Benchmarking Adaptive Multidimensional Indices

Konstantinos Lampropoulos
U. of Ioannina & Aarhus U.
klampropoulos@cse.uoi.gr

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

Nikos Mamoulis
U. of Ioannina & Archimedes, Athena R.C.
nikos@cs.uoi.gr

Panagiotis Karras
U. of Copenhagen & Aarhus U.
piekarras@gmail.com

ABSTRACT

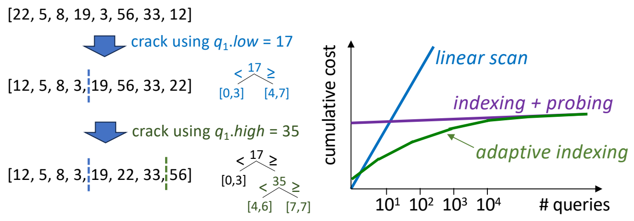
By *adaptive indexing*, an index grows dynamically and progressively through query processing. This mode of index-building, well explored over the past fifteen years, proves especially useful in exploratory scenarios where prebuilt indexes do not pay off the time to construct them, as the query workload variably focuses on particular areas of the search space, or the data become quickly obsolete. Despite a significant body of work in multidimensional adaptive indexing, there remains a gap in comparative studies that evaluate these methods on equal terms in a wide spectrum of settings, including data types, distributions, sizes, and workload patterns. This work fills this gap with a comprehensive benchmark to thoroughly evaluate the performance, strengths, and limitations of existing multidimensional adaptive indexing methods across diverse scenarios, contributing valuable insights that complement previous works. Further, we suggest supplementary technical extensions that enhance the efficiency of existing methods.

PVLDB Reference Format:
Konstantinos Lampropoulos, Fatemeh Zardbani, Nikos Mamoulis, and Panagiotis Karras. Benchmarking Adaptive Multidimensional Indices. PVLDB, 18(11): 4505 - 4517, 2025. doi:10.14778/3749646.3749709

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/DinosL/BenchmarkingAdaptiveIndexes.

1 INTRODUCTION

Adaptive indexing (AIx) has been extensively studied in the database community since 2007 [18, 19]. Consider an unorganized column C of a relation, e.g., in a column store [2, 35]. AIx constructs an in-memory index for C progressively and in response to range (or equality) queries along a single dimension or attribute [21]. Assume the first range query q_1 seeks records r such that $q_1.low \leq r.C < q_1.high$. AIx scans C to answer the query and also swaps its entries and divides it into three segments; the first segment contains the values smaller than $q_1.low$ (at no particular order), the second the query results, and the third the values greater than or equal to $q_1.high$. At the same time, a binary balanced search tree (e.g., an AVL tree) is initialized with nodes $q_1.low$ and $q_1.high$ to indexes the trichotomy. Figure 1a shows an example with $q_1.low = 17$ and $q_1.high = 35$. Each subsequent query q_i uses the tree to find the segments wherein $q_i.low$ and $q_i.high$ fall and partitions these segments in-place to obtain the results of q_i and expand the tree.



(a) Cracking example **(b) Cumulative cost**

Figure 1: Adaptive indexing

Figure 1b illustrates the typical cumulative cost for processing a query sequence by adaptive indexing vs. that of two alternatives: linearly scanning the unorganized data column for each query and probing a bulk-loaded index for each query. Linear scan is cheaper when a few queries are applied, while indexing fares well when the queries are many, amortizing the construction cost. Adaptive indexing is costly for the first few queries, which crack large data segments, yet settles to the per-query cost of searching a fully built index. In effect, for medium query counts, its cumulative cost is significantly lower compared to building an index in advance.

AIx facilitates the exploration of large short-lived datasets that become available in batches with a few, unpredictably distributed queries [4, 6, 8, 26]. For example, in meteorology and satellite imagery, data arrive in batches (e.g., snapshots of an entire area monitored by sensors or views of a big area of the globe), each batch becoming obsolete when the next batch arrives. In such cases, building a traditional, static index upfront with each batch arrival would delay query processing. Besides, if the queries are relatively few or target limited areas of the search space, it is not worthwhile to construct an index for the entire batch. When examining spatial entities from satellite images, the workflow typically involves finding a region of interest and zooming in to refined resolution. In effect, only a small portion of the data is examined, rendering the building of an index for all entities superfluous.

