Updating an Adaptive Spatial Index

Fatemeh Zardbani, *Aarhus University*
Konstantinos Lampropoulos, *U. of Ioannina & Aarhus U.*
Nikos Mamoulis, *U. of Ioannina & Athena RC*
Panagiotis Karras, *U. of Copenhagen & Aarhus U.*

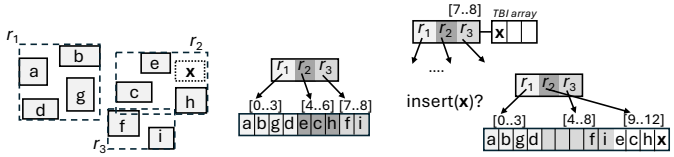
***Abstract*—*Adaptive indexing* allows for the progressive and simultaneous query-driven exploration and indexing of memory-resident data, starting as soon as they become available without upfront indexing. This technique has been so far applied to one-dimensional and multi-dimensional data, as well as to objects with spatial extent arising in geographic information systems. However, existing spatial adaptive indexing methods cater to static data made available in an one-off manner. To date, no spatial adaptive indexing method can ingest data updates interleaved with data exploration. In this paper we introduce GLIDE, a novel method that intertwines the adaptive indexing and incremental updating of a spatial-object data set. GLIDE builds a hierarchical spatial index incrementally in response to queries and also ingests updates judiciously into it. We examine several design choices and settle for a variant that combines gradual self-driven top-down insertions with query-driven indexing operations. In an extensive experimental comparison, we show that GLIDE achieves a lower cumulative cost than upfront-indexing methods and adaptive-indexing baselines.**

***Index Terms*—adaptive indexing, spatial indexing, R-tree**

I. Introduction

Spatial-data management applications periodically collect in bulk spatial objects, such as locations of objects of interest or natural phenomena and require real-time query responses; some of these objects may undergo updates while most remain static across snapshots. Such applications include IoT networks [28], [14], [11], [8], observatories, satellite imaging, management of real estate listings, and environmental data management. Data updates are infrequent and queries may focus on popular regions. In such environments, building a spatial index, such as the popular R-tree [17], [4], [5], upfront for all data would be time-consuming and superfluous, as some spatial data areas may never attract queries, so a large part of the initial and inserted data may need not be accessed.

To address this challenge, past work has introduced *adaptive spatial indexing* methods [34], [19], [33], [39] and adaptive metric-space indexing methods [26] that construct an access method incrementally, in tandem with and as a side-effect of query processing, inspired from counterparts for 1D data [21] By these methods, the cost of queries is initially high yet falls as the index develops. In the 1D case, adaptive indices progressively *crack* the initially unorganized in-memory array of index entries, via an incremental quicksort operation [18], to a growing set of partitions, with each key in a partition being larger than each key in the preceding one. In the multidimensional case, partitions are progressively organized in a tree [19], [39] without the need to preserve a total order. Figure 1(a) illustrates a set of 9 gray-shaded *minimum bounding rectangles* (MBRs) of objects a to i, which are initially in an unorganized array, shown at the bottom of Figure 1(b). In response to three range queries r_1 to r_3 an adaptive R-tree [39] is constructed by swapping array elements to bring together query results. The root of the tree holds query ranges that point to cracked array pieces that hold their results.



(a) MBRs and partitions (b) adaptive R-tree (c) GLIDE update techniques

Fig. 1: Spatial adaptive indexing.

To our knowledge, no existing method handles updates intertwined with queries on an adaptive *spatial* index. The challenge is that all data are packed in a single array, so inserting a new object (e.g., object x) in a leaf would require fitting a new item in a packed subarray (e.g., subarray [4..6] pointed by r_2). Updates on an adaptive index have only been treated in 1D in a rudimentary manner [22]. To fit newly inserted data values relevant to a query in the partitioned 1D array, a *rippling* strategy [22] repeatedly swaps data from one partition to the next, until it reaches a partition outside the query range, whereupon it pushes some data into a temporary log to make space space for the new items. Rippling is imposed by the total order that the cracked array pieces should follow. In multidimensional indexing, there is no requirement that the partitions should follow a total order.

Contribution. Motivated by this, we propose GLIDE, a versatile update management module applicable on any tree-based multidimensional index built by adapting to queries [19], [34], [39], [26]. To design GLIDE, we consider eager vs. lazy and update-driven vs. query-driven options for ingesting query-intertwined data updates into a single data array of contiguous leaf buckets; we opt for a lightweight design that lets inserted data objects gradually trickle down the index, provisionally residing at internal nodes at any level, as illustrated at the top of Figure 1(c). In case an array partition grows too large, GLIDE moves a part to the array's end, leveraging the flexibility to locate data in the multidimensional case, while also introducing *holes* (i.e., empty slots) in the array to efficiently accommodate future insertions, as illustrated at the bottom of Figure 1(c). Our thorough experimental analysis shows that GLIDE robustly offers superior overall performance across query-to-insertion ratios on real-world data sets.

TABLE I: GLIDE vs. other ways to update a spatial index.

	startup cost	query cost	insertion cost	cumulative cost	robustness
R/quad-tree	v. high	low	medium	high	high
SAI+Scan	zero	high→high	v. low	high	low
GLIDE	zero	high→low	low	low	high

Table I positions GLIDE in relation to alternative ways to handle spatial data subject to an unpredictable workload of queries and updates. Classic methods, such as the R-tree and the Quad-tree, have an immense *startup cost* to construct the index before processing a workload. In addition, updating a fully grown index is quite expensive, yielding a high cumulative cost for construction and usage. An alternative, denoted as SAI (Spatial Adaptive Index)+Scan, incrementally constructs an adaptive spatial index on the initially unorganized array (as in previous work [39], [19], [34]), while appending insertions at the end of a separate array, keeping them unorganized. This alternative has low insertion cost, yet its query processing cost remains relatively high throughout the workload, as initial queries apply on an unformed index and later ones have to scan the unorganized inserted data. GLIDE confers the advantages expected from an adaptive index, i.e., zero startup cost and decreasing query cost, while also gradually ingesting insertions with consistently low cumulative cost.

The rest of the paper is organized as follows. Section II reviews the fundamental concepts relevant to spatial adaptive indexing. We discuss the design space of strategies we use in the face of updates in Section III and present our memory organisation strategy in Section IV. Lastly, Section VI outlays a thorough experimental analysis of our proposal in contradistinction to the sate of the art.

II. Related Work

This section overviews fundamentals on spatial indexes, adaptive spatial indexing, and previous work on updates during adaptive indexing.

A. Spatial indexes

The R-tree [17] extends the B^+-tree to multidimensional spaces; it is a balanced tree that hierarchically groups minimum bounding boxes (MBBs) of objects, using heuristics to reduce overlap between MBBs of groups; upon data insertion, it utilizes one of two *node splitting* algorithms, *linear split* and *quadratic split*, to keep MBB enlargement in check. The R^*-tree [4], [5] variant employs enhanced insertion and splitting heuristics. An associated bulk-loading method, sort-tile-recursive (STR) [27], builds such a tree striving to enhance node utilization and reduce empty indexed space, or *dead* space. Aiming at reducing the I/O cost of updates, the R^R-tree [6] consists of a conventional disk-based R-tree paired with a main-memory R-tree, which ingests recent updates; search is applied on both indices, while updates on disk are conducted in bulk when the memory R-tree becomes full. The Quadtree [15] hierarchically divides the space into quadrants and accommodates data in leaves, while ensuring that the number of objects in each leaf does not exceed a maximum capacity constraint. All the above data structures implement insertions *eagerly*; a new object is placed in the leaf that contains it following the most appropriate tree path.

B. Adaptive indexing

Adaptive indexing builds an index in a lazy and progressive manner, during query processing [23], with a minor impact on query response times. An adaptive indexing technique tailored for column-store databases, *database cracking* [21], uses range query bounds as *pivots* to progressively perform steps of the *quicksort* algorithm that divide the array of index entries to partitions, while incrementally building a binary balanced search tree (e.g., an AVL tree) for those partitions. Each query navigates the tree created by prior queries to reach the data values in its range, *cracks* those parts further, up to the resolution of a threshold [21], and expands the tree. Figure 2 shows an example, in which the shaded partition contains the results to range query ($23 \leq A < 27$) on the indexed attribute A; partitions to the left and right of the shaded one contain smaller and larger values, respectively. To ensure robustness, stochastic cracking uses carefully chosen pivots in addition to those specified by queries [18], [38].

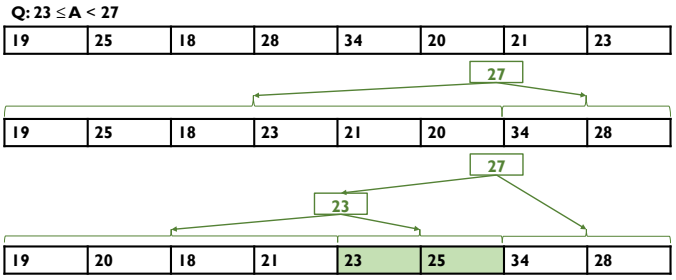


Fig. 2: Database cracking example.

