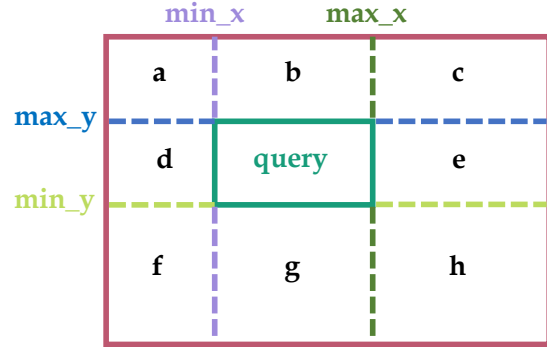

Figure 3: General 2D cracking.

In a case as in Figure 3, we create *at most five* pieces by means of four consecutive cracks, each partitioning an existing MBB on a query bound. For each crack, we choose the query bound to crack on by a *margin-oriented* heuristic on the piece containing the query: we find the axis α on which that piece has largest extent and crack it on the one of two query bounds along α closest to the middle of that extent on α. Thereby, unlike other works [13, 27], we create square-like pieces, bringing the index to a desirable form [11]. A piece thus created may be too *sparse*; still, as long as it is not empty, we create such a piece as a *regular* leaf, never to be cracked again.

ALGORITHM 1: AIR-tree Query

Result: Data objects in range of query q

1 query(q: *query bounds*)
2   Search(q, root); // R-tree range search traversal
3   Regular = set of regular leaves visited;
4   Irregular = set of irregular leaves visited;
5   **if** (Irregular *not empty*)
6   ND-Crack all leaves in Irregular;
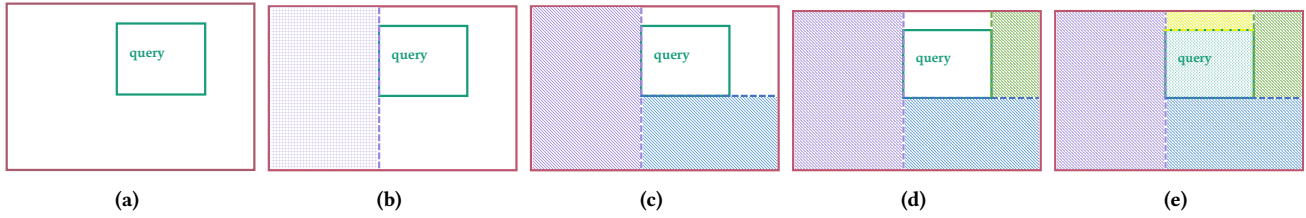7   return results found in Regular, Irregular;

(a) **(b)** **(c)** **(d)** **(e)**

Figure 4: 2D-Cracking: spatial partitioning.

(a) **(b)** **(c)** **(d)** **(e)**
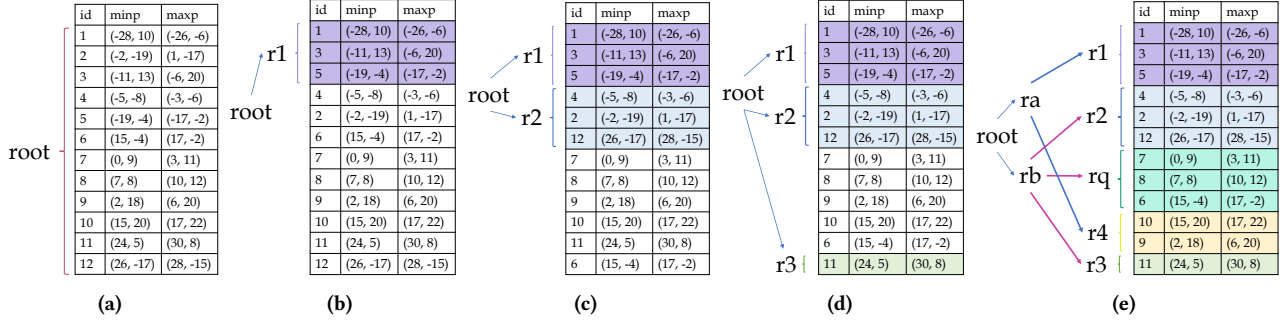
Figure 5: 2D-Cracking: in-place array partitioning.

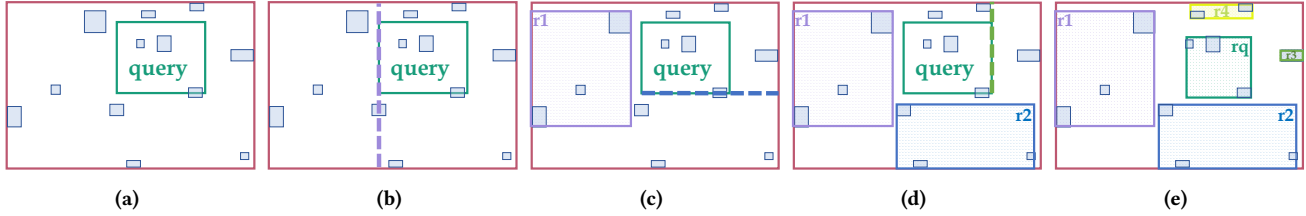**(a)** **(b)** **(c)** **(d)** **(e)**

Figure 6: 2D-Cracking: data partitioning.

ALGORITHM 2: ND-Crack

Result: Crack leaf ℓ based on query q and return results in ℓ

1 ND-Crack(ℓ: *leaf, q: query rectangle*)
2      $tp = \ell$; // this piece
3      $tcp = \ell$; // this choice piece
4      $lta = \{\}$; // leaves to add
5   **for** (*each query bound*)
6   α = longest side of tcp;
7   p = bound of q on α closest to middle of tcp;
8          $qp, op = tp$.partition(α, p); // query/other piece
9          $tp = qp$; // this piece is the query piece
10   **if** (*op not empty*)
11             $lta.append(op)$;
12             $tcp = tp$;
13   **else**
14   set corresponding bound of tcp to p on α;
15   **if** ($tp.size() \leq M_\ell$)
16   break;
17   scan tp for results;
18     $lta.append(tp)$;
19   remove ℓ;
20   recalculate MBBs of regular leaves in lta;
21   add lta to parent of ℓ; // R-tree insertion [11]
22   return results;

Algorithm 1 illustrates the shell of the query processing method, while Algorithm 2 presents the cracking operations. Figure 4 shows

an example where the query overlaps one irregular leaf. We first choose the axis of largest leaf extent (Line 6), which is the x-axis. As the lower x-bound of the query is closer to the middle of the leaf's width (Line 7), we crack on that bound, yielding a new piece on the left of the query (Line 8), as Figure 4b shows. We repeat on the piece left on the query side (Line 9). The height of that piece is larger than its width, hence we crack along the y-axis. As the lower y-bound of the query is closer to the middle of the piece's height, we choose that as a cracking pivot, yielding a new piece in Figure 4c. The process goes on as Figure 4 shows. To determine the cracking pivot in each step, we apply this heuristic on the query piece resulting from the previous crack (Lines 12 and 14).
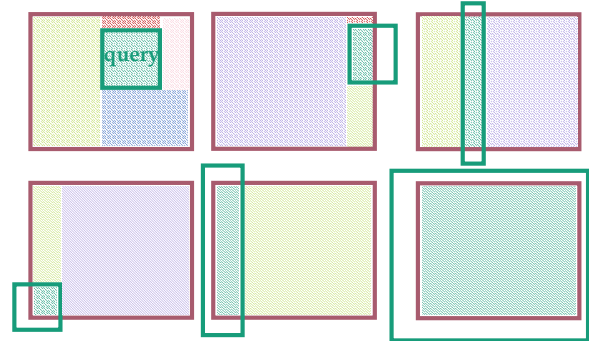

Figure 7: Topologies of overlap between leaf and query.

In Figure 4, a leaf MBB *fully* contains the query. Yet a query may overlap a leaf MBB *partially*. Figure 7 shows all leaf-query overlap

topologies in 2D and the resulting spatial partitions. We always follow the same process, yet do not crack on query bounds fully outside the leaf MBB. The first case in Figure 7 is the general case. In the second case, as the right-side query bound falls outside the leaf MBB, we do not crack on it; likewise in the other cases.