AIx has led to various extensions [1, 10, 11, 31], including updating operations [20], progressive merging variants [12, 15], a hybrid that combines merging with cracking [22], and a *stochastic* variant that introduces random cracks to handle skewed workloads [14]. An extensive experimental study on adaptive indexing [32, 33] also proposed a *coarse granular index* (CGI) that uniformly partitions the data to a few buckets in advance and then progressively indexing buckets by cracking those that contain each query's boundaries.

Proceedings of the VLDB Endowment, Vol. 18, No. 11 ISSN 2150-8097.
doi:10.14778/3749646.3749709

This paper focuses on the *in-memory* adaptive indexing of *multidimensional* data, which has been an area of interest for several years [16, 17, 24, 25, 29, 30, 38]. The most common queries on such data are multidimensional *range* queries, including window queries in spatial databases [27] and multicolumn filtering in analytical database engines [5, 28, 36]. Cracking a multidimensional data space is challenging, as dimensionality provides numerous partitioning options. Furthermore, objects in applications such as spatial databases, publish-subscribe systems, and data stream management systems are themselves multidimensional ranges, adding to the complexity of the ensuing indexes. While 1D adaptive indexes have reached a maturity level already 10 years ago and have been extensively evaluated [32, 33], multidimensional adaptive indexing is an active research area with several recent developments [25, 29, 38] that have yet to be evaluated within the same framework. Besides, ideas developed for one index (e.g., as in [38]) have not been tested on other structures (e.g., those in [29]) and composite solutions for the 1D case [32] have not been transferred and evaluated in the multidimensional case. Further, no practical guideline exists for the selection of an appropriate method based on the data type (points or ranges) and distribution. Lastly, the community currently lacks a testbed for the development and evaluation of new methods.

Contributions. A previous experimental study on multidimensional adaptive indices [23] compared QUASII [30] and AKD [16], the only available methods at the time. Variations of these methods were also evaluated in [29]. However, the studies in [23, 29] only target point data and do not consider hybrid schemes that combine multidimensional partitioning with cracking, as in [32] for 1D data. A later study [38] reevaluated state-of-the-art methods, yet focuses on low-dimensional spatial data. Besides, an adaptive index for metric spaces [25] was not included in those studies. We fill these gaps via the following contributions:

- We comprehensively evaluate existing multidimensional adaptive indexes using real and synthetic datasets and workloads of diverse (i) data types (point or box), (ii) dimensionality, (iii) data distribution, (iv) size distribution (for boxes), (v) query distribution, and (vi) query order. Thereby, we determine the superiority of certain techniques within various areas of the problem space.
- We apply enhancements proposed for AIR [38] to AKD [29] to craft the Advanced AKD (AAKD).
- We implement the concept of Course Granular Index (CGI) [32] in multidimensional spaces and evaluate its effectiveness.
- We devise and evaluate a range-query version of multidimensional distance-based adaptive indexing, originally developed for radius and kNN queries [25].
- We propose an evaluation framework for multidimensional adaptive indexing, including module and method implementations, real datasets, synthetic data generation modules, and generators of query workloads.

Our experimental findings are summarized as follows.

- Among non-adaptive indexes, a uniform grid universally outperforms other in-memory alternatives, such as the R-tree.
- AKD [29] can benefit by adopting the enhancements of AIR [38].
- It is most robust to partition the data apriori using a coarse grid and then adaptively index each cell by AKD for points, or AIR for range objects.
- Among adaptive indexes, the Advanced AKD prevails on point data, while AIR performs best on range data.
- The AV-tree [25], proposed for similarity queries in high dimensional metric spaces, does not perform well on multidimensional range queries in low dimensions.
- Currently, there is no golden use-for-all solution.

2 METHODS

Here, we describe the multidimensional (adaptive) indices that we compare experimentally. We focus on in-memory indexing of multidimensional points and hyperrectangular ranges, which we call, for simplicity, boxes. Table 1 summarizes the compared methods and classifies them into three categories based on their adaptiveness. Pre-bulit, i.e, *non-adaptive* indexes are fully constructed before query processing. The second class includes adaptive indexes that employ database cracking: they progressively construct an index in response to queries on an initially unorganized array. In a class of its own is a composite index that applies CGI [32] in the multidimensional space; this is not a purely adaptive index, as it first coarsely partitions the data and then indexes the partitions adaptively.