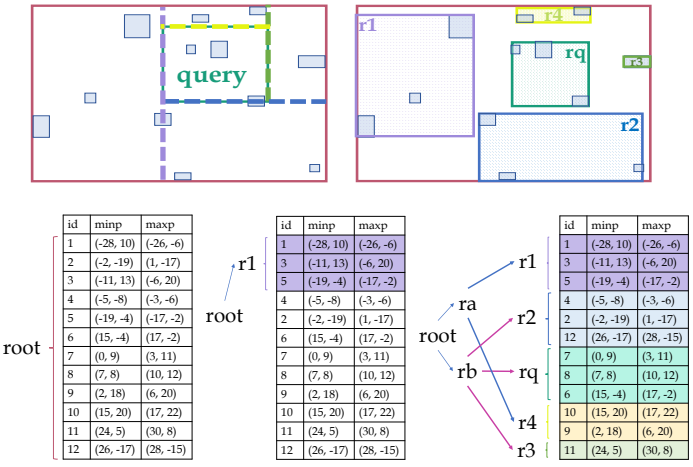


Fig. 3: AIR cracking, spatial and array partitioning [39].

C. Spatial adaptive indexing

The need for responding to spatial queries as soon as the data becomes available led to efforts for *spatial adaptive indexing*. SFCracker [34] transforms multi-dimensional data to one-dimensional using space filling curves. Other efforts [19],

[34] have generalized one-dimensional adaptive indexing to multiple dimensions by handling a different dimension per index level. Those approaches were recently superseded by the *adaptive incremental* R-*tree* (AIR) [39], which builds a compact tree by overseeing all dimensions in each index level and applying quality-aware criteria in splitting and adjusting tree nodes based on query boundaries. AIR commences with an unorganized static data array and progressively organizes queried data areas while responding to queries. It initially comprises a single leaf root enclosing all the data, relaxing the principles of a traditional R-tree, whereby each node holds no more than a predefined number of entries. Leaves of cardinality above a threshold M_ℓ, called *irregular*, are eligible for cracking, while those below M_ℓ, called *regular*, are not cracked further. As the example in Figure 3 shows, in response to a range query, represented by a rectangle, AIR *cracks* the data space through $2d$ hyper-planar cracks (dashed lines), each yielding a spatial partition ($r1$–$r4$), with a remaining partition containing the query results (rq); each crack reorganizes the data array accordingly, as the second row of Figure 3 shows. To avoid the repercussions of a pathologically skewed workload, AIR adds a *stochastic* crack on the largest ensuing piece, totaling at most $2d+2$ pieces in d dimensions. We illustrate this process for a single node fully containing a query range, yet it is applicable on each leaf node that overlaps the query. AIR cracks *irregular* leaves in response to queries, eventually creating *regular* leaves, and evolves into a structure resembling a classic R-tree that outperforms prior multi-dimensional adaptive indexing methods [19], [34] in a variety of workloads across spatial and point multi-dimensional datasets.

D. Updating an adaptive index

An extension of database cracking [22] provisionally stores newly inserted data in a log and triggers their insertion in the data array only once they become relevant to a query by a *rippling* strategy that recursively moves items from one array partition to the next, until it reaches the end of the array or a piece not relevant to the query; in the latter case, it moves some data to the log, to be reinserted in response to future queries. Deletions are also materialized once they become relevant to a query, creating empty array positions (*holes*) that are used whenever possible to accommodate insertions.

E. So-called 'adaptive indexes' in the wild

Several prior works have proposed workload-aware index structures for diverse types of data, which are occasionally described as '*adaptive indexes*'. However, the motivation and intention of those works differs from ours; some of them build a workload-aware index in advance [2], [12], [29], [32]; others first build a regular, non-workload-aware index, and later refine it in response to queries [3], [30], [7]. Besides, the work in [30], enhances throughput by responding to the ratio of queries to updates, while being indifferent to the distribution of queries. Likewise, the work in [7] adjusts an index to accommodate heavy updates. Overall, none of these works *ingests* initially unorganized data into an index in response to queries, as the *adaptive indexing* methods we discuss do.

III. GLIDE

GLIDE is a mechanism that augments any tree-based adaptive spatial index to accommodate updates interleaved with queries. We assume that the data is stored in a single array, with the data items belonging to a leaf residing in contiguous memory space. We investigate our options with respect to *when* and *how* to ingest updates into the index and the data array. Section III-A overviews the design space for GLIDE, considering these issues. We outline strategies for handling insertions in Section III-B and discuss deletions in Section III-C.

A. Design options

Figure 4 arranges the design options we explore along three axes and presents the arising candidates we consider.

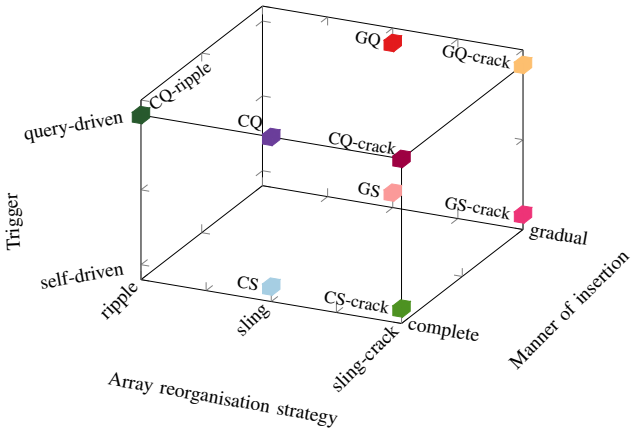


Fig. 4: GLIDE design space.

First, we consider the design choice of what event *triggers* an insertion; we outline two options: by the *self-driven* option, we insert items to the index as they arrive; by the *query-driven* option, we keep insertions in a separate global list and materialize them only once they become relevant to a query. Second, we consider two options for how we materialize a triggered insertion; in the *complete* manner, we fully traverse the index and directly enter newly inserted items into the tree leaves; in the *gradual* manner, we accommodate inserted data in separate, pre-reserved spaces in each tree node, and distribute them in bulk among the node's children once they exceed the size threshold. Lastly, we consider three options on how to reorganize the data in the single array, which we describe in Section IV.

B. Handling insertions

1) Complete self-driven (CS): The *complete self-driven* (CS) insertion strategy processes each insertion fully *upon arrival*, positioning new data in the appropriate place in the tree structure, reorganizing the static array as necessary (see Sec. IV).

2) Complete query-driven (CQ): The *complete query-driven* (CQ) insertion strategy appends each received insertion, temporarily, in a log structure, separate from the index. Each query scans the log, retrieves query-relevant objects, and fully inserts them into the tree in the same manner as the CS strategy does. Insertion is conducted while traversing the tree for query-answering purposes; at each internal node we assign each insertion item to the appropriate child by the tree insertion heuristic, ultimately placing each new object to ts corresponding leaf node.

3) Gradual self-driven (GS): The *gradual self-driven* (GS) strategy, like CS, introduces each data insertion directly into the tree *upon arrival*. However, instead of immediately placing new data objects to leaves by completely traversing the tree, GS lets them *gradually* trickle down the tree, allowing internal tree nodes to temporarily store data objects, called *spares*, up to an empirically determined size threshold θ_s. When the amount of spares in an internal tree node exceeds θ_s, GS *diffuses* them to children nodes. Diffusion recursively propagates downwards from children, if they lack the space to accommodate the new items.

Algorithm 1 outlines the recursive gradual insertion method for a batch of new items to be inserted tbi in a $node$. If there is not enough space in $node$ for its spares plus tbi, diffusion takes place for all these items (Line 5). We create a tbi list of items for each child node and recursive call Algorithm 1 for each child (Line 12). This allows subsequently inserted items to be assigned well and query-driven cracking to operate on an up-to-date index. At a leaf that cannot fit the set of objects to be inserted among its spares, we introduce all accumulated spares and items to be inserted into the data array (Line 14), a process to be discussed in Sec. IV.

Algorithm 1 Gradual insertion

```
1:  procedure INSERT-GRAD(node, tbi)
2:      if node.spares.size + tbi.size < θ_s then
3:          place tbi in node.spares
4:          return
5:      if node is internal then
6:          diffused_tbi = [] for each node.children
7:          for item in node.spares ∪ tbi do
8:              b = PICKBRANCH(item, node.children) [17]
9:              diffused_tbi[b].append(item)
10:         for child in node.children do
11:             if |diffused_tbi[child]| > 0 then
12:                 INSERT-GRAD(child, diffused_tbi[child])
13:     else
14:         REORGANISE(node, node.spares + tbi)
15:         return
```

We show an example of diffusion in a generic spatial-index tree in Figure 5, assuming the tree has a θ_s threshold of 4 spare items per node. At the outset, the root holds items $\{1, 2, 3\}$, node n_1 holds $\{4, 5\}$ and node n_2 holds $\{6\}$ as spares. First, we try to insert item 7. As there is space in the root's spares, we keep it there. Then we try to insert item 8. Now, there is no more space in the root's spares, hence we *diffuse* each of the root's spares and the new item down the tree, based on the tree insertion algorithm. For instance, we assign item 1 to node n_2; as there is enough space among its spares, this branch of the recursion ends here. Item 3 is assigned to one of the other children. However, items $\{2, 7, 8\}$ are assigned to node n_1, which has no spare space to accommodate them; thus, we diffuse all its spares and let each of them find its place in the next tree level.