Yet another complication arises: if all *non-query-side* objects intersect the pivot, they are re-assigned to the query-side piece qp, as we discuss in Section 3.5; thus, the query piece is unaffected and the other piece is empty and superfluous. Figure 8 shows two examples of such *abortive* cracking. If we then choose the next pivot based on the unaffected query piece, we repeat the same choice. To avoid this deadlock, we truncate the unaffected query piece on the pivot anyway (Line 13), yielding the tcp piece in each figure, and apply the heuristic on tcp to choose the next pivot (Line 6).
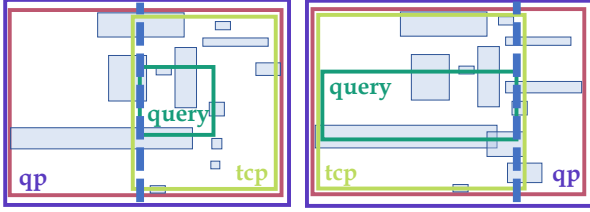

Figure 8: Abortive cracking.

At any time, if a piece to be cracked turns out to be regular, i.e., holds no more than M_ℓ objects, we abandon cracking and scan the piece to gather query results (Line 15).

ALGORITHM 3: Partition on lower (upper) bound

Result: Partition data-array, D[$l : h$] based on p in α

1   partition(*l: start of piece, h: end of piece to crack, α: chosen dim*)
2     $x_1 = l$;
3     $x_2 = h - 1$;
4     $lc[i] = piece\_cover[i] \ \forall i \neq \alpha$; // left chunk
5     $rc[i] = piece\_cover[i] \ \forall i \neq \alpha$; // right chunk
6   case min: $rc[\alpha].max = piece\_cover[\alpha].max$;
7   case max: $lc[\alpha].min = piece\_cover[\alpha].min$;
8   initialize other $lc[\alpha]$ and $rc[\alpha]$ bounds to extreme values;
9   **while** ($x_1 \leq x_2 \wedge x_2 > 0$)
10   **if** ($D[x_1][\alpha].max \ (min) < p$)
11   update lc using $D[x_1]$;
12         $x_1 = x_1 + 1$;
13   **else**
14   **while** ($x_2 > 0 \wedge x_2 \geq x_1 \wedge D[x_2][\alpha].max \ (min) \geq p$)
15   update rc using $D[x_2]$;
16           $x_2 = x_2 - 1$;
17   **if** ($x_1 < x_2$)
18   swap($D[x_1]$, $D[x_2]$);
19   update lc using $D[x_1]$;
20   update rc using $D[x_2]$;
21           $x_1 = x_1 + 1$;
22           $x_2 = x_2 - 1$;
23   return x_1, rc, lc;

3.4 In-place array partitioning

To avoid the overhead of dynamic memory allocation, we store the data in a static array, as in [14]. Each index leaf points to a range of array entries. We perform cracking *in place* [29] by swapping

array entries around the pivot on the chosen dimension. Figure 5 shows an example corresponding to Figure 4. Algorithm 3 shows the quicksort-inspired cracking process for a lower (upper) query bound serving as pivot, which we compare to the max (min) MBB bound values of data objects along the chosen dimension α. In both cases, we set a pointer at each end of the α axis (Lines 2 and 3), one moving rightwards (Lines 12 and 21) while stowing objects of value *less* than the pivot on its left (Lines 10 and 18), the other moving leftwards (Lines 16 and 22), while stowing objects of value *greater* than the pivot on its right (Lines 14 and 18). When the two meet (Line 9), the procedure ends and returns the first pointer, pointing at the pivot location (Line 23).

3.5 Data partitioning

We have hitherto focused on how we partition *space* in response to queries. However, the MBBs we create do not necessarily follow the space partition boundaries. To enhance performance, we adjust the MBB boundaries of the two partitions resulting from cracking to *tightly* enclose the data objects allocated to each partition on the fly, while swapping data objects (Lines 11, 15, 19, 20). As discussed in Section 3.3, the resulting partition on the query side may be cracked next, while the other spawns a new leaf with the associated MBB.

Figure 6 shows the data partitioning resulting from the space partitioning of Figure 4. In case a data object *intersects* a partition boundary, we allocate the whole object to the *query-side* partition and adjust its boundaries accordingly, so that its MBB contains query results. As discussed in Section 3.4, when cracking on a *lower* query bound, as in Figure 6b, we allocate data objects among the two sides by comparing their *max* MBB bounds to the pivot (Lines 10 and 14); when cracking on an *upper* query bound, as in Figure 6e, we allocate data objects by comparing their *min* MBB bounds to the pivot; in effect, we always keep objects intersecting the query range on the query side. The MBBs for each non-query-side data partition are shrunk compared to those of the corresponding space partition. On the other hand, the MBB taking the place of the query window in Figure 6e has its lower y-axis bound lowered to include an object partially overlapping the query window at that locale.
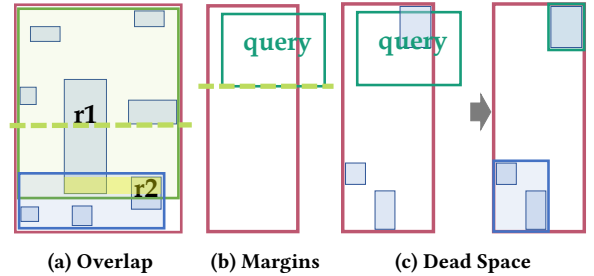

(a) Overlap   (b) Margins   (c) Dead Space

Figure 9: AIR-**tree heuristics.**

To sum up, we aim for the three quantitative characteristics of a well-designed R-tree mentioned in Section 3.1 (i.e., small overlap, small margins, and small dead space) as follows:

- To reduce *overlap* amongst node MBBs, we assign data objects intersecting the crack pivot hyperplane, like $r1$ in Figure 9a, to the resulting piece on the *query side* (in the figure, upper side); any persisting overlap is due to data objects on the non-query-side MBB (in the figure, lower

side) whose *pivot axis* (*y*) range overlaps that of a pivot-intersecting object, as the *y*-range of *r*2 overlaps that of *r*1.

- To achieve small *margins*, we strive to create square-like pieces; to that end, we choose a cracking pivot as the most centrally located query bound along the longest dimension of a piece to be cracked; thereby, we bring MBBs closer to a square-like shape. Figure 9b shows an example where an MBB is broken into two more square-like ones by the query bound near the middle of its *y*-axis stretch.
- Lastly, by their tightness, MBBs avoid indexing *dead space* devoid of data; besides, while cracking, a query that intersects dead space in an MBB spawns more compact MBBs curtailing that dead space; Figure 9c shows an example.

3.6 Lazy boundary maintenance

As discussed in Section 3.4, we calculate the MBBs of the partitions resulting from cracking on the fly, avoiding extra passes on the data. To that end, we should check and possibly update a partition's MBB boundaries whenever we assign a data object therein. We do so in Lines 11, 15, 19, 20 of Algorithm 3. In these updates, we *might* maintain tight boundaries *for all* dimensions. However, this operation does not scale as the number of dimensions grows. In our trial experiments, performance deteriorated already when moving from two to three dimensions. Therefore, we eschew such *eager* MBB boundary maintenance on all dimensions in favor of a *lazy* MBB boundary maintenance along the *cracking dimension only* for the two pieces resulting from cracking. Accordingly, Algorithm 3 updates MBB boundaries of lc and rc *only* on the cracking dimension α; MBB boundaries on other dimensions follow those of the original piece (Lines 4 and 5). Further, we forego updating the *non-pivot-side* boundary of the *query-side* partition along the cracking dimension (Lines 15 and 20); this omission is innocuous, since any data object that defines this boundary *remains* on the query side; in case the boundary is tight, it remains tight; in case it is loose, it remains equally loose.

Figure 10: MBB calculation in partition.

Figure 10 shows an example on the 2-dimensional case. When cracking for the query in the figure along the *x*-axis (the vertical dashed lines), we set the *y*-axis boundaries for both resulting partitions as equal to those of the original parent piece, as the top and bottom (dotted and dash-dotted lines) lines indicate. On the other hand, we eagerly update, with each data object swap, both *x*-axis boundaries of the *non-query-side* partition resulting from cracking, as well as the *pivot-side x*-axis boundary of the *query-side* partition. Still, we do not update the *non-pivot-side x*-axis boundary of the query-side partition (indicated in orange striped lines).

