

# Revisiting Multidimensional Adaptive Indexing\*

Anders Hammershøj Jensen  
Aarhus University

Frederik Aarup Lauridsen  
Aarhus University

Fatemeh Zardbani  
Aarhus University

Stratos Idreos  
Harvard University

Panagiotis Karras  
Aarhus University

## ABSTRACT

Modern applications require managing large data in main memory. *Adaptive indexing* allows for building an index incrementally in response to queries, rather than upfront; in its default form, it treats each attribute independently. However, several data exploration tasks involve multidimensional range queries. Two recent proposals, CKD and QUASII, address the need for multidimensional (especially spatial) adaptive indexing. Both adaptively build an augmented KD-Tree. Still, no previous work has compared these two methods to each other. We conduct the first experimental comparison of CKD and QUASII. Further, we propose a lightweight variant of CKD, the Lazy CKD, which performs data-driven along with query-driven actions for the sake of robustness, and a set of hybrid strategies that combine good convergence and low initialization cost. Our study on synthetic and real data and workloads shows that the enhanced variants of CKD have an advantage in terms of speed of convergence, yet QUASII may eventually achieve lower response times.

## 1 INTRODUCTION

Scientists and analysts need to query and explore large amounts of data in dynamic environments where new data arrive continuously, without building a full index in advance. This need calls for self-organizing DBMSs that eschew human administration. *Adaptive indexing* [8], such as *database cracking* [4, 6], accommodates this need by building and refining a main-memory index incrementally, in response to queries and arriving data. Cracking is applied within the select operator in a column-oriented database [7, 10]. A *stochastic* alternative [4] creates *random cracks* so as to perform robustly on skewed workloads.

Despite the intense interest in adaptive indexing, such methods for multidimensional data have been examined scantily. Recently, two solutions appeared: the QUery-Aware Spatial Incremental Index [12] (QUASII) and the Cracking KD-Tree (CKD) [5] (CKD). Both incrementally build an augmented KD-tree [1]; both conclude that their query response times converges to that of static approaches after processing a sufficient amount of queries; however, no comparison among these two works has been attempted.

In this paper, we conduct the first comparison of these two approaches [5, 12]; we also propose a lightweight CKD variant, the Lazy CKD (LCKD), incorporate stochastic cracking [4] strategies to improve robustness, and propose hybrid strategies that combine desirable traits of different solutions. We evaluate all methods on both synthetic and real datasets and workloads.

\*The two first authors contributed equally to this work.

## 2 1D ADAPTIVE INDEXING

**Cracking** [6] progressively partitions and sorts a column by quicksort while answering range queries. A partition containing one or both query range bounds is further split, or *cracked*, up to a minimum size. A partition containing the entire interval is split into three. Cracking directly into three partitions is possible [6], yet cracking into two partitions twice instead yields better results [13]. Listing 1 presents the basic cracking algorithm.

Listing 1: Crack in two [6].

```

1 def crack_in_two(pivot p, value low, value high):
2   x1 ← point at position low
3   x2 ← point at position high
4   while (position(x1) < position(x2)):
5     if (value(x1) < p):
6       x1 ← point at next position
7     else:
8       while (value(x2) ≥ p &&
9             position(x2) > position(x1)):
10        x2 ← point at previous position
11      exchange(x1, x2)
12      x1 ← point at next position
13      x2 ← point at previous position

```

**Stochastic Cracking** [4] adds *data driven* cracking actions to standard *query driven* ones, which may perform poorly on skewed workloads. The Data Driven Center (DDC) algorithm cracks the partition where a query bound lies at the median recursively, until obtaining a sufficiently small partition, whereupon it cracks at the query bound. Data Driven Random (DDR) avoids median-finding by choosing a random pivot. The DD1C and DD1R variants perform only one median or random crack. Still, these strategies retain the overhead of query-driven cracking. To ameliorate it, Materialization-based DD1R (MDD1R) [4] cracks the piece in which a query bound lies only on a single random pivot and materializes the result. Progressive MDD1R [4] shares the burden across queries, allowing crackin to be partially completed. Refined variants use the median of a sample set instead of a single random pivot, with performance gains [18].

**Adaptive Merging** [2] splits the data into arbitrary initial partitions, from which it progressively extracts query results into to a final sorted partition by incremental mergesort; it achieves faster convergence than cracking at the cost of high initialization cost [8]. A hybrid combination of the two [8] applies cracking on initial partitions and sorting on the final one to achieve both lightweight initialization and quick convergence.

**Multidimensional Cases.** A first study in adaptive multidimensional index structures [16] was about reorganizing, rather *building*, data-oriented hierarchical indexes, in response to a workload, so as to improve performance. Recently, QUASII [12] and CKD [5] extended adaptive indexing to the multidimensional case, by applying cracking on one dimension per tree level to construct a KD-tree-like structure. To our knowledge, these works have not been compared. Next, we discuss them in detail.

## 3 MULTI-D ADAPTIVE INDEXING

**Augmented KD-tree.** Both QUASII [12] and CKD [5] build their indexes on top of an augmented KD-tree [1], progressively redistributing data from the root node to newly created children

nodes while processing queries, allowing for a *variable number of children per node*. As new children nodes are added irregularly, the tree may lose the property of being *balanced*.