GS requires extra precaution when processing a query. During tree traversal, we need to scan the spares of each node to retrieve any arising query results.

4) Gradual query-driven (GQ): As a fourth possible combination of our design choices, the *gradual query-driven* strategy lets insertions be triggered by queries, as CQ does (Section III-B2), yet follows a *gradual* manner of insertion, like GS does (Section III-B3). We store each arriving insertion in the unstructured log of pending insertions and, upon the arrival of a query, we scan the log to identify query-relevant objects and insert them *gradually* into the tree. We let internal nodes store a limited number of *spare* items and employ the diffusion method of Section III-B3; diffusion occurs as a side-effect of the query-driven tree traversal.

C. Deletions: complete self-driven

GLIDE handles deletions in a simple manner; it locates the object to be deleted by tree search, swaps it with the first non-empty slot in its leaf (or list of spares), and increments the number of empty slots, creating space for future insertions. We ignore underflows of the minimum capacity.

IV. Reorganizing the static array

Any data item ingested into the index structure is stored either in the *spare* space of a tree node or in the global, statically allocated array. The entries of a leaf node occupy contiguous memory blocks in the same array partition, facilitating efficient query evaluation. Still, this leaf node design was intended for static data. GLIDE caters to dynamic insertions and deletions at arbitrary positions of the array while retaining the contiguity of data within a partition. To this end, GLIDE adds the following features to this basic design:

- the *beginning* of a leaf partition may have unused slots, or *holes*, as in [22]; we keep a counter h of such holes per leaf; hence, the contents of a leaf with range $[s, e]$ are at array positions $s + h$ to e.
- tree leaves keep linked-list pointers to their left and right adjacent leaves in the array; we note that such adjacent leaves do not necessarily have any spatial relation.

We devise two policies that deal with a set of items tbi that are to be inserted to a leaf with insufficient space: (1) the *ripple* and (2) the *sling* strategy. Rippling cascades excessive items to neighboring leaves as necessary, while slinging moves the entire overflown leaf to the end of the array. In addition, we present an enhancement on the sling policy that cracks a large leaf and moves one or both pieces to the end of the array.

A. The ripple strategy

The *ripple* strategy is driven by queries and presupposes a log of pending insertions, hence can be used with query-driven strategies that utilize such a log, i.e., CQ and GQ. When

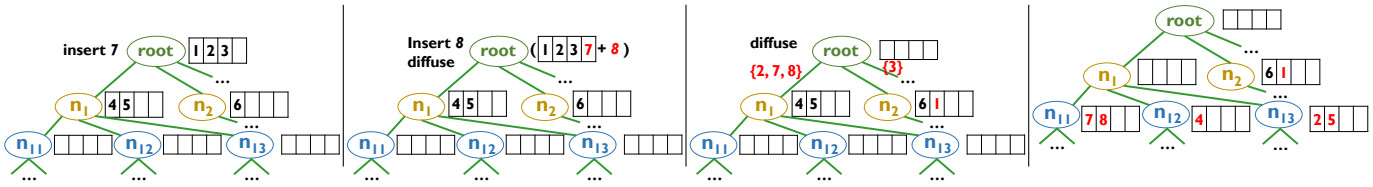


Fig. 5: Diffusion example, tree structure.

processing a query, we denote leaves and items that overlap it as *hot* and those who do not as *cold*. We aim to keep hot items in the array but allow cold ones to leave the array to facilitate an early termination of the process. In contradistinction to a similar method proposed for indexing one-dimensional scalar attributes in column stores [22], when indexing multidimensional spatial data, array partitions corresponding to index leaf nodes do *not* need to obey a particular order. Thus, we may move leaves around liberally, taking in consideration their current position in the array and data insertion requirements.

We consider that data is stored in a static array with available space for new data on its end. Under this design, we first outline the actions taken in some simple cases of inserting k objects into a target leaf's partition in the array:

1) If we insert data to the last leaf in the array, then we append the data directly to the *end of the array*.
2) If the data to be inserted fits into the *holes* at the beginning of the leaf, then we place the data there directly.
3) If k *exceeds* the size of the target leaf l, then we move the leaf along with its new contents to the array's end in $k+l < 2k$ operations; for it would take $3k$ operations to move k items from the array to the temporary log, enter k items in the array, and later return the removed k items from the log to the array.

The above cases notwithstanding, we apply the *ripple* method when inserting a number of new items of size smaller than the target leaf size. To make space for the items to be inserted, we start from that target leaf and cascade across array partitions as in [22] (Section II-D), occupying any available holes and shifting items from the start of a partition to its end, taking over space from the next partition. Once we arrive at a *cold* leaf node, we halt the process and push a sufficient amount of data to the log. The rationale for this measure is that we are interested to keep *hot* data, relevant to the current query, in the array, but may shift cold data out of it. Otherwise, if we arrive at the end of the array and remain in the *hot* area, we expand the last leaf as necessary.

Fig. 6 illustrates the ripple strategy, assuming that the numbers are identifiers of spatial objects. Holes are denoted by an X. Suppose that we insert items $\{15, 16, 17\}$ into leaf i. Consider that leaves i and ii overlap the current query (i.e., they are *hot*), while leaf iii does not (i.e., it is *cold*). Leaf i has one hole, wherein we place object 15. Then, we use the hole of the next leaf ii to accommodate $\{16\}$. Now only item 17 is to be placed. As leaf ii is *hot*, we *ripple* its contents, i.e., item 11 forward to the end of leaf ii and beginning of leaf iii and adjust the range boundaries of leaves i and ii accordingly. As leaf iii is *cold*, i.e., does not overlap the query, we cease rippling, eject cold item 13 from leaf iii into the pending insertions log (to stay there until some other query drives it back into the tree structure). Notably, by this *rippling* method, the size of the pending insertion log fluctuates.

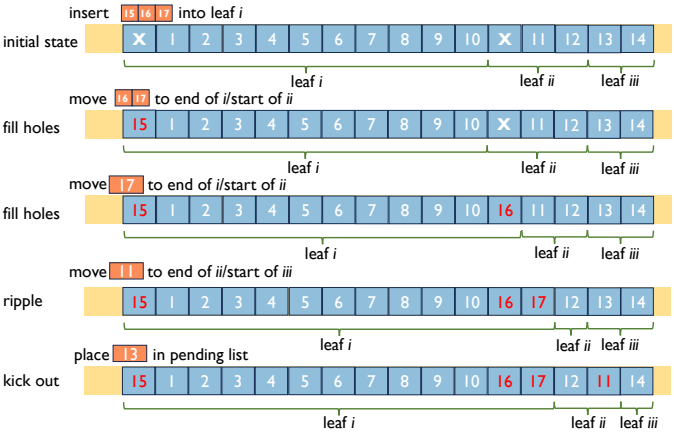


Fig. 6: Ripple reorganisation strategy.

B. *The* sling *strategy*

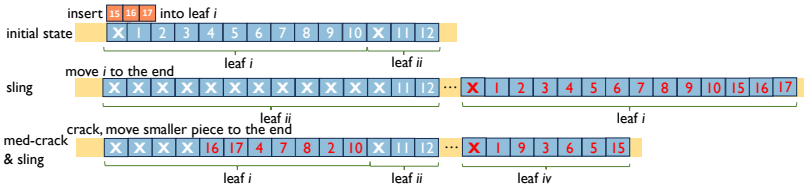
The ripple strategy does not fully exploit the lack of a total order in a multidimensional index and therefore incurs a substantial overhead. We propose an alternative approach that makes space for insertions by deliberately leveraging the flexibility to relocate leaves in the data array.

Algorithm 2 outlines our *sling* strategy. First, in case the target leaf is the last leaf in the array, we append the inserted data to its end (Line 2). Likewise, if there is enough available space in a leaf's *holes*, we avail of them for the insertion (Line 4). Otherwise, the sling strategy *ejects* the entire leaf to the end of the array alongside the new data and offers the empty space created by the move as holes to the next leaf (Line 8). For the sake of robustness, we endow the moved leaf with a certain amount of *default holes* Δ_H (Line 9).

Figure 7 illustrates the sling method with an example. We insert items $\{15, 16, 17\}$ into leaf i, which is not the last leaf and does not have enough holes to accommodate the new items. We then move leaf i along with the items to be inserted *tbi* to the space available at the end of the array, along with one default hole, and assign the available space left behind as holes to leaf ii.

C. *Sling with a crack*

By the sling strategy, an insertion to a large leaf may cause superfluous movement of data in the array and leave behind a lot of empty space. To ameliorate this effect, we amend the sling strategy with a *stochastic cracking* [18] step, replacing



Algorithm 2 Sling

```
1:  procedure SLING(node, tbi)
2:      if leaf is right-most then
3:          append tbi to end of array
4:      else if leaf.holes > tbi.size then
5:          fill holes with tbi
6:          reduce leaf.holes by tbi.size
7:      else
8:          set leaf.size holes at the start of leaf.right_sibling
9:          move leaf to array's end with Δ_H holes
10:         append tbi to leaf's end
```

Fig. 7: Sling reorganisation strategy: plain, with mediocre crack, with quantile crack.