Notably, this lazy boundary maintenance lets some loose boundaries exist in newly added tree leaves for some time during adaptive indexing. Such loose boundaries may result in redundant tree traversal and search operations in subsequent queries. However, this

overhead is negligible and overshadowed by the gains of avoiding eager boundary updates, as we update at most 2 rather than 2*d* boundaries per swap operation. Besides, whenever we crack a piece along a dimension on which there is a loose boundary, that boundary becomes tight. To ensure the eventual tree has tight boundaries, we recalculate tight MBB boundaries on *all* dimensions when adding to the tree a *regular* leaf node, not to be ever cracked again. Thus, when all leaves become regular, all MBBs are tight.

ALGORITHM 4: S-ND-Crack
---
1 psc = most populated piece in lta; // piece to scrack
2 α_s = longest side of psc;
3 p_s =mediocre of α_s-axis values in psc; // pivot for scrack
4 $lp, rp = psc.\,\text{partition}(\alpha_s, p_s)$; // left, right pieces
5 **if** (rp *not empty* $\wedge$ lp *not empty*)
6      $lta.remove(psc)$;
7      $lta.append(lp)$;
8      $lta.append(rp)$;

3.7 Stochastic extension

Skewed workloads may have a negative effect on the performance of adaptive indexing, as they trigger repetitive cracking actions on large non-indexed areas [12]. To address such repercussions, we extend our procedure with a benign stochastic element: that is, we crack on an additional randomly chosen pivot within each cracked tree node, after all cracks on query bounds. We implement this extension by adding the pseudo-code in Algorithm 4 to Algorithm 2 after Line 18. In particular, we choose the largest remaining piece (Line 1), find its axis of longest extent (Line 2), and crack on a value along that extent (Line 4). This choice is similar to the pivot choice in quicksort; for best results, we should choose the *median* value among all data objects, hence partition them in two halves. However, the overhead of finding a median is too high a price to pay. Instead of an exact median, we opt for a *mediocre* element, i.e., the median of a small data sample (Line 3). Even the median of a small constant-sized sample serves the purpose well [35], while the single extra crack does not cause a distressing amount of overhead. By the analysis in [35], we employ a small constant-size sample.

3.8 Three-dimensional case

While we have discussed the two-dimensional case, our solution applies to the general *d*-dimensional case, for any *d*. Figure 11 illustrates how the discussed principles apply to three dimensions. The initial irregular leaf node is a cuboid anchored at (0, 0, 0), with width and height 10 and depth 20. The query range is a cuboid with origin at (3, 5, 8), width 3, depth 2, and height 7, as Figure 11a shows. Since the leaf node is larger on depth, we first crack along the *z* axis; the *z*-axis query bounds are 8 and 15, while the *z*-axis middle of the leaf's MBB is at 10, thus we use bound 8, which is closer to the middle, as pivot. Figure 11b shows the first crack as a plane perpendicular to the *z* axis. We move on to the subspace above this plane, which contains the query; the extent of this subspace along the *z* axis, 12, is still larger than others, so we crack along *z* again, this time on the higher query bound 15, as in Figure 11c. Now the piece containing the query has largest extent 10 along both axes *x* and *y*; we pick *y* at random. The *y*-axis query bounds are 5 and 7, the former exactly at the *y*-axis middle of the piece's extent, hence
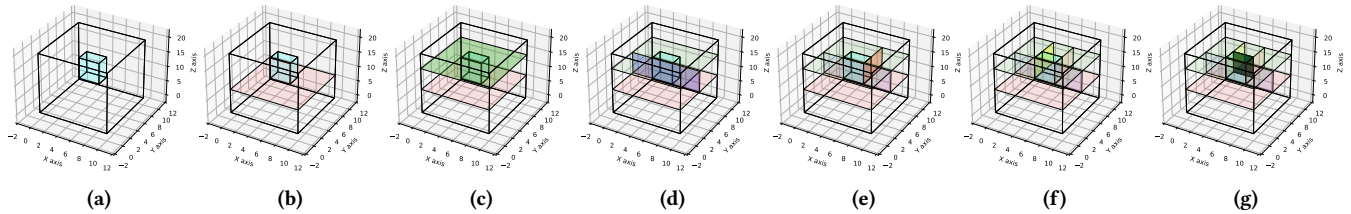
Figure 11: 3D-Crack: spatial partitioning.

we crack on that, as in Figure 11d. The resulting query-side piece has largest extent along the x axis, while the query's x-bounds are 3 and 6; we crack at pivot 6, which is closer to the piece's middle, as in Figure 11e. The resulting query-side piece still has larger extent, 6, on the x axis, so we crack on x again, now with query bound 3 as pivot, as in Figure 11f. Lastly, we crack on the higher y-bound, 7, as Figure 11g shows. By the same principle, in dimensionality d we do one crack for each of two query bounds per dimension, hence $2d$ cracks in total, spawning at most $2d + 1$ pieces. When a query is not fully contained in a leaf node's MBB, we have fewer cracks.

The natural application domain of the AIR-tree is real-world data objects with spatial extent in two or three dimensions; in this domain, R-trees outperform indexes designed for point data [4]. Besides, the size of an R-tree is not affected by dimensionality d, while the height of KD-tree-based indexes increases with d [17]; the AIR-tree preserves this advantage. In effect, the cost of a *root-to-leaf traversal* in an AIR-tree does not grow with d, while the cost of *cracking operations* grows only linearly with d.

4 EXPERIMENTS

We conducted exhaustive experiments on the AIR-tree vs. previous work on in-memory multidimensional adaptive indexing using real and synthetic data and realistic workloads. We implemented[1] all methods in C++ and compiled them in g++ 7.4.0 with the -o3 switch; experiments ran on a 10-core Intel Xeon machine at 3.10GHz with 396G RAM running Ubuntu 18.04.3 LTS. We compare the AIR-tree and its stochastic extension, s-AIR, against standard implementations of the following methods:

- QUASII [27];
- Cracking KD-trees [13]: Adaptive KD-tree (AKD), Progressive (PKD) and Greedy Progressive KD-tree (GPKD);
- A static[2] in-memory R-tree.

As QUASII and Cracking KD-Tree[3] target point data, we employ *query window extension* [27] to accommodate data objects with spatial extent: we represent data objects by their lower coordinates in each dimension, and extend each query window by the maximum data object extent towards the lower side in each dimension; this way, a query hits the lower coordinates of any object overlapping its range. We filter false hits in a post-processing step. We measure runtime including post-processing, yet omit the geometric refinement of results checking object shapes, common to all methods. All methods produce the same query results. We use the settings recommended in previous works: QUASII and Cracking KD-trees use a partition size of 1024 and PKD uses $\delta = 0.2$. The static R-tree uses

fanout 16, as recommended in the implementation. We set AIR-tree parameter M_ℓ to 64 and M_f to 16, as discussed in Section 4.4. In s-AIR, we select the mediocre as the median of 3 sampled points.

Table 1: Data sets.

Name	Size	dim.	max ext.	avg ext.
Uniform	64M	2	(5.73, 5.4)	(0.3, 0.3)
ROADS	20M	2	(0.27, 0.25)	(4e-4, 3e-4)
EDGES	70M	2	(0.5, 0.1)	(1e-4, 1e-4)
LAKES	7M	2	(7.9, 5.5)	(4e-3, 3e-3)
Uniform	64M	3	(5.8, 4.9, 5.5)	(0.3, 0.3, 0.3)
TLC	75M	3	(0.01, 0.98, 0.99)	(1.5e-5, 0.07, 0.1)

4.1 Data sets

We focus on the adaptive indexing of data objects with spatial extent. As real-world spatial data objects exist in two or three dimensions, that is the natural application domain of our methods.

We crafted a large synthetic 2D data set of 64M objects. This data set consists of rectangles whose lower-left corner is located uniformly in $(0, 10)$ on each dimension. The objects' width and height follow an exponential distribution $g(v) = 3\,e^{-3v}$. We use subsets of this data set for our scalability experiments in Section 4.6.1.