**Listing 2: QUASII query [12].**

```

1 def query(query q, data D, slices S, result R):
2   S' ← ∅ // to store newly created (refined) slices
3   dim ← S[0].l // current level/dimension of slices in S
4   i ← binarySearch(S, lower(q[dim]))
5   while (i < |S| and lower(S[i].box[dim]) ≤ upper(q[dim])):
6     if q ∩ S[i].box = ∅ then continue
7     S'' = refine(S[i], q, D)
8     for each s ∈ S'':
9       if q ∩ s.box ≠ ∅:
10        if s.l is the bottom level:
11          for j ∈ s.ids:
12            if D[j] ∩ q ≠ ∅:
13              R ← R ∪ D[j]
14        else:
15          if |s.children| == 0:
16            createDefaultChild(s)
17          query(q, D, s.children, R)
18   S' ← S' ∪ S''
19   i ← i + 1
20   S ← S ∪ S'
21   sort(S)

```

QUASII [12] comprises a hierarchical index structure of depth equal to the dimensionality  $d$ ; the index starts off as a root node containing all data objects, unsorted, and grows while processing range select queries. A bottom-level node may be split if it contains more than  $\tau$  objects; thus, QUASII slices a space with  $n$  objects up to  $r = \lceil \sqrt[n]{\tau} \rceil$  times in each dimension; at level  $\ell$  above the bottom the threshold becomes  $\tau[\ell] = r^\ell \tau$ . Listing 2 shows how QUASII processes a multidimensional range query  $q$ ; it finds the first slice hitting  $q$  by binary search (Line 4) along dimension (tree level)  $dim$ , on which slices are sorted along. It then scans and refines (i.e., slices further) all slices intersecting  $q$  (Lines 5–7), cracking them up to size  $\tau$ , by both query bounds and artificial cracks. If a resulting slice at the bottom level intersects  $q$  (Line 9), qualifying data objects therein are appended to the result  $R$  (Lines 10–13); in case the slice is at an internal level, QUASII recursively queries that slice's children (Lines 15–17). After refining all relevant slices in level (dimension)  $dim$ , QUASII resorts the slices therein (Line 21) with respect to the lower bound of their bounding boxes. After recursively traversing, cracking, and resorting slices as needed, it returns the range query result  $R$ .

**Listing 3: Refine [12].**

```

1 def refine ( slice s, query q, data D):
2   if (|s| ≤ τ[s.l]):
3     return {s}
4   S ← ∅
5   t ← determineSliceType(s,q)
6   switch(t):
7     case both: S' ← sliceThreeWay(s, q, D)
8     case one: S' ← sliceTwoWay(s, q, D)
9     default: S' ← sliceArtificial(s, q, D)
10  for each s' ∈ S':
11    if (|s'| > τ[s.l] and q[s.l] ∩ s'.box[s.l] ≠ ∅):
12      S'' ← sliceArtificial(s', q, D)
13      S ← S ∪ S''
14    else:
15      S ← S ∪ s'
16  return S

```

Listing 3 illustrates the process that refines each slice  $s$  that intersects the query  $q$  along the examined dimension  $s.l$  and exceeds the size threshold  $\tau[s.l]$ . QUASII cracks on any bound of  $q$  along dimension  $s.l$  that lies within  $s$  (Lines 7–8); otherwise, if  $q$  contains  $s$  along  $s.l$  (i.e., both bounds of  $q$  lie outside  $s$ ), it slices based on an artificially introduced coordinate  $c = \lfloor (x_l + x_u)/2 \rfloor$  (Line 9), where  $x_l$  ( $x_u$ ) is the lower (upper) bound of  $s$  along  $s.l$ ; it recursively slices further each produced slice that exceeds the  $\tau$  size threshold and overlaps with  $q$  (Lines 11–12), otherwise adds it to the output (Line 15). Listing 4 presents the recursive procedure for such *artificial slicing*, which configured to handle multiple points having the same coordinate when cracking (Line 5).

**Listing 4: Artificial slicing.**

```

1 def sliceArtificial ( slice s, query q, data D):
2   if |s| ≤ τ[s.l]:
3     return {s}
4   c = ⌊(xl + xu)/2⌋
5   slices = crack(s, c, s.l)
6   ret = []
7   for s' ∈ slices:
8     ret = ret ∪ sliceArtificial(s', q, D)
9   return ret

```

QUASII represents each spatial data object using its lower coordinate only along any dimension. A slice is first defined by its cracking coordinates, or *cuts*, yet obtains its own *minimum bounding box* (MBB) embracing the spatial extent of each object therein, with overlapping among slice MBBs allowed. To avoid the MBB computation overhead, QUASII computes MBB bounds for a slice  $s$  only on the dimensions  $s$  has been fully refined along. During slice refinement and binary search, to capture any result whose representative corner point lies in an unrefined slice outside the query range, QUASII extends the lower coordinate of  $q$  by the maximum object extent in each unrefined dimension, as in [15]. QUASII was designed as an adaptive spatial index for data in 2 or 3 dimensions; in our experiments we test its performance on higher dimensionality too.