2.1 Non-adaptive indexes

Non-adaptive indexes are general-purpose index structures for multidimensional data, applicable to index multidimensional points and ranges. These include the R-tree [13], the quadtree [9], and a multidimensional uniform grid [37]. The R-tree and quadtree adapt to data skew. The R-tree yields a balanced hierarchical structure, originally designed for disk-based indexing, yet has been implemented for efficient in-memory search too [3]. Contrariwise, the quadtree partitions the data space at varying resolution per region adapting to data skew to yield an imbalanced structure, and is designed for points, yet can be extended to handle ranges. Grids are highly efficient data structures for in-memory indexing, originally proposed for point data, yet extensible to handle non-points [37]. While they perform best on non-skewed data, they were shown to perform well for moderately skewed data [37]. R-tree partitions may overlap on each level of the hierarchy, while quadtree and grid partitions are spatially disjoint.

2.2 Adaptive indices

The *QUery-Aware Spatial Incremental Index* (QUASII) [30] is the first proposed multidimensional adaptive index. For each query, it cracks the data partitions that overlap the query range at one dimension per level by fixed order, following the recipe of 1D cracking. In effect, it associates one dimension with each index level. A partition is finalized and never re-cracked once the objects it comprises are no more than a *cracking threshold* τ, which ensures that the indexing overhead is worth the benefits it brings.

The adaptive KD-tree (AKD) [16, 29] repetitively cracks a lead node of the incrementally constructed KD-tree in two pieces, along a multidimensional range query boundary, as long as the piece which includes the query has more than τ elements. For each query, it processes all lower bounds before all higher bounds, and associates dimensions with tree levels in round robin fashion. Unlike QUASII, it cracks on the same dimension multiple times at different levels. The *Progressive KD-Tree* (PKD) and its extension, the Greedy

Table 1: Classification of tested methods

	Method name	points	boxes	replication (boxes)	tree	tree balance	overlapping partitions
Pre-built	R-tree [13]	✓	✓	X	✓	✓	✓
	Quadtree [9]	✓	✓	✓	✓	X	X
	grid [37]	✓	✓	✓	X	–	X
Adaptive	QUASII [30]	✓	✓	X	✓	✓	X
	AKD [16, 29]	✓	X	X	✓	X	X
	AIR [38]	✓	✓	X	✓	✓	✓
	AV-tree [25]	✓	X	X	✓	X	✓
	AAKD (combines [29] and [38])	✓	X	X	✓	X	X
Hybrid	CGI (applies CGI [32] to *k*D spaces)	✓	X	X	X	X	X

Progressive KD-Tree (GPKD) [16], mitigate some disadvantages of the AKD associated with the initial query cost. PKD lets a parameter δ dictate the fraction of the dataset indexed per query, to achieve a tradeoff between indexing overhead and pace of index-building. Smaller δ values reduce the overhead, yet also slow down the progress, whereas larger values accelerate construction at the cost of higher overhead. GPKD uses a cost model to estimate the execution time of each query and ensures that each query has a consistent and robust execution time during index growth.

The *adaptive Incremental R-tree* (AIR) [38] progressively constructs an in-memory R-tree [13], distinguishing its leaf nodes into *regular* and *irregular* ones. AIR cracks each irregular leaf along each rectangular range query boundary that intersects the leaf; it prioritizes dimensions that split the data space evenly by choosing, in each cracking step, the query bound closest to the leaf's midpoint along the leaf axis with the largest extent. A partition holding fewer than τ elements becomes a regular leaf, not to be further cracked. Although designed for range data (i.e., MBRs), AIR is also applicable on point data. AIR is robust to workloads by stochastically cracking [14] the largest piece resulting from a query.