We also use three public real-world 2D data-sets: the ROADS data feature shapes of U.S. roads and the EDGES data comprise lines on the U.S. map, including roads, rivers, and borders. The LAKES data hold global boundaries of bodies of water extracted[4] from OpenStreetMaps. All three data sets are available[5] at the University of Minnesota [9]. To examine 3D data, we generate a synthetic data set following the same process and parameters as in two dimensions. For a real-world analysis of 3D data, we use taxi cab trip records[6] of the year 2010 by the New York City Taxi and Limousine Commission (TLC), normalizing pick-up and drop-off longitudes, latitudes, and timestamps to represent 3D boxes in the location-time space. Table 1 summarizes data statistics.

4.2 Workloads

Table 2 shows the characteristics of our workloads. We generate synthetic workloads, as in [25], comprising 100K queries each. Query locations (i.e., lower-left corner coordinates) follow a *uniform* or a *clustered* Gaussian distribution. Query dimensions (widths, heights, depths) follow a *normal* distribution. We also generate a skewed *sequential* workload for the synthetic data set. In more detail:

Uniform. To devise *uniform* workloads, we generate corner points from a uniform distribution within the range of each data set. The synthetic and TLC data are normalized to the range $[0, 1]$. Thus, for d-dimensional synthetic data, we produce query ranges

---

[1]Code available at https://gitlab.com/adaptivertree/artree
[2]Code available at https://www.boost.org/users/history/version_1_61_0.html
[3]Code at http://github.com/pdet/MultidimensionalAdaptiveIndexing

[4]https://www.openstreetmap.org/
[5]http://spatialhadoop.cs.umn.edu/datasets.html
[6]https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page

from Gaussian distribution $N(1 \times 10^{-4/d}, 1 \times 10^{-5})$, resulting in boxes that cover 0.01% of the space on average; for the TLC data, we create query boxes with width in $N(1 \times 10^{-3}, 1 \times 10^{-5})$, height in $N(0.02, 1 \times 10^{-5})$, and depth in $N(0.05, 1 \times 10^{-5})$. The ROADS and EDGES data comprise latitudes and longitudes, so we use the minimum and maximum values per dimension to obtain ranges; we generate query extents that cover 0.01% of the space on average, as for the synthetic data, with appropriate scaling.

Skewed. To devise *clustered* workloads, we generate corner points as isotropic Gaussian blobs using the make_blobs function of Python's scikit-learn [28] module. Figure 12a renders such a distribution of the lower-left query corners in a workload. We also use a *sequential* workload of consecutive queries as in Figure 12b, fixing query extents to ensure the chosen selectivity; the corners of query rectangles are points on the identity line.
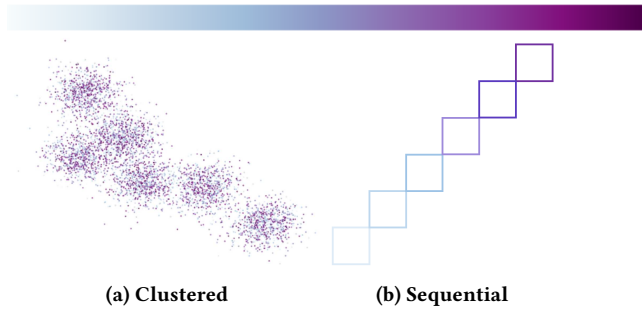

(a) Clustered   **(b) Sequential**

Figure 12: Skewed workload distributions.

Data Distribution. Unlike the ROADS and EDGES data, which pertain to the contiguous United States, the LAKES data feature bodies of water all over the world. In effect, oceanic areas, which cover some 69% of the earth's surface, are empty. To avoid a prevalence of empty query results, for the LAKES data we use a workload that follows the data distribution. We randomly select 100K objects from the data set and extend their widths and heights by 1.5 times on each side, rendering the query area 16 times larger than, and centered on, the data object MBB.

Table 2: Workloads.

Location distribution	Size distribution	Source	Size
Uniform	Uniform	Synthetic	100k
Clustered	Uniform	Synthetic	100k
Data Distribution	Data Distribution	Synthetic	100k
Sequential	Fixed	Synthetic	1000

4.3 Measures

We measure the progressively evolving response times and sizes of the constructed indexes. In terms of response times, we follow the conventions in previous works [12, 13, 15, 27], measuring: (i) the oscillating cost per query over a workload, averaged over 10 runs — which we aim to progressively lower and eventually render indistinguishable from, or even lower than, that achieved with a static pre-built index; and (ii) the monotonically growing cumulative cost, which aggregates the cost per query over a workload. For the sake of fairness, all methods perform identical count queries; we verified the correctness of results. Query response times inevitably fluctuate, as some queries trigger more cracking operations than others; to visualize results comprehensibly, we add a continuous

moving-average line in plots, with window size ranging from 5 to 20 based on the variance of measurements.

Table 3: Parameters, uniform 2D shape data and workload.

M_ℓ \ M_f	4	8	16	32
16	104.559	92.68	91.254	-
64	94.323	87.433	86.132	88.078
256	94.584	91.378	91.367	92.505
1024	116.793	114.685	114.496	115.536
4096	176.418	173.539	174.047	174.337

⚠ 9 piece static lazy   ⊙ 5 piece dynamic lazy   ▫ 5 piece static lazy   ▽ 5 piece static eager

(a) Time per query, Uniform   **(b) Cumulative time, Uniform**

(c) Time per query, ROADS   **(d) Cumulative time, ROADS**

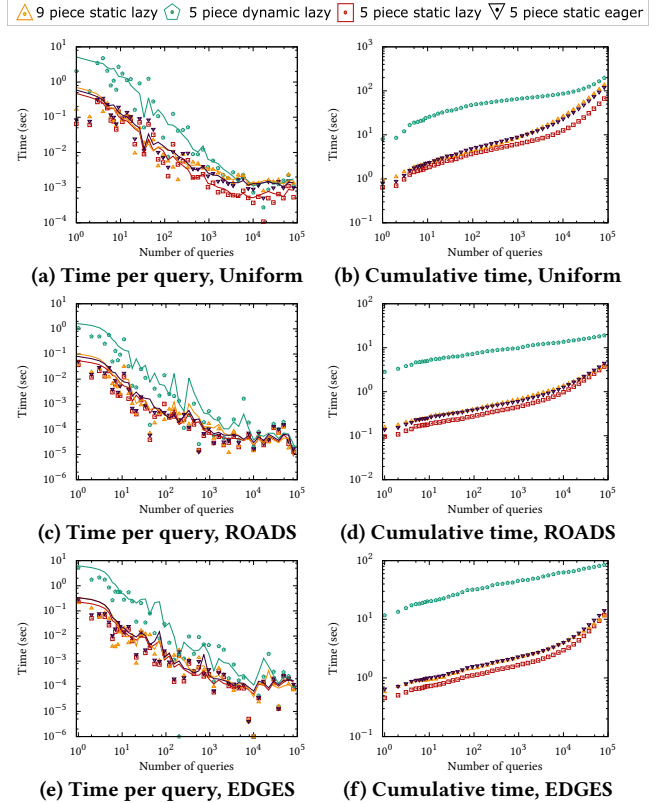**(e) Time per query, EDGES**   **(f) Cumulative time, EDGES**

Figure 13: Ablation study, uniform workload.