Lines 8–10 in Algorithm 2 by Algorithm 3. When inserting to a leaf larger than twice the regular leaf size threshold, $2M_\ell$, we crack it on a *mediocre*, i.e., the median of a small number of samples [38], to split it into two approximately equal pieces (Line 4); if at least one of those pieces leaves behind enough space for the data to be inserted, we move the smallest such piece to the array's end; if none of them leaves behind enough space, we move both to the array's end, keeping in check the amount of moved data and the memory space occupied by the data including holes.

Algorithm 3 Crack Upon insertion

```
1:  if leaf.size > 2M_ℓ then
2:      sca = longest axis of leaf area                    ▷ stochastic crack axis
3:      scp = mediocre on sca                              ▷ stochastic crack pivot
4:      lp, rp = crack leaf on scp value in sca axis       ▷ left and right piece
5:      if tbi.size > lp.size ∧ tbi.size > rp.size then    ▷ neither fits
6:          move lp with Δ_H holes and tbi.size extra space to array's end
7:          move rp with Δ_H holes to array's end
8:      else if (tbi.size ≤ lp.size ∧ lp.size ≤ rp.size) ∨
9:              tbi.size > rp.size then                    ▷ lp is smallest fitting
10:         move lp with Δ_H holes to array's end
11:     else if (tbi.size ≤ rp.size ∧ rp.size ≤ lp.size) ∨
12:             tbi.size > lp.size then                    ▷ rp is smallest fitting
13:         move rp with Δ_H holes to array's end
14:     scan tbi and place values in lp or rp using scp pivot
15: else
16:     set leaf.size holes at the start of leaf.right_sibling
17:     move leaf to array's end with Δ_H holes
18:     append tbi to leaf's end
```

We also crack the items in *tbi* on the same pivot and distribute them among the two leaf pieces. As they may all be assigned to one of the two pieces, we need to ensure there is adequate space to accommodate them. In the worst case, neither of the pieces is big enough to accommodate *tbi* in the space it leaves behind (Line 5); in that case, we move both pieces to the array's end. In the most fortuitous case, the smallest piece we create is large enough to accommodate *tbi* in the space it leaves behind; then we move the smallest piece to the array's end and safely distribute *tbi* among the two pieces (Lines 8 and 11). If the smaller piece is not large enough to accommodate *tbi* in the space it leaves behind, we move the larger piece to the array's end (Lines 9 and 12).

The third row in Figure 7 shows an instance of the sling method with cracking in action. Considering leaf i as larger than $2M_\ell$, we choose a spatial axis and pivot as the median of 3 samples taken from the leaf, and crack items in the leaf's partition thereby. Assume items $\{1, 9, 3, 6, 5\}$ fall on the one (left) side of the pivot and items $\{4, 7, 8, 2, 10\}$ on the other (right) side by our spatial cracking criterion. As the two pieces have equal size and the *tbi* items fit in it, we move the first of the two pieces to the array's end. We then crack *tbi* items on the same pivot. Let item $\{15\}$ fall on the left side and the rest on the right side. We leverage the holes created by slinging $\{1, 9, 3, 6, 5\}$ to the array's end to place *tbi* alongside each cracked partition, creating the new leaf iv, which we add to the tree structure with one initial hole ($\Delta_H = 1$). We experimented with more sophisticated choices for the cracking pivot, but did not find a better option.

We combine the designs of Section III with the reorganization strategies discussed here to create indexing methods. Standard methods use the *sling* strategy (Section IV-B) without extra cracking, denoted with the suffix *-sling*. We refer to combinations with the *ripple* reorganization strategy (Section IV-A) by the suffix *-ripple and to those with the *sling* strategy (Section IV-B) with a mediocre crack by the suffix *-crack.

V. Theoretical Analysis

We analyse the cost of insertions and queries separately, and combine results. An analysis of adaptive indexing in 1D is available in [38]; its results are applicable, mutatis mutandis, to the multidimensional case. In adaptive tree indexes, the query response comprises *index traversal* and *index extension*. In the 1D case, traversal is done on a binary search tree with logarithmic complexity. In balanced trees, queries need $O(\log N + T)$ operations for N data objects and T query results. As GLIDE is implementable on top of any adaptive tree index, let the expected tree traversal operations be $\Psi(N, d, T)$ for queries on N d-dimensional data objects, each yielding up to T results. Also, let the expected total number of operations for index adaption be $\Gamma(N, r, d)$. Then the expected number of operations is $\lambda(N, r, d, T) = \Gamma(N, r, d) + r\Psi(N, d, T)$.

The complete self-driven (CS) design follows the same querying strategy, and will therefore perform the same amount of query operations. For the GS design, scanning the spares in each accessed node will add some overhead resulting in $\lambda(N, r) = \Gamma(N, r, d) + r\frac{\theta_s}{2}\Psi(N, r, d)$ operations in the expectation, where θ_s is the space available for spares in a node, assuming half the spares are occupied when visited.

Insertions in the CS design comprise tree traversal as well as array reorganisation efforts. For the array reorganization,

we need to move pieces of size $N/2^i$ on average in the ith insertion. For a total of ω insertions, we then need to perform $\Phi(N,\omega) = \omega\Psi(N,r,d) + N(1-\frac{1}{2}^{\omega-1})$ operations in expectation. In the GS design, these expressions capture the worst case, as traversals terminate early and some items never get inserted into the array. The traversal cost for some items is amortized over several insertions, but the accumulated cost remains the same. Adding the extra stochastic crack upon insertion to the methods, i.e. CS-slingCrack and GS-slingCrack, would only add a small constant factor to the second term, as the moved pieces go through a partitioning but the amount of copying may become smaller. Therefore the total cost of a workload of r queries and ω insertions on N shapes adds up to $\lambda(N,r)+\Phi(N,\omega)$ operations in expectation.

VI. Experimental Analysis

A. Implementation

AIR. We implemented[1] all different design options on top of a query-adaptive R-tree, AIR [39]. We tailored the GLIDE module to this particular structure as follows.

First, by all candidate designs (§III-A), new data do not enter not only the data array, but also the tree structure. Regarding leaf-splitting, if the leaf that receives the inserted data is *irregular* (§II-C), then we simply allow the index to accommodate the new data. If the leaf is *regular*, i.e., has size below the cracking threshold, and in effect exceeds that threshold, then we switch the leaf to *irregular*, allowing it to be *cracked* again by future query-driven cracking operations. Thereby, the index structure accepts updates with little overhead. As we trickle down the tree on the path to the location where newly inserted data is to be introduced, we extend the MBB of each encountered node to accommodate that data.

Second, when cracking a leaf node by AIR [39] procedures, we disperse its spares among the ensuing pieces by the R-tree insertion heuristic [17], always choosing the cracked piece that undergoes the least area enlargement.

Third, regarding deletions, we deliberately do not tighten MBBs to reflect deletions, to make the procedure more lightweight, in the same spirit as lazy-update R-trees [25]. As deletions are relatively rare compared to queries and insertions, we anticipate that eschewing the tightening of MBBs benefits overall GLIDE performance; node MBBs subjected to deletions are eventually tightened due to queries and insertions.

AV-tree. To evaluate the generality of the proposed methods, we apply them on the Adaptive Vantage Tree (AV-tree) [26], an adaptive index structure designed for indexing high-dimensional data in metric spaces. The AV-tree partitions the space around query centers into units defined by hyperspheres, utilizing distance bounds. It supports both range and k-nearest neighbor (kNN) queries. Since we evaluate range queries on AIR, we focus on kNN queries on the AV-tree to cover more of the spectrum of query types.

The query-driven design presumes an absolute criterion of query relevance, such as belonging to a given range, to guide query-driven insertion. Such a criterion does not apply to kNN queries, where one object's query relevance depends on other data objects being near neighbors to the query center. Therefore, we have only implemented the self-driven designs for this index. Inserting items in to an AV-tree is inherently simpler when compared to AIR, as no changes are required to be made to the tree structure, and the tree grows top-down.

[1]Code available at https://github.com/fatemeh-zardbani/GLIDE

B. Experimental setup

We conduct an extensive experimental study to evaluate GLIDE using real and synthetic data on realistic workloads. We implemented GLIDE and competitors 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.

Performance measures. Following the common practice in prior work [19], [34], [20], [21], [22], we measure the progressively evolving response time during the workload. In terms of response times, we measure: (i) the cost per query over a workload, averaged over 5 runs; and (ii) the cumulative cost, which aggregates the cost per query over a workload; including the creation time for static indices. For the sake of fairness, all methods perform identical count range queries. Insertion times inevitably fluctuate, as some trigger cascading diffusion; to visualize results comprehensibly, we add a continuous moving-average line in those plots, with window size 30. In cases where the full workload progression offers no new insights, we show only the time to evaluate the entire workload.