The AV-tree [25] is designed for distance-based range and NN queries in high-dimensional spaces, where the distance measure is typically Euclidean distance, defining spherical ranges. To apply AV-tree for orthogonal range queries, which is the focus of this paper, we have to convert orthogonal range queries to distance queries. This is done by computing the geometric center of the orthogonal query range and expand it in each dimension by half the extent of the query in that dimension. Specifically, consider a rectangular query q, expressed by an interval $[q^d.low, q^d.high]$ in each dimension d. We convert this query to a *weighted* L_{max} query using its geometric center $q.p = \{\frac{q^d.low+q^d.high}{2}, \forall d\}$ as a pivot point and retrieve all data points p, such that for each dimension d, $|q.p^d - p^d| \leq q.b^d$, where $q.b^d = (q^d.high - q^d.low)/2$.

However, since each node of the the AV-tree uses the same (median to all data under the node) distance bound along all dimensions (otherwise we would have the overhead of keeping multiple bounds at each node), when constructing the AV-tree, we use these geometric centers as pivots and median distances as ϵ bounds. To determine the cracked pieces (i.e., AV-tree leaves) relevant to a query q, we use the L_{max} distance of the pivot $v.p$ at each tree node v, and:

- access the left subtree of $v.p$, which includes all data points p such that $\forall d : |v.p^d - p^d| \leq v.\epsilon$, when $\exists d : |v.p^d - q^d| - v.\epsilon \leq q.b^d$.
- access the right subtree of $v.p$, having all data points p such that $\exists d : |v.p^d - p^d| > v.\epsilon$, when $\exists d : |v.p^d - q^d| - v.\epsilon > q.b^d$.

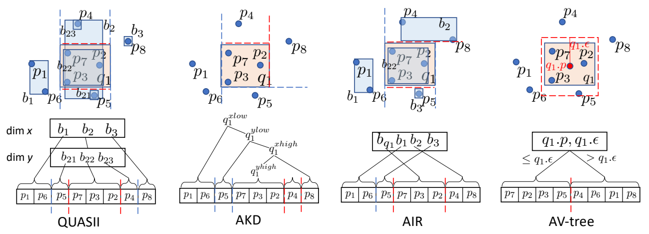


Figure 2: Multidimensional adaptive indices

Figure 2 illustrates the cracks that each of the four compared multidimensional adaptive indexes induces to an unorganized array $[p_1, p_2, \ldots, p_8]$ of 2D points. For the sake of fairness, apart from the aforementioned methods, we also test an *Advanced AKD* (AAKD), which, instead of a round robin process, prioritizes the cracked dimensions by the heuristics of AIR, proved in Ref. [38] to be more robust than other alternatives. We crack a piece on the minimum query bound along the dimension of its largest extent. We also introduce a stochastic crack to the largest ensuing piece, as also successfully applied in AIR.

2.3 Hybrid indexing

The *Coarse Granular Index* (CGI) [32] is hybrid method proposed for 1D data. Initially, CGI scans and partitions the data domain to equi-width ranges. For each query q, CGI determines the partitions relevant to q in O(1), cracks the borderline relevant partitions on the query bounds, updates an AVL tree accordingly, and returns the query results. Hence, in 1D, each query requires at most two cracks, one for each bound.

As CGI has yet to be generalized to the multidimensional case, we explore the effectiveness of a coarse multidimensional grid that partitions the data before the first query, and then cracks each partition using a local adaptive multidimensional index, such as AKD or AIR. Figure 3 sketches a multidimensional GCI. First, we partition the data space into tiles T_1 to T_{16} by a 4×4 uniform grid and place the data in each tile T_i into an (unorganized) array. Upon the first query q_1, we identify the set of relevant tiles, T_4, T_5, T_8, T_9 and crack the array of each of those on the query boundaries to initialize a local adaptive index (e.g., AKD tree), which guides and grows with subsequent queries.

We name CGI methods by the granularity and adaptive index they use, e.g., CGI100+AKD denotes a CGI of granularity 100 per dimension where each partition hosts an AKD index. We include CGI with AIR in experiments with shape data, where AIR is competent vs. AKD. We apply CGI only on 2D and 3D data, as the number of tiles, and hence the partitioning and storage costs and partition sparsity, grow exponentially with dimensionality.