4.4 Parameter settings

We first inspect the settings of two AIR-tree parameters: fanout M_f and leaf threshold M_ℓ. We measure the cumulative response time to a workload under different settings on the 64M Uniform synthetic data and the Uniform workload. Table 3 shows the results. We observe that the best-performing set of parameters is fanout 16 and leaf threshold 64; we use these parameters in all subsequent experiments. These parameters diverge from traditionally recommended R-tree parameters in terms of leaf threshold. This divergence is reasonable, since the traditional settings optimize disk I/O, as opposed to main-memory indexing. Indeed, the benefit of scanning a relatively small set of objects per leaf turns out to be worth the cost of traversing the deeper index structure resulting from a smaller leaf threshold. We set the lower threshold m_f on entries an internal node may have to rounded 0.4 of M_f, i.e., 6.
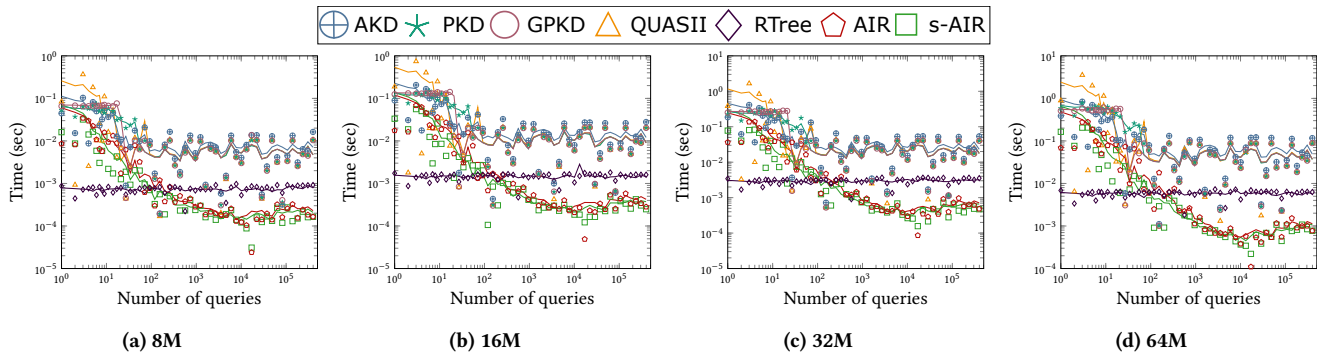
Figure 14: Uniform 2D shape data & workload, time per query.
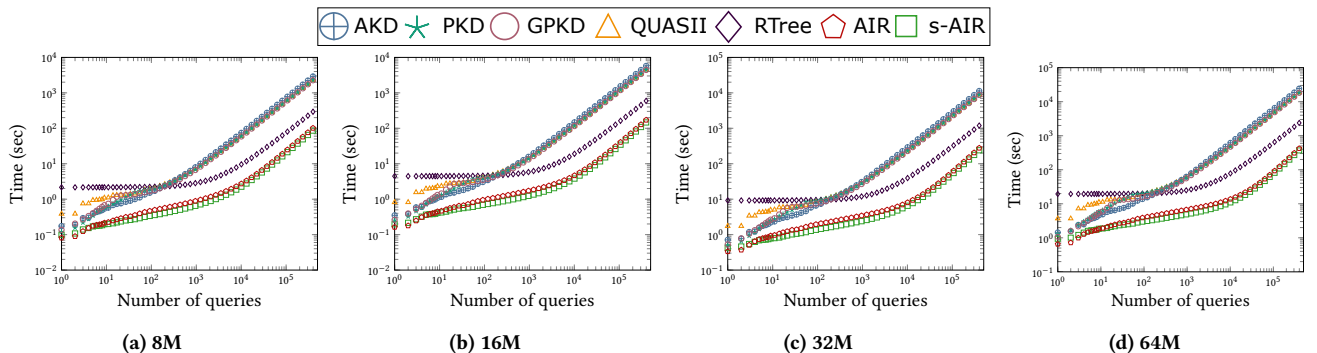

Figure 15: Uniform 2D shape data & workload, cumulative time.

4.5 Ablation study

We now study the effect of AIR-tree design choices: the number of crack pieces created, the data storage scheme, and lazy partitioning.

Number of pieces. We opted to crack in at most $2d + 1$ pieces. Another option is to crack at all query bounds to their full extent, creating a grid, as in Figure 3, resulting in at most 3^d pieces in d dimensions. Figure 13 reports time per query and cumulative time results on a 9-piece static lazy variant, which cracks to 9 pieces in 2D vs. the default 5-piece static lazy one, on the 64M Uniform synthetic, ROADS, and EDGES data with the uniform workload. Observably, cracking to 5 pieces is the dominant strategy.

Static array. Another design choice is to store the data in a *static array A*, with each index leaf pointing to an interval in A and cracking operations swapping elements within A, rather than use a dynamic data structure. Figure 13 also presents results on a 5-piece dynamic lazy variant, which indexes data in leaves by dynamic memory allocation. The static-memory variant vastly outperforms the dynamic-memory one. This dynamic-memory variant only gains traction after the index has converged and cracking actions are no longer performed. As we aim for good performance in the early stages of a workload, we opt for the static-memory AIR-tree.

Lazy boundary maintenance. Lastly, we examine the effect of *lazy boundary maintenance* (Section 3.6). For the sake of scalability, while cracking we update MBB boundaries along the cracking dimension only. We argue that this strategy does not impair query performance. Figure 13 presents results on a 5-piece static eager variant, which maintains exact boundaries, vs. its lazy counterpart. On all data sets, the lazy variant has a clear advantage during

the initial queries, while performances converge later, as the two variants are indistinguishable once leaves become regular.

4.6 Synthetic 2D data

Here, we present experimental results on synthetic 2D data.

4.6.1 Scalability. We first assess the scalability of the AIR-tree against competing methods on *uniform* synthetic 2D data with the *uniform* workload, using subsets of size 8M, 16M, 32M, and 64M. Figure 14 shows the time per query results. By virtue of managing all spatial dimensions concurrently while adapting to the workload, both AIR-tree variants achieve response times faster than QUASII and Cracking KD-tree variants and, after about 200 queries, even faster than the static R-tree. Figure 15 shows cumulative times; the predominant performance of AIR-tree variants is again apparent.

4.6.2 Comparison to static R-tree. Surprisingly, the incrementally built AIR-tree not only reaches, but also surpasses the response times of the pre-built static R-tree on these data. Still, the Boost R-tree we use has fanout 16 for internal and leaf nodes, leading to a tall tree, and does not allow tuning those parameters. To investigate the matter, we tried the Superliminal[7] R-tree, in which we can set the same parameter values as in AIR-tree, i.e., internal node fanout $M_f = 16$ and leaf threshold $M_\ell = 64$. Figure 16 depicts the results, along with those for the Boost R-tree as an R-tree with $M_f = M_\ell = 16$ and AIR-tree variants for each of these settings.

The Boost R-tree implementation builds the index by STR packing [20], whereas the superliminal implementation inserts data one by one following the quadratic split [11], which causes a large

---

[7]Code available at https://superliminal.com/sources/

difference in index-building time. In steady-state query response times, AIR-tree variants outperform their static R-tree counterparts in both cases of parameter settings. This result suggests that the reason for the observed performance gap does not lie with parameter settings; if that were the case, then the AIR-tree response times would converge to those of the respective static R-tree in each case. Since such a convergence does not occur, we conclude that the AIR-tree index, progressively built by crafting tree nodes for query results, is inherently well-adjusted to these synthetic spatial data that follow an exponential size distribution. This finding suggests that the AIR-tree may provide a viable index-building method with lightweight workloads, in addition to quick adaptation.
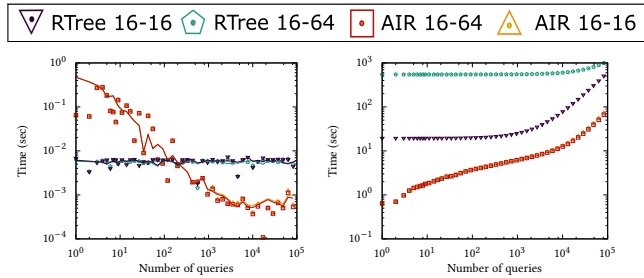

(a) Time per query
(b) Cumulative time

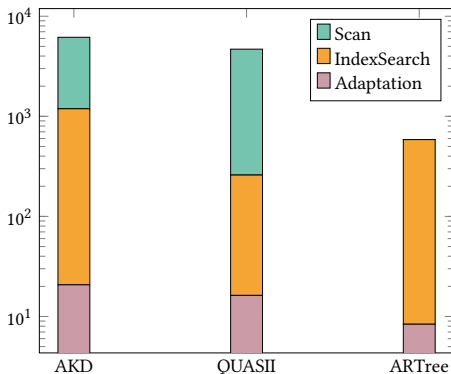**Figure 16: Uniform 2D shape data & workload, R-tree forms.**

Figure 17: Time breakdown on uniform data.