**Cracking KD-Tree (CKD)** [5] also lets an augmented KD-Tree grow through queries, and uses a minimum node size threshold  $\tau$ . However, it assigns dimensions to tree levels in round-robin fashion by a *modulo* operation, allowing for multiple levels cracking on the same dimension. Listing 5 shows how CKD processes a multidimensional range query  $q$  on a slice  $s$ ; if  $s$  is fully contained within  $q$  (Line 2), it extracts the contents of the given slice  $s$ , otherwise traverses any children of  $s$  (Lines 4–5); if  $s$  has no children and contains fewer than  $\tau$  elements, a check of those vs.  $q$  yields the result (Lines 6–7). If none of the above is the case, CKD cracks the slice; it further queries resulting slices lying outside the query bounds on  $dim$ , to refine the index on other dimensions (Line 13), and those within the query bounds, to obtain results (Line 16).

**Listing 5: CKD query.**

```

1 def query(query q, slice s, dimension dim):
2   if (isIncluded(s, q)):
3     return extractPoints(s, q)
4   if (s.children > 0):
5     return traverseTree(s, q, dim)
6   if (|s| ≤ τ):
7     return extractPoints(s, q)
8   slices = crack(s, q, dim)
9   nextDim ← dim + 1 mod q.maxDim
10  for s' in slices:
11    s.add_slice(s')
12    if (s' ∩ q == ∅):
13      query(q, s', nextDim)
14    else
15      relevantSlice ← s'
16  return query(q, relevantSlice, nextDim)

```

Listing 6 shows the tree traversal procedure. CKD finds the first slice within the bounds of  $q$  by binary search (Line 3) and examines all slices within those bounds (Line 4–8), adjusting query bounds for the current dimension to fit the data in the slice (Line 6) and builds the result by querying those slices (Line 7).

**Listing 6: CKD and LCKD traverse tree.**

```

1 def traverseTree ( slice s, query q, dimension dim):
2   res ← []
3   i ← binarySearch(S, lower(q[dim]))
4   while (i < |S| and lower(S[i]) ≤ upper(q[dim])):
5     s' ← S[i]
6     q' ← adjustQueryToSliceBoundaries(q, s', dim)
7     res ← res ∪ query(q', s', nextDim)
8     i ← i + 1
9   return res

```

## 4 DISCUSSION AND ENHANCEMENTS

**Lazy Cracking KD-Tree.** The CKD [5] gratuitously refines all resulting slices after a crack on the query range according to query bounds. This gratuitous refinement is redundant; as it suffices to crack the pieces overlapping the query. We propose a variant that does so, the *Lazy Cracking KD-Tree* (LCKD). Listing 7 displays how LCKD processes a query, the change seen in Lines 10–13. In each dimension, LCKD only cracks partitions between query bounds, rather than cracking all pieces in excess.

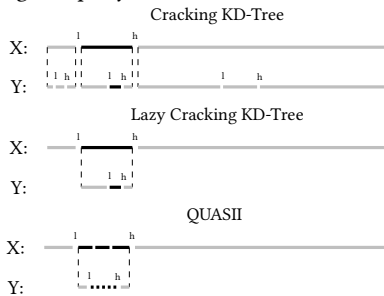
**Listing 7: LCKD query.**

```

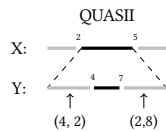
1 def query(query q, slice s, dimension dim):
2   if (isIncluded(s, q)):
3     return extractPoints(s, q)
4   if (s.children > 0):
5     return traverseTree(s, q, dim)
6   if (|s| ≤ τ):
7     return extractPoints(s, q)
8   slices ← crack(s, q, dim)
9   nextDim ← dim + 1 mod q.maxDim
10  for s' in slices:
11    s.add_slice(s')
12    if (s' ∩ q ≠ ∅):
13      relevantSlice ← s'
14  return query(q, relevantSlice, nextDim)

```

**Intuitive Comparison** We now have three strategies to compare: QUASII, CKD, and LCKD. All start with a tree consisting of a single node, and progressively build a hierarchical index while answering queries. In QUASII, the tree height is equal to the number of dimensions. In CKD and LCKD, the maximum tree height is determined by the size threshold  $\tau$  for cracking a partition. Figure 1 sketches the tree structures resulting after processing a single query under these strategies in the two dimensions; each tree layer corresponds to a different dimension. In CKD, there is an initial three-way crack along dimension  $x$ , followed by cracking all resulting pieces along query bounds on  $y$ . LCKD only cracks the  $x$ -partition relevant to the query along the bounds on  $y$ . The tree LCKD builds is less balanced than that of CKD, as successive queries may refine the index in the same region. Lastly, QUASII fully refines the entire part of the index relevant to the query in each dimension, making the tree wider, down to the minimal partition size  $\tau$  [12]; CKD cracks the chunk relevant to the query, as well as pieces irrelevant to the query, on all dimensions according to query bounds, yet not necessarily down to final partitions; LCKD refines only the relevant part of the data along the query bounds in each dimension.