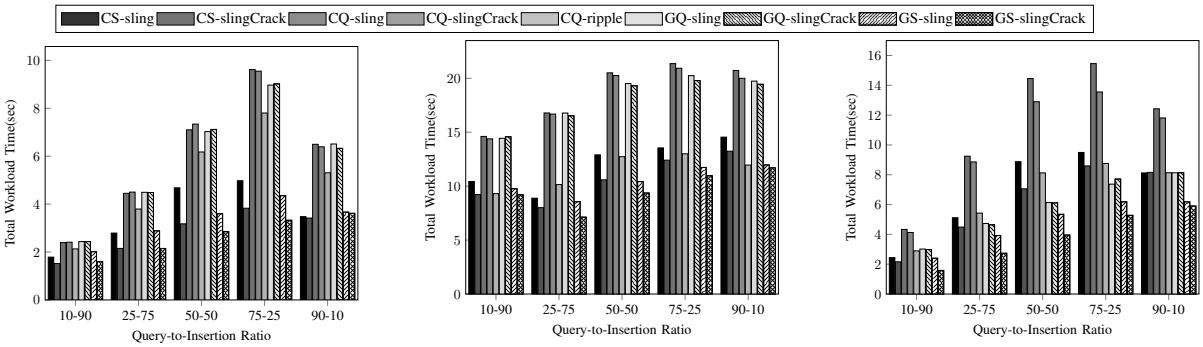
TABLE II: Data sets.

Name	Synth	ROADS	EDGES	BUILD'S	TLC	MNIST
Size	64M	19M	70M	115M	153M	70k
Dim.	2	2	2	2	3	50

Datasets. In experiments where GLIDE is applied on AIR, our focus is on the adaptive indexing of spatial objects. We generated a large 2D synthetic dataset of 64M rectangular shapes using SpiderWeb [24]. The location, as well as the width and height of these objects adhere to a uniform distribution within the [0, 1] and [0,0.01] range, respectively. We also experiment with publicly available 2D and 3D real datasets:[2] ROADS and EDGES from the US Census Bureau and BUILDINGS from OpenStreetMap. The ROADS data feature shapes of U.S. roads and the EDGES data comprise of lines on the U.S. map, including roads, rivers, and borders. The BUILDINGS dataset is comprised of the boundaries of all buildings worldwide. We also use a real-world 3D data set of taxi cab trip records[3] from year 2010 normalising pick-up and drop-off longitudes, latitudes, and timestamps to represent 3D boxes. Finally, we use the popular MNIST[10][1], database of handwritten digits, to test the application of GLIDE on AV-tree. We used UMAP [31] to reduce their dimensionality down to 50. Table II summarizes data characteristics.

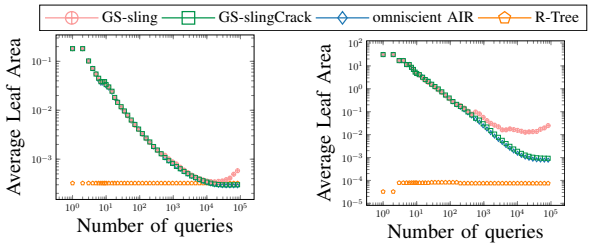
[2]http://spatialhadoop.cs.umn.edu/datasets.html at the University of Minnesota [13]

[3]https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page



(a) ROADS (b) BUILDINGS (c) Synthetic

Fig. 8: Ablation study on range workloads.



(a) Synthetic, Leaf Areas (b) ROADS, Leaf Areas

Fig. 9: Average Leaf Areas on range workloads

TABLE III: Query workloads.

Location distribution	*Size distribution*	*Source*	*Size*
Uniform	Uniform	Synthetic	100k
Zipfian	Uniform	Synthetic	100k

C. Workloads

We measure time per query and cumulative time over workloads of queries intertwined with insertions. For the range workloads, we use **query workloads** consisting of at most 100K rectangular queries placed according to either a Uniform distribution or a Zipfian distribution with $\alpha = 4$ using Python's scikit-learn [35] module. Table III summarises workload characteristics. As the default option, with the ROADS and EDGES datasets, we tailor each query extent so that it has a result size in the order of 0.001% (e-3%) of all data objects. With the BUILDINGS dataset, which features locations and shapes of buildings, to create queries of the desired selectivity on built areas, we select at most 100K random objects from the data and extend their width and height. While we use a default query selectivity of e-3%, we also look into the effect of the query result size by investigating other values: e-4%, e-2%, and e-1%. We interleave queries with insertions as follows.

Each workload performs 100K actions encompassing a shuffled combination of insertions and range queries, with a set ratio between the amount of queries and insertions. We draw the inspiration for this class the YCSB benchmark workload type E [9], a well-established Yahoo! data management system benchmark. The default ratio of queries to insertions is 75% range queries to 25% insertions (i.e., 75K queries and 25K insertions). To draw data for insertion, we set aside a properly sized subset of the data set at hand *apart* from the fixed data set sizes reported in Table II. Moreover, as the query-to-insertion ratio is seldom known a priori. To affirm robustness with respect to variations in that ratio, we examine a range of query-to-insertion ratios other than the default one. We dub these query-and-update workloads, ***action ratio*** workloads.

For the kNN workloads on the MNIST data, the queries are samples of the dataset. As the dataset is fairly small, we could not set aside part of it to add later; and as it is clustered and in a meaningful distribution, we could not synthetically generate new data to insert. So, we insert a sample of the data as duplicates. The interleaving of the actions are done in the same manner as described for the range workloads but the size of the workload is kept to 10k given the dataset size.

TABLE IV: Parameters, uniform 2D shape data, 25% inserted.

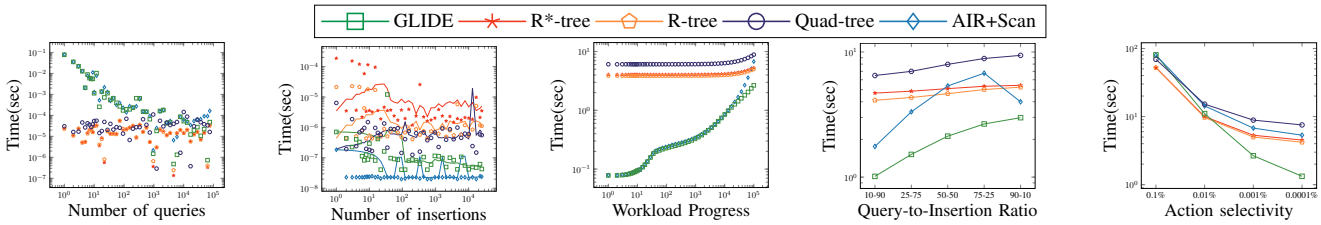
Q-to-I ratio	25-75			50-50			75-25		
Δ_H \ θ_s	8	16	32	8	16	32	8	16	32
0	3.47	3.62	3.75	5.106	5.29	5.41	6.53	6.68	7.01
32	3.53	3.64	3.70	5.22	5.20	5.35	6.613	6.92	6.88
64	3.53	3.59	3.65	5.04	5.14	5.47	6.47	6.81	6.94

D. Parameter Tuning

We use the range workloads on AIR to investigate the parameter values. Firstly, we investigate the choice of values for GLIDE parameters: regular leaf size threshold M_ℓ, tree fan-out f, default number of holes Δ_H, and limit of spare items stored in nodes θ_s. For the first two, we use the values found best in [39], i.e., $M_\ell = 64$ and $f = 16$. Regarding Δ_H and θ_s, Table IV shows the total time (in seconds) spent by GLIDE on the whole Uniform workload of queries and insertions of various ratios with different configurations. Following these results, we choose to allow 64 holes on moved leaves ($\Delta_H = 64$) and to up to $\theta_s = 8$ spare items in each internal node, by virtue of the dependability of these values. We stress that the performance of GLIDE is not overly sensitive to parameter values. For the kNN workloads in the AV-tree we use the same Δ_H and θ_s values, and let the cracking threshold parameter, θ, to be set as 128 as suggested in [26].

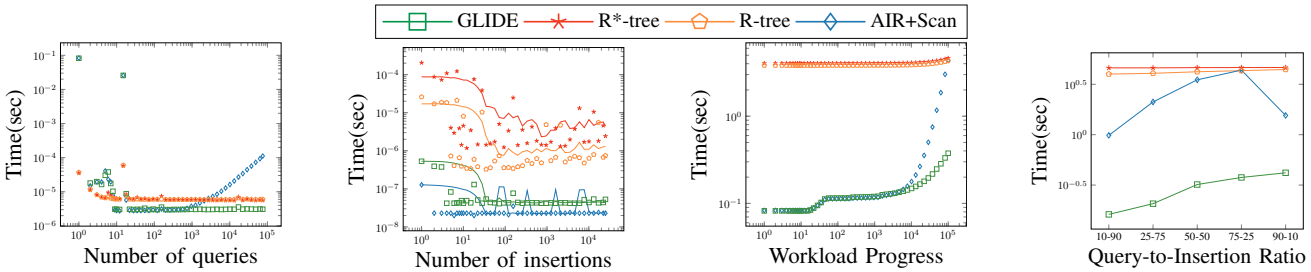
E. Ablation study

We perform analysis of the designs on the range workloads implemented on the adaptive R-tree. We compare GLIDE variants differentiated by the trigger and manner of insertions and



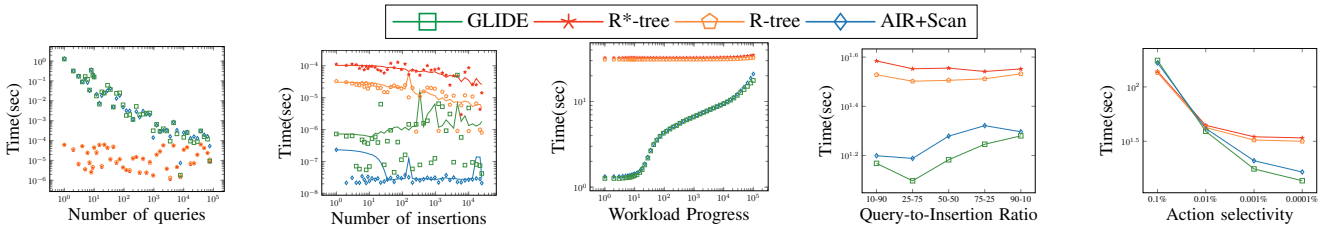
(a) 75-25, e-3%: per query (b) 75-25, e-3%: per insert (c) 75-25, e-3%: cumulative (d) Total workload time (e) Total workload time

Fig. 10: ROADS dataset, Uniform range queries.