4.6.3 Time breakdown. To further understand the measured cumulative response times on the uniform data and workload, we segment them by distinct type of operation, namely (i) *adaptation*, time to extend the index structure; (ii) *index search*, time to traverse the index; and (iii) *scan*, time to scan through results in leaf nodes. As Figure 17 shows (on a logarithmic scale), QUASII and AKD spend a considerable amount of time scanning, attributed to the additional filtering they perform due to query window extension. By contrast, tree traversal amounts to most of the AIR-tree response time. As a result of building a shallower index, QUASII takes less time for index traversal, albeit more time for other operations.

4.6.4 Sequential workload. We now inspect the performance of the *stochastic* variant (Section 3.7) on the the uniform data set with the *sequential* workload (Figure 12b). Such workloads engender precarious performance with standard cracking [12], which *stochastic* cracking actions aim to assuage. As our results in Figure 18 show, AIR-tree, here represented by its stochastic variant, outstrips competitors throughout the workload. In this case, the per query

performance struggles to converge to that of the fully built index, due to the challenging nature of the sequential workload. Still, the effort pays off eventually over the entire end-to-end workload, answering 1000 queries in less time than the time the static R-tree needs to build its index. The AIR-tree remains the best choice.
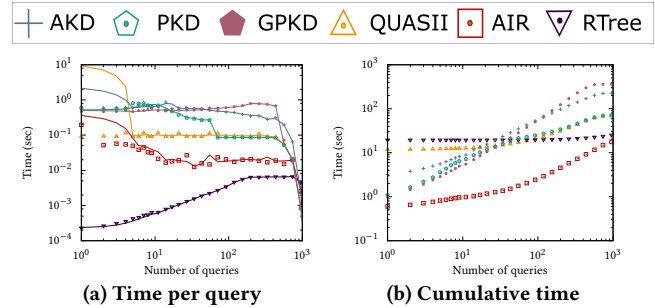

(a) Time per query
(b) Cumulative time

Figure 18: Uniform 2D shape data, sequential workload.

Interestingly, the static R-tree response time grows linearly over the first 100 queries. This is due to the fact that, to generate these uniform data, we choose a lower-left corner *uniformly at random* and expand it to an object MBB; thus, the two ends of the sequential query workload yield fewer results than its middle parts.
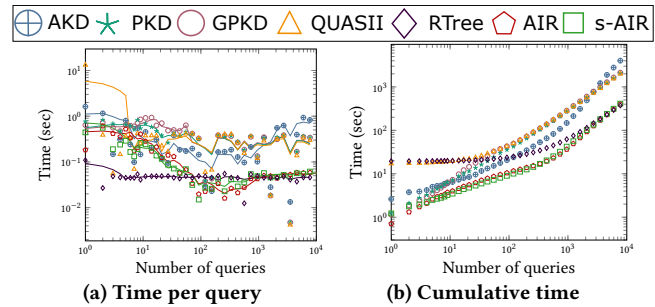

(a) Time per query
(b) Cumulative time

Figure 19: Uniform 2D shape data, partial match workload.

4.6.5 Partial-match workload. We created a *partial match* workload by removing the bounds on a randomly chosen dimension from each of 10K queries from the synthetic workload. Figure 19 shows our results. As these queries yield larger result sets, they take longer. AIR-tree variants still stand out in the first thousand queries.
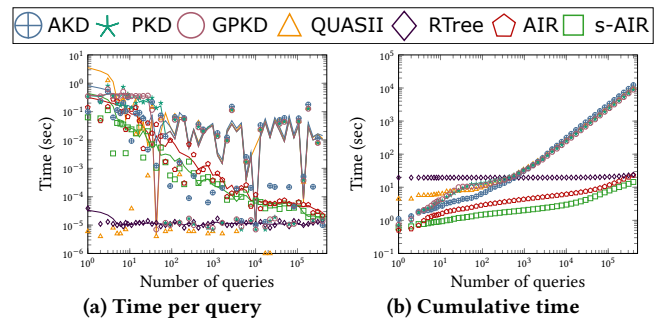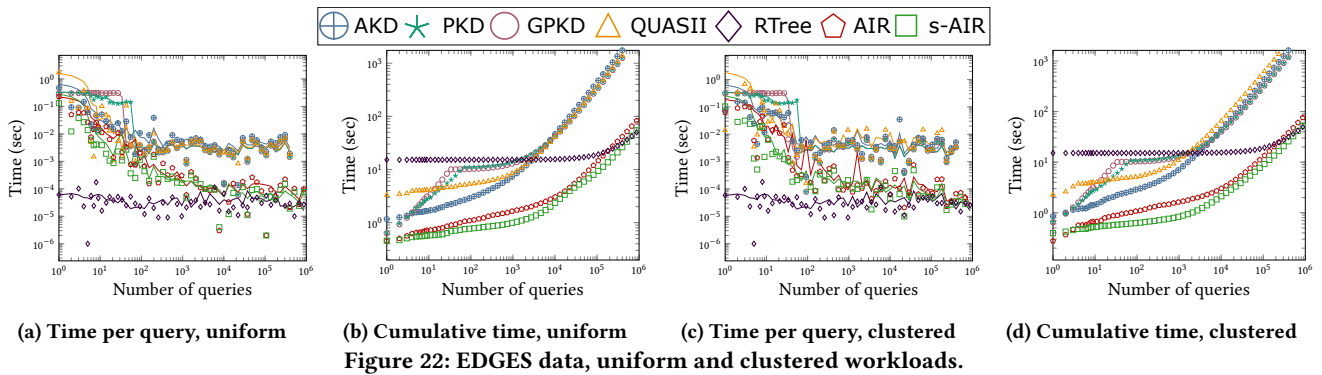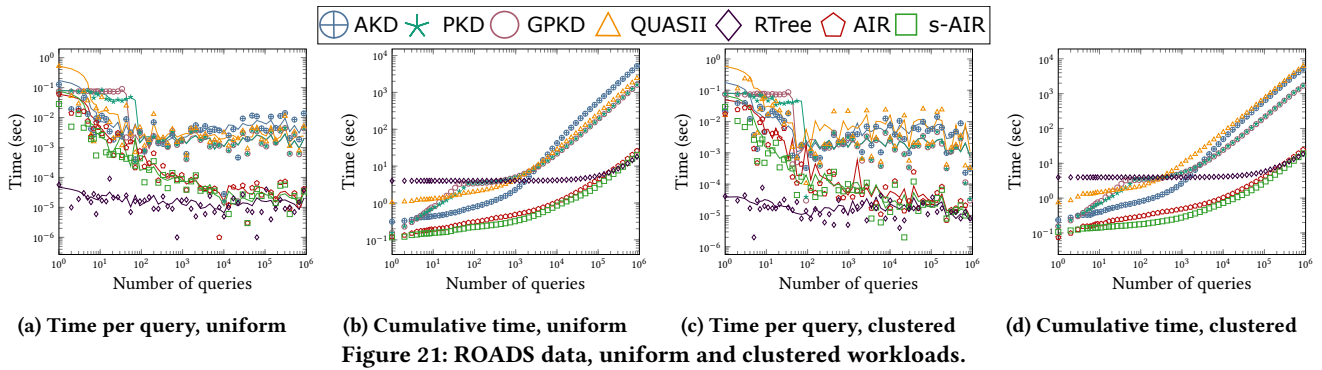

(a) Time per query
(b) Cumulative time

Figure 20: Uniform 2D point data, uniform workload.

4.6.6 Synthetic 2D point data. All preceding experiments were on data sets of objects with spatial extent. The performance advantage of the AIR-tree may yet depend on the nature of data objects, owing to the fact that Cracking KD-trees were proposed for point data. To investigate this matter, we compare all methods on a synthetic point

(a) Time per query, uniform   **(b) Cumulative time, uniform**   **(c) Time per query, clustered**   **(d) Cumulative time, clustered**

Figure 21: ROADS data, uniform and clustered workloads.