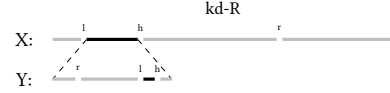
**Figure 1: Three cracking strategies in action.**



**Figure 2: QUASII rigidity; Q1[(2,4)–(5,7)], Q2[(3,6)–(6,9)]**

Figure 2 illustrates the problem that would arise in case QUASII did not perform full refinement. Assume a query on the range defined by lower left coordinates (2, 4) and upper right coordinate (5, 7), triggering a cracking of the  $x$  dimension on range [2, 5]

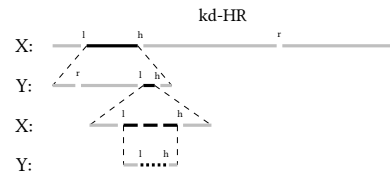
and the  $y$  dimension on range [4, 7]. Consider two points, (4, 2) and (2, 8), which belong in the  $x$ -range [2, 5], but get separated by the  $y$ -interval [4, 7]. If a subsequent query requested the range from (3, 6) to (6, 9), we should crack the  $x$  dimension at 3, hence swap points (2, 8) and (4, 2), destroying the sorted order on  $y$ . To prevent such an eventuality, QUASII cracks exhaustively upon the first query on any data range. Given this rigidity of QUASII, we discuss *stochastic cracking* on LCKD only.



**Figure 3: kd-R in action.**

**Multidimensional stochastic cracking** As described in Section 2, we may spur query-driven cracking via data-driven cracks, to achieve faster convergence with a slight initialization overhead [4]. In [4], the DD1R strategy emerged as most commendable. We mend LCKD using DD1R in the multidimensional case; we call the resulting method *kd-R*, where R stands for *random*. In each dimension, when cracking a data segment, *kd-R* cracks at a random point in addition to the query bounds, as Figure 3 shows. Next, we examine other ways to improve upon LCKD.

**Hybrid cracking** We propose *kd-HR*, a *hybrid* strategy that aims to combine the fast convergence of QUASII and the low initialization cost of LCKD; *kd-HR* initially behaves as *kd-R*, yet switches to QUASII and stops creating new levels once the tree reaches a specified threshold. We design two *kd-HR* variants, depending on the nature of the threshold: *kd-HR<sub>s</sub>*, with a threshold on node size, and *kd-HR<sub>ℓ</sub>*, with a threshold on tree height. Figure 4 shows the operation of *kd-HR* on the last level before switching to QUASII. *kd-HR* postpones QUASII-like operation until the data becomes sufficiently small (*kd-HR<sub>s</sub>*) or the area in question has been refined enough times (*kd-HR<sub>ℓ</sub>*); this precaution should bring about faster convergence, as the tree stops growing further on branches switching to QUASII.



**Figure 4: kd-HR in action.**

## 5 EXPERIMENTAL STUDY

Here, we present our experimental study featuring QUASII, CKD, and their stochastic and hybrid variants. We implemented all methods<sup>1</sup> in C++ and compiled in g++ 7.4.0, and conducted experiments on a 10-core Intel Xeon CPU E5-2687W v3 machine at 3.10GHz with 396G RAM running Ubuntu 18.04.3 LTS.

ID	Name	Size	Distribution
0	Random	240000	Uniform distribution
1	Skyserver full	722711000	Skyserver data
2	Neuroscience	1000000	Neuronal data

**Table 1: Datasets**

**Data.** Table 1 lists our data sets. Random is a synthetic containing points distributed uniformly at random between 0 and 1 in each dimension; the default dimensionality is 2. The Skyserver data are downloaded from [14], a public astronomical data repository, by the CasJobs functionality. We chose two attributes, *declination*, *dec*, and *right ascension*, *ra*. To experiment with data

<sup>1</sup>The code is available at <https://github.com/MULTIDAI/MultiDAI>

of varying size, we construct truncated versions of this data, selecting every  $i$ th point,  $i \in \{2, 4, 8, \dots, 512\}$ . The Neuroscience dataset consists of 3-dimensional model of a neocortical column in a brain tissue with MBBs matched to neuronal axons.

Name	Size	Distribution
Random	10000	Random distribution
Sequential	1000	Queries along diagonal
Skyserver chronological	1000000	Skyserver workload
Random clusters	50000	Synthetic normal clusters

**Table 2: Workloads**

**Workloads.** Table 2 lists our query workloads. The first synthetic workload, Random, issues range queries at locations selected uniformly at random, with extent 1% of the value domain per dimension. The second synthetic workload, Sequential, issues a non-overlapping sequence of consecutive range queries along the diagonal of the domain of celestial coordinates, again with extent 1% of the value domain per dimension; as this workload explores the data space incrementally in small steps, the index constructed under its guidance never gets an opportunity to exploit previous indexing. Skyserver workloads derive from the SqlLog table [14], containing queries executed by scientists in nonrandom patterns, focusing on one sky area at a time [4]. We filtered the range selection predicates on declination dec and right ascension ra. We use three versions of this workload: chronological preserves the original order of queries; sorted on  $x$  contains the same queries, sorted on lower dec coordinate; sorted on size sorts them on total area of query range. The latter two workloads present a skewed pattern resembling the skewed sequential workload we apply on synthetic data. The Random cluster workload is synthetically generated for use with the Neuroscience data; queries belong to Gaussian clusters surrounding 5 randomly chosen centers in the range of the data values, with a maximum query volume of 0.01% of the whole data volume.