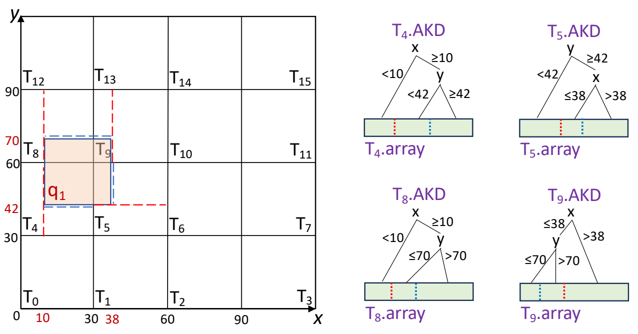


Figure 3: Multidimensional CGI using AKD

3 EXPERIMENTAL SETUP

We implemented all methods in C++ and compiled them in g++ 7.4.0 with the -o3 switch; experiments ran on a 3.10GHz 10-core Intel Xeon machine with 396G RAM running Ubuntu 18.04.3 LTS. Here, we provide details on datasets (Section 3.1), query workloads (Section 3.2), and performance measures (Section 3.3). We also conduct tuning experiments (Section 3.4) that suggest the most robust parameter values for the compared methods.

Table 2: Data sets

	Size	Dim	Max Ext.
Uniform	20M	2	0.0035 0.0035
Clustered	20M	2	0.0038 0.0038
SkewNormal	20M	2	0.00069 0.00069
ROADS	19M	2	0.0076 0.029

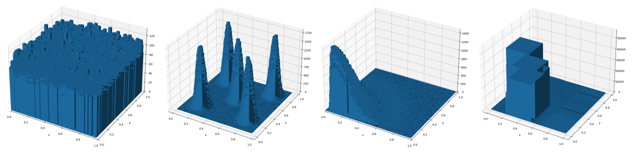
3.1 Datasets

3.1.1 Synthetic data. To test adaptive indexing across various datasets and query workloads with diverse characteristics, we generate synthetic datasets comprising of 20 to 80 million data objects, both points and range objects. The dimensionality of point datasets ranges from 2 to 6, while for range (i.e., hyperrectangular) data, we tested 2 and 3 dimensions, as ranges of higher dimensions rarely arise in real applications [37, 38]. We normalize the values in each dimension to the range $[0, 1]$. We generated the following synthetic point datasets:

- **Uniform data.** Random values following a uniform distribution in each dimension.
- **Clustered data.** We generate isotropic Gaussian blobs using the *make_blobs* function of Python scikit-learn module, with standard deviation 0.37 to ensure the formation of five non-overlapping clusters of equal cardinality. These values were set empirically to ensure minimum overlap, which challenges indices with the problem of handling white space.
- **Skewed data.** We generate random values by the *skewnorm* function from the SciPy library in Python, which produces data that follow a skew-normal distribution, which extends the normal distribution to incorporate non-zero skewness, allowing for asymmetric data shapes. We set the skewness parameter to $\alpha = 20$. For $\alpha = 0$, the distribution reverts to a standard normal distribution. We chose the value empirically to ensure that the generated data points exhibit sufficient skewness.

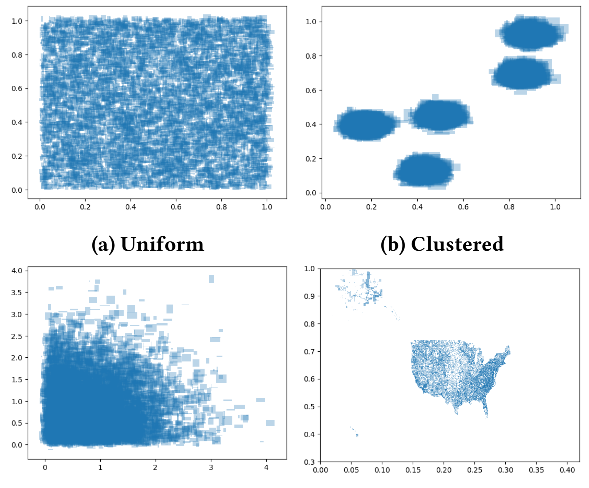
For range data, we generate points as above and use them as centers, then generate two random numbers within the range $[0.0001, 0.02]$ as mid-height and mid-width, to be added to and subtracted from the coordinates of the center point to determine the top right and bottom left corners. To assess the performance of methods such as the AKD, which is affected by the extent of range objects as they require query extension, we generated a 2D dataset with objects centered uniformly, and width and height following an exponential distribution $g(u) = 3^{-3u}$.