(a) Time per query, uniform   **(b) Cumulative time, uniform**   **(c) Time per query, clustered**   **(d) Cumulative time, clustered**

Figure 22: EDGES data, uniform and clustered workloads.

data set: we obtain point data using the lowest coordinates of data objects on all dimensions, and apply the same uniform workload as before. Cracking KD-tree variants treat those data as points, whereas the AIR-tree, R-tree, and QUASII treat them as shapes with no spatial extent. Our results on Figure 20 match those obtained with other data, while the task is easier. The stochastic AIR-tree has the best performance. As the advantage of AIR-tree variants also holds with points, it is not an artifact of the nature of the data.


(a) Time per query   **(b) Cumulative time**

Figure 23: ROADS data, partial match workload.
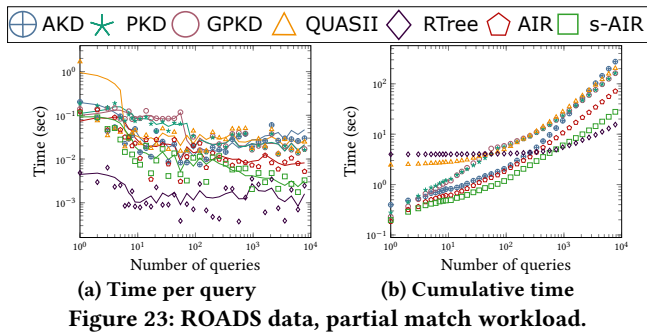
4.7 Real 2D data

ROADS data. Next, we examine performance with real 2D data, starting with the ROADS data. Figure 21 presents results with the *uniform* and *clustered* workloads. In both cases, AIR-tree variants have an advantage over other methods from the very first query and outperform all adaptive methods in cumulative performance, as Figures 21b and 21d show. These results reconfirm our findings with synthetic data. Besides, after more than 100K queries, the cumulative query response time of the default AIR-tree just passes that of the static R-tree, while the stochastic variant retains its advantage.

The per-query performance of AIR-tree variants approaches that of the pre-built index already after 1000 queries, while other adaptive indexes do not reach that response time even after 100K queries. Further, the AIR-tree is robust to different workloads. QUASII does not share in such robustness, as its query response time fails to converge with the clustered workload as quickly as it does with the uniform one. Overall, the AIR-tree achieves an up to 205x speedup in total response time compared to other adaptive methods.

We also tried a *partial match* workload made from the uniform one on this data set, as in Section 4.6.5. Figure 23 shows the results; while this workload is challenging due to large results sets, s-AIR stills answer the first thousand queries competitively vs. the R-tree.

(a) Time per query   **(b) Cumulative time**

Figure 24: LAKES data, synthetic workload.
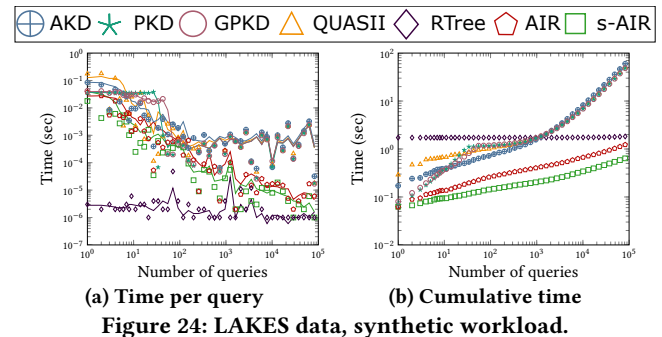
EDGES data. We now examine performance on the EDGES data. Figure 22 shows our results. Once again, AIR-tree variants surpass competitors, while the trends are similar to those we observed with the ROADS data set. These results further corroborate our previous findings. Still, as the EDGES data is twice the size of the ROADS data, convergence occurs later. As with the ROADS data, we observe an up to 43x speedup vs. other adaptive methods.

LAKES data. As discussed in Section 4.2, on the LAKES data we use a workload following the data distribution. Results in Figure 24 present a comport similar to that with other real-world data. AIR-tree variants clearly outperform other adaptive methods and the stochastic variant has an even clearer advantage. Incidentally, Figures 21–24 reveal that the index construction of PKD and GPKD ends around query 50, as the progressive KD-tree completes indexing after a certain number of queries, causing a sudden drop in response times. AKD reaches its best performance more smoothly.
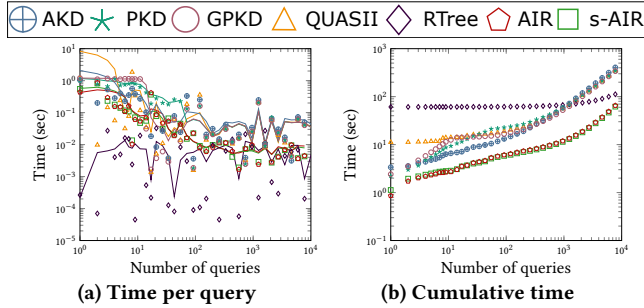

(a) Time per query   (b) Cumulative time

Figure 25: Uniform 3D shape data, uniform workload.

4.8 3D data

4.8.1 Synthetic 3D data. We now turn our attention to 3D data. Figure 25 shows the results on the synthetic 3D data set with the *uniform* workload. The demeanor of all methods resembles that in 2D; AIR-tree variants surpass other adaptive methods in cumulative time with a 6x speedup and matches the performance of a static R-tree within about 100 queries; after 10K queries, AIR-tree cumulative times remain lower than that of the pre-built R-tree. This outcome is reasonable, as the overhead of a cracking operation is independent of dimensionality d thanks to *lazy boundary maintenance*, while the number of cracks grows only linearly with d. We surmise that these trends are preserved in higher dimensions.
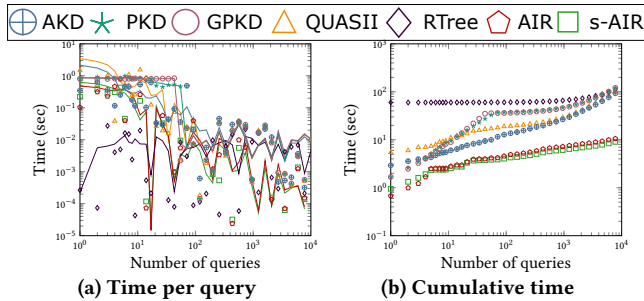

(a) Time per query   (b) Cumulative time

Figure 26: TLC data, uniform workload.

4.8.2 Real 3D data. Lastly, we try the TLC real-world 3D data with a *uniform* workload. In TLC, MBBs reflect pick-up and drop-off locations and timestamps of taxi trips in New York City. Figure 26 shows our results. Again, AIR-tree variants outstrip other methods, with a clear competitive edge after 100 queries and a 10-fold speedup over the whole workload. Due to the nonuniform 3D nature of the data, response times present a discontinuity compared to other experiments, even in the case of the pre-built R-tree. Still, the AIR-tree converges to faster query response times than the static R-tree, reconfirming that it offers not only a competitive adaptive indexing method, but a viable index-building method too.

4.9 Space usage on uniform 2D data

As outlined in Section 4.3, we also measure the evolving size of adaptive indexes. We exclude progressive KD-tree variants, which eventually reach the same size as the corresponding adaptive variant, and, for reference, we plot the size of the static R-tree using the superliminal R-tree implementation. To compare sizes in an implementation-independent manner, we measure the number of nodes in each index. We opted for AIR-tree parameters $M_f = 16$ and $M_\ell = 64$, yet the recommended leaf size for other adaptive methods is 1024 [25]. To create a level playing field, we also use an AIR-tree with $M_\ell = 1024$. Figure 27a shows our node count results on the Uniform 2D synthetic data and workload including separate counts for regular leaves, irregular leaves, and internal nodes for $M_\ell = 64$. The AIR-tree creates more leaves with $M_\ell = 64$ than with $M_\ell = 1024$, while, as reported in Table 3, the total response time for the latter is approximately 32% longer. QUASII ceases creating new nodes quite early and builds the shallowest and widest tree, yet underperforms in efficiency. AKD builds a more sizable index due to its binary structure. The AIR-tree converges last to a stable size, yet achieves smaller size than the static R-tree in both configurations. The number of irregular leaves plateaus and starts decreasing after 10,000 queries as irregular leaves turn regular.
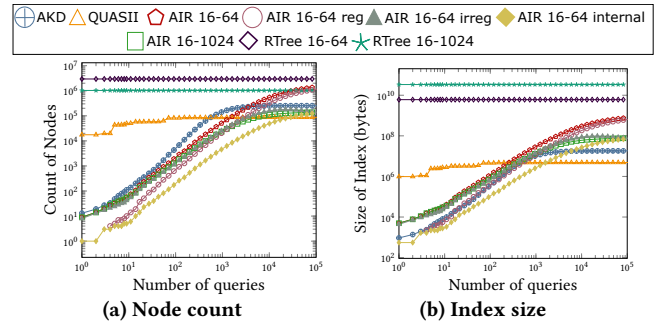

(a) Node count   (b) Index size

Figure 27: Space usage, uniform 2D shape data & workload.