**Compared methods.** We juxtapose QUASII [12]; CKD [5]; LCKD; kd-R; kd-HR<sub>s</sub> switching to QUASII when nodes that are smaller than  $s = 1000$ ; kd-HR<sub>l</sub> switching to QUASII after  $\ell = 6$  levels; kd-C, a variation of kd-R that performs data-driven cracks on the center of the value domain; Static, a static implementation of a KD-tree. Previous works have already compared QUASII [12] to a static R-tree [3] and CKD [5] to a static KD-tree [1].

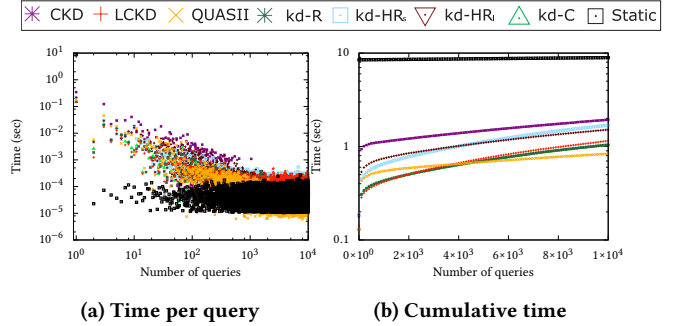
Tree	Synthetic	Skyserver	Neuroscience
QUASII	400	40	200
CKD	1000	170	400
LCKD	400	90	200
kd-R	400	80	200
kd-C	400	70	200
kd-HR	400	80	200

**Table 3: Chosen  $\tau$  values for synthetic and real data.**

**Parameter tuning.** All methods require a  $\tau$  parameter — the minimal cardinality of a data slice that may be further cracked. We conducted a series of experiments on the Random data (Table 1), with the Random and Synthetic workloads (Table 2). For each method, we chose as default the value of  $\tau$  for which we observed best performance. Table 3 presents those choices. In size-based hybrid, kd-HR<sub>s</sub>, we chose  $s$  through experimentation with several values;  $s = 1000$  yielded the best performance.

**Random workload.** We first examine the Random data with the Random workload. Figure 5a presents time per query. Evaluating the first query with Static counts for full index building. CKD is the slowest among adaptive methods in the first query, as it scans all data twice: once to crack on the first dimension, and again to crack the three resulting slices on the second dimension. QUASII comes second, while LCKD is the fastest. The initialization costs of stochastic and hybrid structures are almost identical, and slightly lower than that of QUASII. After the first query, CKD

variants reduce time per query, yet LCKD converges faster than CKD. The time of QUASII falls intermediately, yet keeps falling further than LCKD. QUASII performs more work in early stages, yet achieves fast response times later on, traversing a shallower tree. As the workload evolves, fewer queries require additional indexing actions in QUASII, letting time per query fall to even less than that of Static. Figure 5b presents cumulative running time; the divergence between QUASII and the others is conspicuous; Static is much slower than the adaptive approaches; CKD incurs higher cumulative runtime than other adaptive structures.

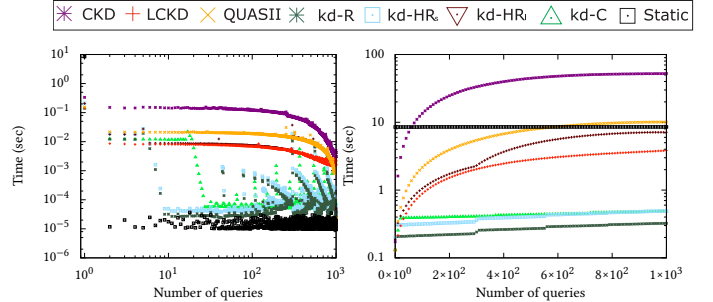


(a) Time per query

(b) Cumulative time

**Figure 5: Results on Random workload.**

A main takeaway from this experiment is that, over time, QUASII starts reaping the fruits of the index it has built more than other methods. We found this trend to be the same regardless of dataset: LCKD outperforms QUASII in the first queries at the expense of the quality of the index it builds, yet later queries do not benefit as much from the work done previously as they do with QUASII, and QUASII eventually outperforms LCKD. Unfortunately, the kd-HR hybrids present response times worse than LCKD, kd-R, and kd-C. We infer that the combination of kd-R and QUASII leads to a poor structure. Stochastic variants, kd-R and kd-C, present similar runtimes to LCKD, with an advantage in the long term, as data-driven cracks show their benefit.



(a) Time per query

(b) Cumulative time

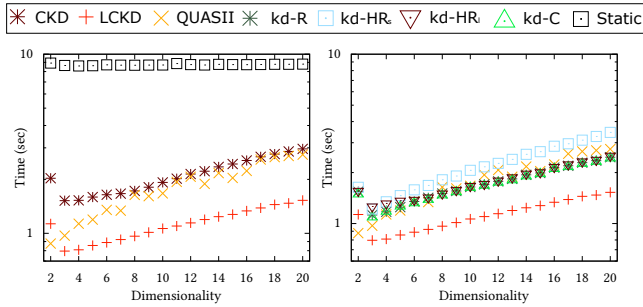
**Figure 6: Results on sequential workload.**