(a) 75-25: time per query (b) 75-25: time per insertion (c) 75-25: cumulative time (d) Total workload time

Fig. 11: ROADS dataset, Zipfian range queries.



(a) 75-25, e-3%: per query (b) 75-25, e-3%: per insert (c) 75-25, e-3%: cumulative (d) Total Workload time (e) Total Workload time

Fig. 12: BUILDINGS dataset, Uniform range queries.

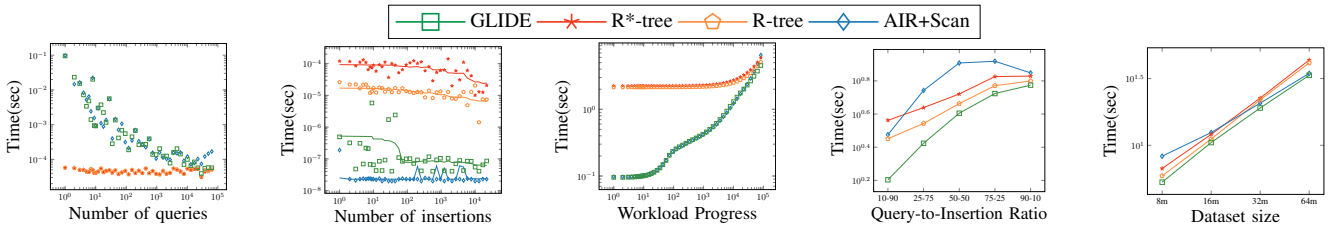
the array reorganisation strategy. We use each on a synthetic dataset of 8M objects, the ROADS data, and the BUILDINGS data, with workloads of varying action ratios represented on the x-axis. We use a shuffled mix of queries and insertions as described in § VI-C with ratio of 75 to 25 respectively. We compare the total workload runtime of the variants presented in Section III, including CQ-sling, CQ-slingCrack, CQ-ripple, GQ-sling, GQ-slingCrack, GS-sling, GS-slingCrack, CS-sling, and CS-slingCrack.

Figure 8 presents our results. Observe that GS-slingCrack achieves the best cumulative time in insertion-heavy workloads and proves to be robust against various action ratios. Query-driven methods are all burdened with keeping an extra array for *pending* new items, which they need to scan with each query, whereby query-driven GLIDE variants fully or gradually insert the related items into the index. In such query-driven approaches, insertions are cheap whereas range queries can be expensive. One might intuitively expect such variants to perform worse on insertion-heavy workloads, as they need to scan through more inserted items; however, such workloads invoke fewer queries that warrant those expensive scans, causing performance to deteriorate as the query-to-insertion ratio tilts towards the query-heavy side; performance relapses after the unsorted list becomes small enough that its scans are less burdensome. Still, GS-slingCrack presents the most dependable performance regardless of workload arrangement.

We observe that the extra stochastic crack upon insertions drastically improves the performance of GS-sling. To understand this phenomenon, we perform the following experiment: We measure the average area of tree leaves as the workload progresses vs. those of an R-tree receiving insertions using the Superliminal[4] R-tree implementation, which allows setting fan-out 16 and leaf size threshold 64, as in GLIDE, and, for reference, a version of AIR with all data pre-loaded, denoted as *omniscient* AIR, which does not face data insertions. Figures 9a and 9b show our results on the synthetic shape data and ROADS data, respectively. Notably, in both datasets GS reaches large leaf areas after the 100th action, while GS-slingCrack keeps leaf sizes checked, as it mostly cracks leaves that receive insertions, and reaches average leaf size as small as the R-tree, and slightly larger than AIR, which is privileged in this comparison.

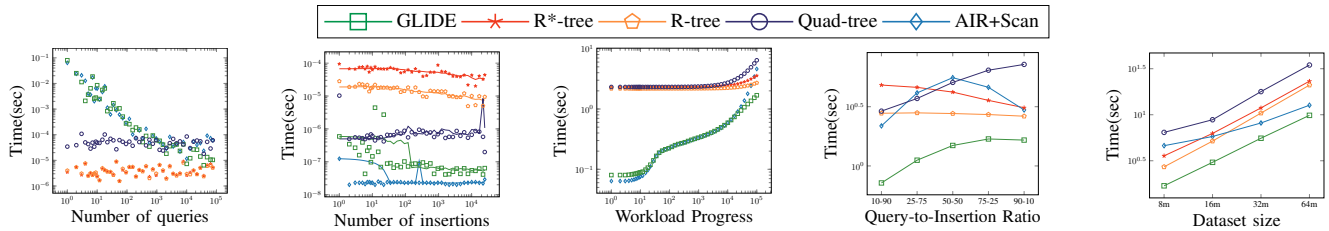
Summary. In insertion costs, the query-driven design with gradual insertion (GQ) is best. In query time, the self-driven design with complete insertion (CQ) is best. The gradual self-driven design with an extra crack upon insertion (GS-

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



(a) 8M : per query (b) 8M : per insertion (c) 8M : cumulative time (d) Total workload time (e) Total workload time

Fig. 13: Uniform 2D shape data, uniform 75-25 range workload.



(a) 8M: per query (b) 8M: per insertion (c) 8M: cumulative (d) Total workload time (e) Total workload time

Fig. 14: Uniform 2D point data, uniform 75-25 range workload.

slingCrack) is the most robust option, performing best in terms of total throughput in the plethora of experimental settings we have tried.

F. Range workloads comparative study

As our main baseline, we include a naive extension of AIR with a simplistic array, denoted as AIR+Scan, which appends inserted data to a separate log-like structure without an index. When we evaluate a query, we also scan this auxiliary data structure to retrieve results from the inserted data. Besides AIR+Scan, we compare GLIDE against implementations of the following methods:

- A static in-memory R-tree with quadratic split;
- A static in-memory R*-tree;
- A static in-memory Quad-tree.

Both static tree implementations were taken from the Boost[5] library. We set the leaf size of R-tree variants at 16, as recommended, and conduct the comparison on different workloads of queries and updates applied to our synthetic and real datasets. The Quad-tree [15] implementation is provided in [36].

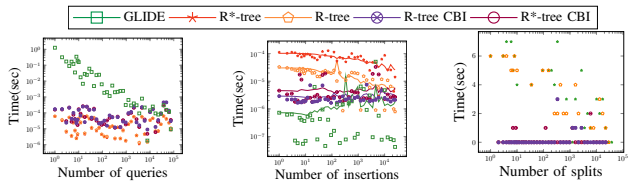
We organize the remainder of this section as follows: We evaluate GLIDE under varying action-ratios (§VI-F1), different dataset sizes (§VI-F2), and query selectivity (§VI-F3). Next, we assess performance on point data (§VI-F4) and examine how the system handles workloads that include deletions (§VI-F5). We also investigate behavior when the order of range queries is pathologically sequential (diagonal in space) (§VI-F6). Lastly, we examine performance on 3D datasets (§VI-F7).

1) Varying Query-to-Insertion ratio: We use shuffled action-ratio workloads that contain 100K actions, as defined in Section VI-C. To study the index robustness across different environments, we test a range of ratios between range queries and insertions in the workload, intertwining insertions into Uniform and Zipfian queries on the ROADS data-set. Figures 10 and 11 show our results with the x-axes of 10d and 11d representing action ratios. To illustrate workload progression, Figures 10a, 10b and 10c display the trend of action times for 75-25 action ratio workloads as an example, and Figure 10d shows the total workload time across different ratios. The observed decreasing per query times of AIR and GLIDE are typical for adaptive indexes. As Quad-tree is a space-partitioning index, it handles dynamic insertions better than the data-partitioning R-tree variants. However, that minor advantage does not counterbalance other costs in the overall workload time; this behaviour persists across different action ratios, as Figure 10d shows; given these results, we exclude Quad-tree from subsequent shape-data experiments.

Figures 11a, 11b, and 11c present the progression of the 75-25 ratio workload with Zipfian queries, while Figure 11d shows the trend of total workload times. In a Zipfian workload certain areas are queried more often than others, hence GLIDE builds a compact index with a few regular leaves thoroughly indexing oft-queried areas and a few irregular leaves that are rarely accessed. It achieves shorter times per query (Figure 11a) and hence cumulative times. The AIR index behaves similarly, yet the extra work of scanning the newly-arrived data eventually becomes cumbersome. The insertion performance of GLIDE also outpaces classic R-tree methods (Figure 11b), superseding their burdensome index creation time and slower query times to result in a striking improvement in the total workload time (Figure 11c). This effect remains unabated by the ratio of the actions in the workload, as Figure 11d reveals. The trend for AIR+Scan is similar to the one observed in the ablation studies for the query-driven designs, as discussed in Section VI-E, and exhibits the same trends across ratios for Uniform and Zipfian queries. Figures 13d and 14d replicate this study on synthetic shape data and point data (cf. Section VI-F4), with analogous results.