Figure 27b shows the implementation-dependent memory each index structure occupies. The AIR-tree needs less memory than the R-tree, yet more than other adaptive indexes, as it stores MBBs.

5 CONCLUSION

We proposed the AIR-tree, an inherently multidimensional in-memory adaptive index for objects with spatial extent, which is built while processing queries, reapplying criteria for a well-designed spatial index to cracking purposes. Our experimental results on synthetic and real data and workloads in 2D and 3D establish the AIR-tree, especially its stochastic variant, as the method of choice for in-memory multidimensional adaptive indexing. With a lightweight query workload, the AIR-tree also offers a viable method for building an in-memory index. In the future, we aim to extend our methods to handle updates, queries beyond simple range queries, and alternative ways of measuring distance [32].

ACKNOWLEDGMENTS

REFERENCES

[1] Daniar Achakeev, Bernhard Seeger, and Peter Widmayer. Sort-based query-adaptive loading of R-trees. In *CIKM*, pages 2080–2084, 2012.

[2] Ahmed M. Aly, Ahmed R. Mahmood, Mohamed S. Hassan, Walid G. Aref, Mourad Ouzzani, Hazem Elmeleegy, and Thamir Qadah. AQWA: adaptive query-workload-aware partitioning of big spatial data. *Proc. VLDB Endow.*, 8(13):2062–2073, 2015.

[3] Lars Arge, Mark de Berg, Herman J. Haverkort, and Ke Yi. The priority R-tree: A practically efficient and worst-case optimal R-tree. *ACM Trans. Algorithms*, 4(1):9:1–9:30, 2008.

[4] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *SIGMOD*, pages 322–331, 1990.

[5] Norbert Beckmann and Bernhard Seeger. A revised R*-tree in comparison with related index structures. In *SIGMOD*, pages 799–812, 2009.

[6] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509—-517, September 1975.

[7] Jialin Ding, Vikram Nathan, Mohammad Alizadeh, and Tim Kraska. Tsunami: A learned multi-dimensional index for correlated data and skewed workloads. *Proc. VLDB Endow.*, 14(2):74–86, 2020.

[8] Gisbert Dröge and Hans-Jörg Schek. Query-adaptive data space partitioning using variable-size storage clusters. In *SSD*, pages 337–356. Springer, 1993.

[9] Ahmed Eldawy and Mohamed F. Mokbel. SpatialHadoop: A MapReduce Framework for Spatial Data. In *ICDE*, pages 1352–1363, 2015.

[10] Christos Faloutsos, Timos K. Sellis, and Nick Roussopoulos. Analysis of object oriented spatial access methods. In *SIGMOD*, pages 426–439, 1987.

[11] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57, 1984.

[12] Felix Halim, Stratos Idreos, Panagiotis Karras, and Roland H. C. Yap. Stochastic database cracking: Towards robust adaptive indexing in main-memory column-stores. *Proc. VLDB Endow.*, 5(6):502–513, 2012.

[13] Pedro Holanda, Matheus Nerone, Eduardo Cunha de Almeida, and Stefan Manegold. Cracking KD-tree: The first multidimensional adaptive indexing (position paper). In *7th Intl Conf. on Data Science, Technology and Applications DATA*, pages 393–399, 2018.

[14] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. Database cracking. In *CIDR*, pages 68–78, 2007.

[15] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. Self-organizing tuple reconstruction in column stores. In *SIGMOD*, pages 297–308, 2009.

[16] Stratos Idreos, Stefan Manegold, Harumi A. Kuno, and Goetz Graefe. Merging what's cracked, cracking what's merged: Adaptive indexing in main-memory column-stores. *PVLDB*, 4(9):585–597, 2011.

[17] Anders Hammershøj Jensen, Frederik Lauridsen, Fatemeh Zardbani, Stratos Idreos, and Panagiotis Karras. Revisiting multidimensional adaptive indexing. In *EDBT*, pages 469–474, 2021.

[18] Richard M. Karp, Rajeev Motwani, and Prabhakar Raghavan. Deferred data structuring. *SIAM Journal on Computing*, 17(5):883–902, 1988.

[19] Allen Klinger and Charles R. Dyer. Experiments on picture representation using regular decomposition. *Comput. Graph. Image Process.*, 5(1):68–105, 1976.

[20] Scott T. Leutenegger, Jeffrey M. Edgington, and Mario Alberto López. STR: A simple and efficient algorithm for R-tree packing. In *ICDE*, pages 497–506, 1997.

[21] Zhe Li, Man Lung Yiu, and Tsz Nam Chan. PAW: data partitioning meets workload variance. In *ICDE*, pages 123–135, 2022.

[22] Hui Luo, Jingbo Zhou, Zhifeng Bao, Shuangli Li, J. Shane Culpepper, Haochao Ying, Hao Liu, and Hui Xiong. Spatial object recommendation with hints: When spatial granularity matters. In *ACM SIGIR*, pages 781–790. ACM, 2020.

[23] Donald Meagher. Geometric modeling using octree encoding. *Comput. Graph. Image Process.*, 19(2):129–147, 1982.

[24] Vikram Nathan, Jialin Ding, Mohammad Alizadeh, and Tim Kraska. Learning multi-dimensional indexes. In *SIGMOD*, pages 985–1000, 2020.

[25] Matheus Agio Nerone, Pedro Holanda, Eduardo Cunha de Almeida, and Stefan Manegold. Multidimensional adaptive & progressive indexes. In *ICDE*, pages 624–635, 2021.

[26] Sadegh Nobari, Farhan Tauheed, Thomas Heinis, Panagiotis Karras, Stéphane Bressan, and Anastasia Ailamaki. TOUCH: in-memory spatial join by hierarchical data-oriented partitioning. In *SIGMOD*, pages 701–712, 2013.

[27] Mirjana Pavlovic, Darius Sidlauskas, Thomas Heinis, and Anastasia Ailamaki. QUASII: query-aware spatial incremental index. In *EDBT*, pages 325–336, 2018.

[28] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[29] Orestis Polychroniou and Kenneth A. Ross. A comprehensive study of main-memory partitioning and its application to large-scale comparison- and radix-sort. In *SIGMOD*, pages 755–766, 2014.

[30] Timos K. Sellis, Nick Roussopoulos, and Christos Faloutsos. The R+-tree: A dynamic index for multi-dimensional objects. In *VLDB*, pages 507–518, 1987.

[31] Emmanuel Stefanakis, Yannis Theodoridis, Timos K. Sellis, and Yuk-Cheung Lee. Point representation of spatial objects and query window extension: A new technique for spatial access methods. *Int. J. Geogr. Inf. Sci.*, 11(6):529–554, 1997.

[32] Anton Tsitsulin, Marina Munkhoeva, Davide Mottin, Panagiotis Karras, Alexander M. Bronstein, Ivan V. Oseledets, and Emmanuel Müller. The shape of data: Intrinsic distance for data distributions. In *ICLR*, 2020.

[33] Thatcher Ulrich. Loose octrees. In Mark DeLoura, editor, *Game Programming Gems*, pages 444–453. Charles River Media, 2000.

[34] Zongheng Yang, Badrish Chandramouli, Chi Wang, Johannes Gehrke, Yinan Li, Umar Farooq Minhas, Per-Åke Larson, Donald Kossmann, and Rajeev Acharya. Qd-tree: Learning data layouts for big data analytics. In *SIGMOD*, pages 193–208, 2020.

[35] Fatemeh Zardbani, Peyman Afshani, and Panagiotis Karras. Revisiting the theory and practice of database cracking. In *EDBT*, pages 415–418, 2020.