**Sequential workload.** Now we turn to the Random data with the Sequential workload. Figure 6 shows our results. Unsurprisingly, CKD performs poorly. More surprisingly, in contrast to the preceding experiment, QUASII also performs poorly compared to LCKD; the thorough index refinement QUASII performs is a liability with the sequential workload. LCKD benefits from its lazy nature. This experiment reveals that the order of query execution has a significant effect on running time, due to the query-driven nature of these data structures: on a sequential workload, each new query processes an unindexed data region. Notably, kd-R performs best; its data-driven operation confers an advantage on this workload. The spikes in the plot arise when entering a region refined by data-driven cracks to a lesser extent. Interestingly, kd-C does not match the performance of kd-R: its center-based cracks prove to be insufficiently robust, vindicating



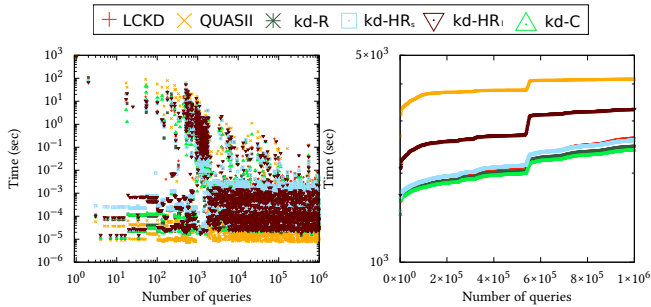
our choice to build hybrids on top of kd-R rather than kd-C. Yet those hybrids do not match the performance of kd-R either; their QUASII component appears to be a liability; this effect is apparent on kd-HR<sub>s</sub> and even more consequential in kd-HR<sub>ℓ</sub>.

**Effect of dimensionality.** We now evaluate the impact of data dimensionality on the Random dataset and the Random workload. Figure 7 depicts our results in two plots for the sake of readability. As dimensionality grows, the cumulative time of QUASII rises drastically, due to the thorough refinement performed on *each* dimension. CKD and LCKD are less affected by dimensionality growth. LCKD presents a modest runtime growth with dimensionality. Surprisingly, the cumulative runtime of the KD-tree variants initially drops as dimensionality increases, as they gain from the indexing they perform. In the global trend, CKD incurs a heavier cumulative runtime burden, as additional dimensions beget a higher overhead than the benefit of cracking. QUASII, which performs well on random workloads on data of dimensionality 2, forfeits this advantage in higher dimensions. QUASII and QUASII-based hybrids, especially kd-HR<sub>s</sub>, are affected by data dimensionality to a greater extent than LCKD, kd-R, and kd-C. We deduce that the QUASII strategy is detrimental on an increasing number of dimensions. Interestingly, the gap between LCKD and kd-R grows slightly with dimensionality, due to the additional random cracks performed by kd-R. On a random query workload, such random cracks bring little benefit, while incurring an overhead that rises with dimensionality.



(a) CKD, QUASII, Static. (b) Stochastic and hybrid.  
Figure 7: Effect of dimensionality.

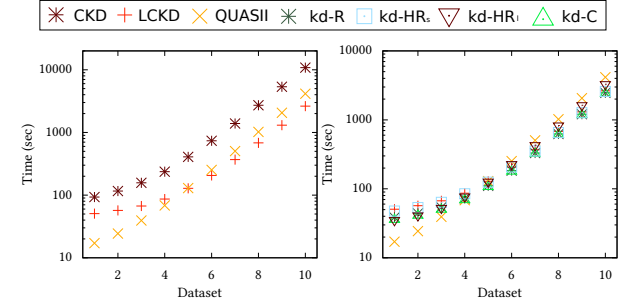
Henceforward, we use only LCKD and its stochastic and hybrid variants as representatives of cracking KD-trees and drop Static from figures for the sake of readability.



(a) Time per query (b) Cumulative time  
Figure 8: Skyserver 1/4, chronological order.

**Skyserver workload, chronological.** Now we apply the Skyserver chronologically ordered workload on Skyserver data. Figure 8 shows our results with the Skyserver 1/4 dataset. In the first 500 queries, QUASII occasionally reaches response times comparable to the other methods, which should correspond to accessing already indexed data areas; progressively expensive

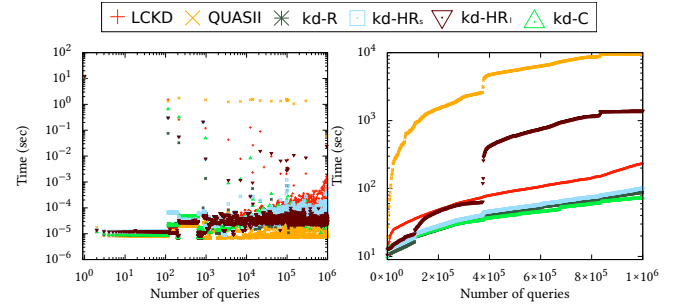
queries become rarer, and eventually QUASII converges to response times lower than those of other methods. This converged response time of QUASII does not render its cumulative time lower than those of others for the entire workload in this configuration; kd-C and kd-R achieve the best cumulative times. Still, as Figure 9 shows, as we reduce the data size (10 for full, 1 for 1/512) under the same workload so as to render the workload to data ratio larger, eventually QUASII becomes the fastest.



(a) QUASII, CKD, LCKD. (b) Stochastic and Hybrid.

Figure 9: Skyserver; regular workload, variable data sizes.