[5]Code available at https://www.boost.org/users/history/version_1_61_0.html

Figure 12 illustrates that GLIDE preserves its advantages on the BUILDINGS data, with a realistic workload and across various ratios. To inspect the separate costs more thoroughly, we present those times decoupled in Figure 15. Notably, as Figure 15b shows, insertion times of GLIDE are lower than those of the classic solutions. Moreover, the insertion times for both static R-tree variants start at high values and decrease as the workload progresses. To understand this *front-loaded* behaviour, we measured the insertion time of pre-built R-tree and R*-tree variants that are initialized on the data set by inserting data items *one-by-one*, instead of using the default Sort-Tile-Recursive (STR) bulk-loading [27] method; we denote these variants as CBI (**c**reated **b**y **i**nsertion). As Figure 15b shows, CBI variants exhibit stable, rather than front-loaded, insertion cost. We infer that the STR bulk-loading method builds *packed* trees that initially necessitate intensive leaf-splitting to accommodate insertions, while the space created by such initial splits suffices to absorb insertions later in the workload, depicted in Figure 15c. Previous work has used the term *waves of misery* [37] for this front-loaded, in general *oscillating*, behavior of indexes.



(a) Query time (b) Insertion time (c) Splits

Fig. 15: Buildings data, 75-25 per query decoupled time.

Summary. Inspecting the behaviour of GLIDE under various distributions, i.e. different ratios between the count of insertions and queries, we observe that it performs most robustly with superior total workload time.

2) Varying data size: We also compared all methods, for both shape and point data (cf. Section VI-F4), using different synthetic data set sizes: 8, 16, 32, and 64 million. Figures 13a, 13b, 14a, and 14b plot the decoupled per query and per insertion times for each case, for the 8M size experiment, while Figures 13c and 14c show the respective cumulative times to tackle the workload, which remain favorable to GLIDE throughout the 100K actions. Other dataset sizes follow similar trends. Most pertinently, as the results in Figures 13e and 14e illustrate, GLIDE scales well with dataset size, indicated on the x-axis, with both shape data and points. All methods display a linear growth of cumulative time in response to dataset size. AIR with linear scan has a less steep ascent, yet that is only due to the fact that it always scans at most 25K insertion items, while other methods manage growing indexes in their insertions; this advantage is only an artifact of our specific experimental design, which keeps the number of insertions stable and only lets the initial data size grow. Naturally, it does not scale with a growing insertion workload.

Exhibiting the typical behavior of indexes assembled by adaptation to queries, GLIDE starts out with high per-query cost that follows a descending trend. Eventually GLIDE reaches the per-query performance of the R-tree while maintaining a lower cumulative time even by 10^5 queries. Besides, while the AIR index starts out with a performance similar to GLIDE, in later stages of the workload the burden of linearly scanning the inserted data escalates, raising the cost per query. Regarding insertion costs, appending each new item to an unordered list, i.e., the modus operandi of AIR with linear scan, is the quickest insertion strategy. On the other hand, R-tree variants present divergent insertion costs, as the R*-tree is tailored to reduce node overlaps, yet that design feature deteriorates its insertion performance compared to the standard R-tree. By virtue of its gradual insertion scheme, GLIDE achieves average insertion times almost one order of magnitude lower than classic R-tree variants throughout the workload. Moreover, the insertion cost of GLIDE descends, because as the tree grows, it offers more *spare* spaces.

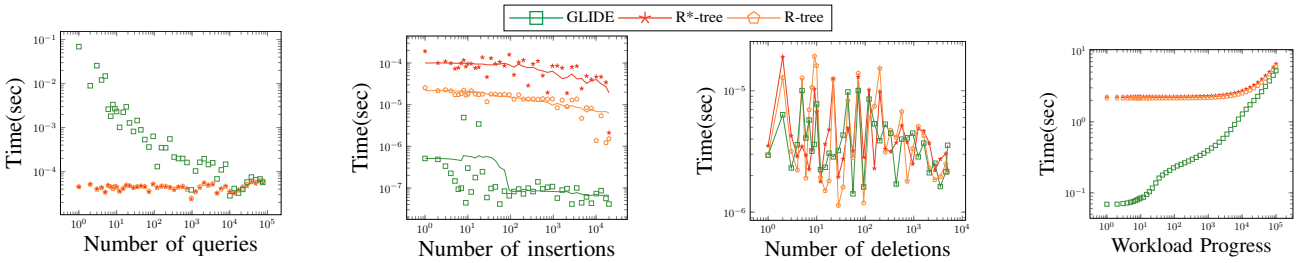
Summary. We find that GLIDE performs competitively in total workload time under growing dataset size.

3) Varying query selectivity: We now study the robustness of GLIDE to the size of query results using the real-world ROADS and BUILDINGS data sets with 75% shuffled action ratio workload. We tune selectivity to the order of magnitude of 0.0001%, 0.001% (which is the default), 0.01%, and 0.1%. As the size of the query results grows, all indexes register longer total times, as Figures 10e and 12e show with the x-axis representing selectivity. The higher the selectivity, the smaller the difference in cumulative times; this behavior was also observed in [16]. Still, under realistic query result sizes, GLIDE has a clear advantage.

4) Point datasets: As mentioned, to demonstrate the generality of our methods, we conducted experiments with point data sets. We use a synthetic 2D point dataset and validate GLIDE's robustness to workload distribution and dataset size. We reinstate the Quad-tree in this experiment, as it is designed for point data. Figure 14 shows the resulting trends, which are similar to those obtained with shape data, with the Quad-tree showing a slight improvement.

5) Effect of deletions: We apply expanded workloads that also include deletions on the synthetic data set. We find the object to delete in the index using its geometry; if the id of the object is given, we use it to access its geometry. To adhere to standard realistic workloads, we design a workload comprising 75% of range queries, 20% of insertions, and 5% of deletions, in shuffled order. Figure 16 presents our results, decoupling the query (16a), insertion (16b) and deletion times (16c), and also displays the progression of the overall workload time (16d). In all cases, GLIDE gracefully integrates deletions in its operation.

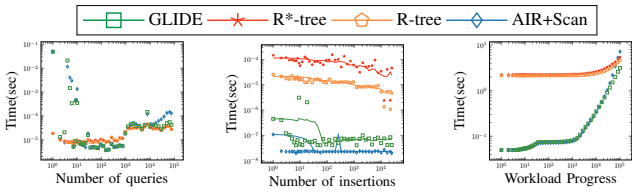
6) A pathological workload: Adaptive indexing methods are vulnerable to query workloads that explore the data space in a skewed manner, especially by a *sequential* pattern [18], [39]. Here, we study such a synthetic workload of queries ordered across a diagonal line in the 2D space intertwined with insertions; we let query extents follow a Uniform distribution in the range of [0, 0.005] to create the default target selectivity of 0.001%, and interleave them with insertions to create a



(a) per query (b) per insertion (c) per deletion (d) cumulative time

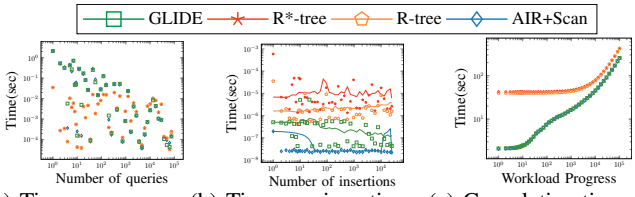
Fig. 16: Synthetic shape data, 75-20-05, deletion

75% shuffled action-ratio workload. Figure 17 depicts our results. GLIDE reaches a steady-state performance with per-query time in the same order of magnitude as the R-tree in fewer than 1000 queries, while maintaining a cumulative-time advantage. This achievement is due to both the *stochastic* cracks created upon range queries and the *extra* cracks created upon insertions, as explained in Section IV-B. Other methods follow the trends observed in preceding experiments.



(a) Time per query (b) Time per insertion (c) Cumulative time

Fig. 17: Uniform 2D shape data, 75-25, sequential queries



(a) Time per query (b) Time per insertion (c) Cumulative time

Fig. 18: TLC data, 75-25, Uniform queries.

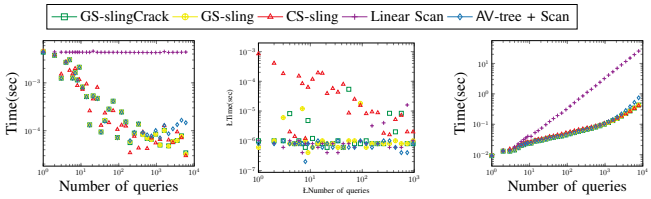
7) 3D data: As in previous work [39], we focus on objects with spatial extent, which naturally occur in two or three dimensions. In the 3D space, we experimented with TLC. Figure 18 depicts our results, which corroborate GLIDE's resilience with respect to dimensionality. Owing to the small number of insertions relative to the data set's size in this case, AIR+Scan achieves performance close to GLIDE.

G. kNN workloads comparative study

We let the action ratio be 75-25 queries to insertions and investigate the results of searching for 20 nearest neighbors. As our main baselines, we include (i) a simple Linear Scan, and (ii) a naive extension of AV-tree with a simplistic array, denoted as AV-tree+Scan. We include the first as high-dimensional indexes are susceptible to the curse of dimensionality, which a brute-force algorithm may avoid. The second baseline appends inserted data to a separate log structure. When evaluating a query, we also scan this auxiliary data structure to retrieve results from inserted data. Figure 19 depicts these results.

The observed decreasing per query times of AV-tree and GLIDE are typical for adaptive indexes. For the AV-tree, the extra scanning of newly-arrived data eventually becomes cumbersome, as evinced in the per-query and cumulative plots. Overall, the GLIDE design of gradual insertion performs best.

We observe that the extra crack on gradual insertion with *sling* has a negligible effect. We found that this is due to the large tree height, which allows for many *spare* spaces in the nodes and thus hardly lets insertions ever reach the leaf level to be inserted in the array, where an extra crack could make a difference. This is a result of the binary-ness, and hence tall height of the tree. On the other hand, the declining trend in the per-insertion CS-sling times is due to *sling* operations, as initial insertions incur many slings and later ones exploit the created holes. Still, due to the tree height, complete insertion designs cost more than gradual insertion ones.



(a) Time per query (b) Time per insertion (c) Cumulative time

Fig. 19: MNIST data, 75-25, 20NN workload.

VII. Conclusion

We proposed GLIDE, an update mechanism applicable to adaptive in-memory indexes for multi-dimensional objects. While the index is built in response to queries, it absorbs data insertions as they arrive, remaining up to date. We investigated the design space and arrived at a design that introduces insertions directly into the structure, allows them to gradually progress down as they accumulate, and reorganizes the underlying data array in response by moving and cracking partitions; we extended the design to manage deletions as well. Through a comprehensive experimental analysis on synthetic and real multi-dimensional data, we validated that GLIDE outperforms both patchwork extensions of previous adaptive indexing solutions to accommodate updates and static indexes when responding to the same workloads.

ACKNOWLEDGMENTS

Work supported by AUFF, DFF, and project MIS 5154714 of the National Recovery and Resilience Plan Greece 2.0 funded by the EU under the NextGenerationEU Program.

REFERENCES

[1] http://yann.lecun.com/exdb/mnist/. [Accessed 10-Mar-2023].

[2] D. Achakeev, B. Seeger, and P. Widmayer. Sort-based query-adaptive loading of R-trees. In *CIKM*, pages 2080–2084, 2012.

[3] A. M. Aly, A. R. Mahmood, M. S. Hassan, W. G. Aref, M. Ouzzani, H. Elmeleegy, and T. Qadah. AQWA: adaptive query-workload-aware partitioning of big spatial data. *Proc. VLDB Endow.*, 8(13):2062–2073, 2015.

[4] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The r*-tree: An efficient and robust access method for points and rectangles. In *SIGMOD*, pages 322–331, 1990.

[5] N. Beckmann and B. Seeger. A revised r*-tree in comparison with related index structures. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, page 799812, New York, NY, USA, 2009. Association for Computing Machinery.

[6] L. Biveinis, S. Saltenis, and C. S. Jensen. Main-memory operation buffering for efficient r-tree update. In *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*, pages 591–602, 2007.

[7] H. Cha, X. Hao, T. Wang, H. Zhang, A. Akella, and X. Yu. Blink-hash: An adaptive hybrid index for in-memory time-series databases. *Proc. VLDB Endow.*, 16(6):1235–1248, feb 2023.

[8] N. Chaudhry, M. M. Yousaf, and M. T. Khan. Indexing of real time geospatial data by iot enabled devices: Opportunities, challenges and design considerations. *Journal of Ambient Intelligence and Smart Environments*, 12(4):281–312, 2020.

[9] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *SoCC*, pages 143–154, 2010.

[10] L. Deng. The MNIST database of handwritten digit images for machine learning research [best of the web]. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

[11] Q.-T. Doan, A. S. M. Kayes, W. Rahayu, and K. Nguyen. A framework for iot streaming data indexing and query optimization. *IEEE Sensors Journal*, 22(14):14436–14447, 2022.

[12] G. Dröge and H. Schek. Query-adaptive data space partitioning using variable-size storage clusters. In *SSD*, pages 337–356. Springer, 1993.

[13] A. Eldawy and M. F. Mokbel. Spatialhadoop: A mapreduce framework for spatial data. In *ICDE*, pages 1352–1363, 2015.

[14] Y. Fathy, P. Barnaghi, and R. Tafazolli. Large-scale indexing, discovery, and ranking for the internet of things (iot). *ACM Comput. Surv.*, 51(2), 2018.

[15] R. Finkel and J. Bentley. Quad trees: A data structure for retrieval on composite keys. *Acta Inf.*, 4:1–9, 03 1974.

[16] T. Gu, K. Feng, G. Cong, C. Long, Z. Wang, and S. Wang. The rlr-tree: A reinforcement learning based r-tree for spatial data. *Proc. ACM Manag. Data*, 1(1), 2023.

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

[18] F. Halim, S. Idreos, P. Karras, and R. H. C. Yap. Stochastic database cracking: Towards robust adaptive indexing in main-memory column-stores. *Proc. VLDB Endow.*, 5(6):502–513, 2012.

[19] P. Holanda, M. Nerone, E. C. de Almeida, and S. Manegold. Cracking KD-tree: The first multidimensional adaptive indexing (position paper). In *7th International Conference on Data Science, Technology and Applications, DATA*, pages 393–399, 2018.

[20] S. Idreos. *Database Cracking: Towards Auto-tuning Database Kernels*. PhD thesis, CWI, 2010.

[21] S. Idreos, M. L. Kersten, and S. Manegold. Database cracking. In *CIDR*, pages 68–78, 2007.

[22] S. Idreos, M. L. Kersten, and S. Manegold. Updating a cracked database. In *SIGMOD*, pages 413–424, 2007.

[23] R. M. Karp, R. Motwani, and P. Raghavan. Deferred data structuring. *SIAM Journal on Computing*, 17(5):883–902, 1988.

[24] P. Katiyar, T. Vu, A. Eldawy, S. Migliorini, and A. Belussi. Spiderweb: A spatial data generator on the web. In *SIGSPATIAL*, pages 465–468, 2020.

[25] D. Kwon, S. Lee, and S. Lee. Indexing the current positions of moving objects using the lazy update r-tree. In *Proceedings of the Third International Conference on Mobile Data Management (MDM 2002), Singapore, January 8-11, 2002*, pages 113–120, 2002.

[26] K. Lampropoulos, F. Zardbani, N. Mamoulis, and P. Karras. Adaptive indexing in high-dimensional metric spaces. *Proc. VLDB Endow.*, 16(10):2525–2537, 2023.

[27] S. T. Leutenegger, J. M. Edgington, and M. A. López. STR: A simple and efficient algorithm for r-tree packing. In *ICDE*, pages 497–506, 1997.

[28] M. Liu, D. Li, Q. Chen, J. Zhou, K. Meng, and S. Zhang. Sensor information retrieval from internet of things: Representation and indexing. *IEEE Access*, 6:36509–36521, 2018.

[29] H. Luo, J. Zhou, Z. Bao, S. Li, J. S. Culpepper, H. Ying, H. Liu, and H. Xiong. Spatial object recommendation with hints: When spatial granularity matters. In *ACM SIGIR*, pages 781–790. ACM, 2020.

[30] S. Luo, B. Kao, G. Li, J. Hu, R. Cheng, and Y. Zheng. Toain: a throughput optimizing adaptive index for answering dynamic $k$nn queries on road networks. *Proc. VLDB Endow.*, 11(5):594–606, jan 2018.

[31] L. McInnes, J. Healy, and J. Melville. UMAP: uniform manifold approximation and projection for dimension reduction. *CoRR*, abs/1802.03426, 2018.

[32] V. Nathan, J. Ding, M. Alizadeh, and T. Kraska. Learning multi-dimensional indexes. In *SIGMOD*, pages 985–1000, 2020.

[33] M. A. Nerone, P. Holanda, E. C. de Almeida, and S. Manegold. Multidimensional adaptive & progressive indexes. In *ICDE*, pages 624–635, 2021.

[34] M. Pavlovic, D. Sidlauskas, T. Heinis, and A. Ailamaki. QUASII: query-aware spatial incremental index. In *EDBT*, pages 325–336, 2018.

[35] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vander-Plas, 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.

[36] D. Tsitsigkos, K. Lampropoulos, P. Bouros, N. Mamoulis, and M. Terrovitis. A two-layer partitioning for non-point spatial data, 2021.

[37] L. Xing, E. Lee, T. An, B.-C. Chu, A. Mahmood, A. M. Aly, J. Wang, and W. G. Aref. An experimental evaluation and investigation of waves of misery in r-trees. *Proc. VLDB Endow.*, 15(3):478–490, nov 2021.

[38] F. Zardbani, P. Afshani, and P. Karras. Revisiting the theory and practice of database cracking. In *EDBT*, pages 415–418, 2020.

[39] F. Zardbani, N. Mamoulis, S. Idreos, and P. Karras. Adaptive indexing of objects with spatial extent. *Proc. VLDB Endow.*, 16(9):2248–2260, 2023.