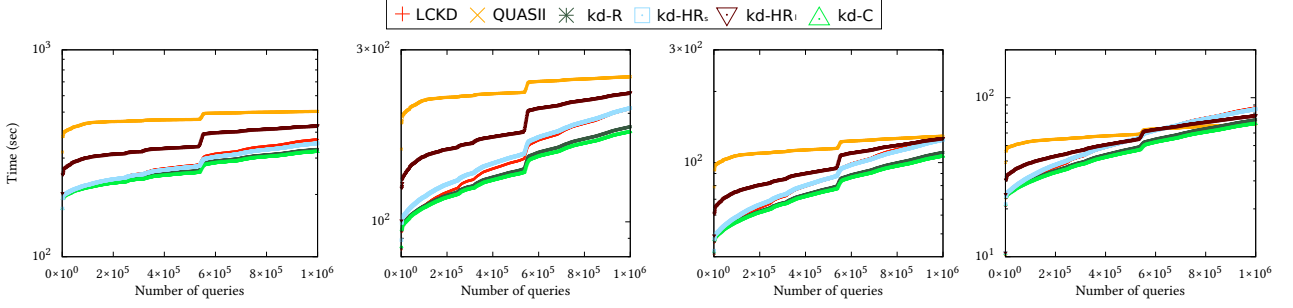
Figure 11 presents cumulative times with the same workload and four different Skyserver sizes. As data size falls, the time of QUASII approaches and supersedes, those of kd-HR, kd-R, and LCKD; still, kd-R and kd-C remain better options than QUASII for workloads reasonably large compared to the data.



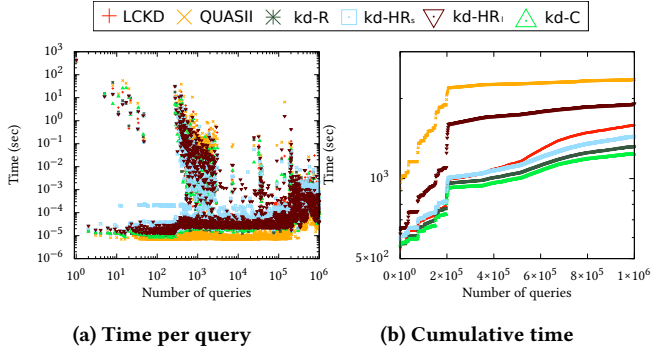
(a) Time per query (b) Cumulative time  
Figure 10: Skyserver 1/64, sequential workload.

**Skyserver workload, sequential.** We now assess performance on the Skyserver 1/64 data against the sequential Skyserver workload, sorted by the dec celestial coordinate. Figure 10 presents our results. Despite the smaller data size compared to those we examined previously, the sequential nature of the workload bears upon QUASII, which presents the worst result in cumulative time. LCKD achieves better cumulative time than QUASII, yet its response time does not converge as well; the tree it builds grows progressively higher in a lopsided manner. The stochastic variants, kd-C and kd-R, eschew the deficiencies of query-driven methods and attain best performance, while kd-HR<sub>s</sub> follows suit. On the other hand, kd-HR<sub>ℓ</sub>, which resorts to QUASII quite early, inherits the liability of QUASII. This result reconfirms that the QUASII strategy is a liability more than an asset in hybrids.

**Skyserver workload sorted by size** Now we apply the Skyserver workload ordered by query size on the Skyserver 1/4 data. Figures 12 and 13 show the results for *ascending* and *descending* order, respectively. On the ascending order, as expected, QUASII is initially slower, but achieves better query response times later. In cumulative time, kd-C performs best, closely followed by kd-R; kd-HR<sub>s</sub> does not gain from its hybrid character, while kd-HR<sub>ℓ</sub> has a clear disadvantage. LCKD is superseded by kd-R and kd-HR.

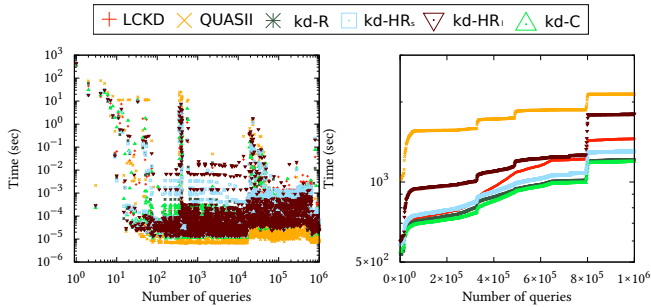
Figure 11: Skyserver, regular workload, datasets:  $1/8$ ,  $1/16$ ,  $1/32$ ,  $1/64$ .

the descending order, time per query is initially higher, as queries of larger extent require more indexing work; cumulative time grows more steeply in the early stage than with the ascending order, yet performance resembles the ascending case.



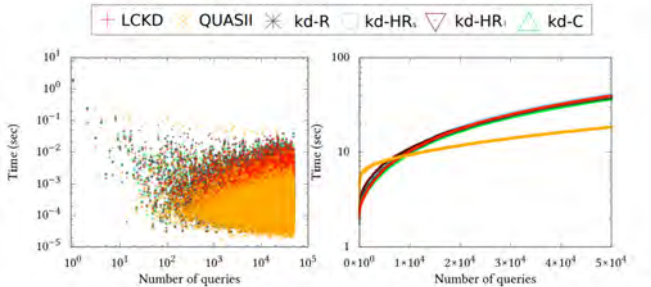
(a) Time per query

(b) Cumulative time

Figure 12: Skyserver  $1/2$ , workload sorted on size (asc.).

(a) Time per query

(b) Cumulative time

Figure 13: Skyserver  $1/2$ , workload sorted on size (desc.).

(a) Time per query

(b) Cumulative time

Figure 14: Neuroscience data, random cluster workload.

**Neuroscience data.** We now assess performance on the Neuroscience data with the random clustered workload. This dataset contains objects with spatial extent rather than points; thus, we employ *query window extension* [15], as in [12]: we represent shapes by their lower coordinates per dimension and *extend* query ranges by the maximum object extent towards the lower side of each dimension, to hit the lower coordinates of any object

overlapping the query's range; we filter *false hits* in a refinement step. Figure 14 presents our results. No method converges as robustly as with point data, due to the overhead caused by query extension. Still, QUASII converges more robustly than others.

## 6 CONCLUSION

We conducted a comparative experimental evaluation of works on multidimensional adaptive indexing and enhancements leveraging stochastic and hybrid strategies. We found that adaptations of the Cracking KD-tree achieve better performance compared to QUASII in terms of initialization and with short workloads, while QUASII yields attractive performance with long-running workloads. We combined the Cracking KD-Tree with stochastic measures that ameliorate the sensitivity to the order in which queries are posed. Further research is needed on multidimensional adaptive indexing of objects with spatial extent and accommodating updates; we also aim to investigate the adaptive indexing of graph structures with privacy constraints [11, 17] and adaptive multidimensional synopses [9].

## REFERENCES

- [1] Jon Louis Bentley. 1975. Multidimensional Binary Search Trees Used for Associative Searching. *Commun. ACM* 18, 9 (1975), 509–517.
- [2] Goetz Graefe and Harumi A. Kuno. 2010. Self-selecting, self-tuning, incrementally optimized indexes. In *EDBT*. 371–381.
- [3] Antonin Guttman. 1984. R-Trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD*. 47–57.
- [4] Felix Halim, Stratos Idreos, Panagiotis Karras, and Roland H. C. Yap. 2012. Stochastic Database Cracking: Towards Robust Adaptive Indexing in Main-Memory Column-Stores. *PVLDB* 5, 6 (2012), 502–513.
- [5] Pedro Holanda, Matheus Nerone, Eduardo Cunha de Almeida, and Stefan Manegold. 2018. Cracking KD-Tree: The First Multidimensional Adaptive Indexing (Position Paper). In *DATA*. 393–399.
- [6] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. 2007. Database Cracking. In *CIDR*. 68–78.
- [7] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. 2009. Self-organizing tuple reconstruction in column stores. In *SIGMOD*. 297–308.
- [8] Stratos Idreos, Stefan Manegold, Harumi A. Kuno, and Goetz Graefe. 2011. Merging What's Cracked, Cracking What's Merged: Adaptive Indexing in Main-Memory Column-Stores. *PVLDB* 4, 9 (2011), 585–597.
- [9] Panagiotis Karras and Nikos Mamoulis. 2008. Hierarchical synopses with optimal error guarantees. *ACM Trans. Database Syst.* 33, 3 (2008), 18:1–18:53.
- [10] Panagiotis Karras, Artyom Nikitin, Muhammad Saad, Rudrika Bhatt, Denis Antyukhov, and Stratos Idreos. 2016. Adaptive Indexing over Encrypted Numeric Data. In *SIGMOD*. 171–183.
- [11] Sadegh Nobari, Panagiotis Karras, HweeHwa Pang, and Stéphane Bressan. 2014. L-opacity: Linkage-Aware Graph Anonymization. In *EDBT*. 583–594.
- [12] Mirjana Pavlovic, Dariusz Sidlauskas, Thomas Heinis, and Anastasia Ailamaki. 2018. QUASII: QUery-Aware Spatial Incremental Index. In *EDBT*. 325–336.
- [13] Felix Martin Schuhknecht, Alekh Jindal, and Jens Dittrich. 2013. The Un-cracked Pieces in Database Cracking. *PVLDB* 7, 2 (2013), 97–108.
- [14] DR16 Sloan Digital Sky Survey. 2018. <http://cas.sdss.org/>.
- [15] Emmanuel Stefanakis, Yannis Theodoridis, Timos K. Sellis, and Yuk-Cheung Lee. 1997. Point Representation of Spatial Objects and Query Window Extension: A New Technique for Spatial Access Methods. *IJGIS* 11, 6 (1997).
- [16] Yufei Tao and Dimitris Papadias. 2002. Adaptive Index Structures. In *Vldb*.
- [17] Mingqiang Xue, Panagiotis Karras, Chedy Raissi, Panos Kalnis, and Hung Keng Pung. 2012. Delineating Social Network Data Anonymization via Random Edge Perturbation. In *CIKM*. 475–484.
- [18] Fatemeh Zardbani, Peyman Afshani, and Panagiotis Karras. 2020. Revisiting the Theory and Practice of Database Cracking. In *EDBT*. 415–418.