3.1.2 Real data. As real data, we utilize the publicly available **ROADS** dataset which features the shape of roads in the US [7]. We use the bottom left corner of each road shape as points. For points of higher dimensionality we use the **Taxi**[1] dataset which contains records of New York yellow-taxi trips. Each record captures the pick-up and drop-off location and time, trip distance, and fare amount for a period that spans January to July 2024.



(a) Uniform (b) Clustered (c) SkewNormal (d) ROADS

Figure 4: Distribution of point datasets



(a) Uniform (b) Clustered

(c) Skewed (d) Roads

Figure 5: Distribution of shape datasets

Figures 4 and 5 visualize the distribution of our data, both synthetically generated and real, points and ranges. The distribution

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

of the real dataset depicting U.S. roads is notably uneven, denser in urban areas and sparser in rural regions, leading to a skewed distribution. The generated shapes vary in both width and length, reflecting realistic and diverse spatial extents.

3.2 Workloads

Figure 6 depicts the access patterns in our workloads. For the random access pattern, we randomly select a point from the data and assign it an extent such that the average query selectivity is 0.01% (the default selectivity). The query distribution thus follows the data distribution. The sequential workload consists of non-overlapping diagonally consecutive queries. This workload presents a worst-case scenario for adaptive indexing, as each new query cracks a large area without benefit from previous ones. Lastly, the 'Zoom In' workload models a stylized exploratory search scenario.

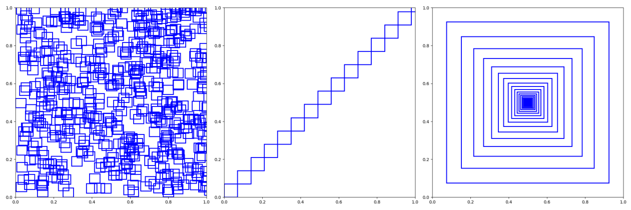

(a) Random (b) Sequential (c) Zoom In

Figure 6: Access pattern of synthetic workloads

3.3 Measures

As all adaptive indexes are in-memory access methods, the main evaluation measure is the evolving cumulative cost for a sequence of queries. Side-by-side to cumulative cost, we plot how the per-query cost changes while constructing the adaptive index. We expect the cost per query to progressively become lower and settle to that of a pre-built index. We average both the per-query and cumulative costs over 5 runs. For pre-built indexes, such as the R-Tree, we add their construction cost to the cumulative time prior to the first query. We also measure space requirements.

Table 3: Grid size tuning

	Grid			CGI+AAKD		
Grid Size	100	200	500	100	200	500
Uniform	1.140	**1.029**	1.348	1.050	**0.970**	1.364
Clustered	**1.181**	1.190	1.234	1.881	**1.133**	1.199
Skewed	1.183	**1.166**	1.186	1.844	**1.101**	1.185
Roads	0.909	**0.826**	0.935	3.274	**0.972**	1.430

3.4 Tuning

Prior to comparative studies, we select values of the following parameters for our investigation: the cracking threshold for AIR, QUASII, and AKD, node cardinality for the RTree, the δ variable for the GPKD, and grid size for the static grid and the course granular index. The δ variable for the GPKD controls the percentage of data that are used to expand the index in each query. A value of 0 means that no indexing occurs, only full scans, while $\delta = 1$ means that the index is fully built upon the first query.

We tune parameters across different values on our four main datasets: the synthetic uniform, clustered, and skewed points, as well as the ROADS dataset. Tables 3 and 5 show the total (cumulative) cost for workloads of 10k queries. We chose the most robust parameters, indicated via highlighted cells, for further experiments. Interestingly, while we expected the best size for the coarse granular index (CGI) to be smaller, i.e., *coarser*, than for the static grid, it turned out to have the same empirically best size. We attribute this result to the low creation cost of regular grids, which do not warrant waiting for the adaptation. When applying an irregular (data-adaptive) grid, we use a sample of the data to estimate the data distribution, whose size affects the quality of the estimate, hence the need to tune it. Similarly, the quad-tree has two parameters, node capacity and height. The former controls the amount of data a quadrant can hold before it splits further, while the latter controls the maximum tree depth, which can help avoid excessive splitting on skewed data. Table 4 details the search for the best empirical values of these parameters.

Table 4: Irregular Grid and Quadtree tuning

		Irregural - Grid Size			QuadTree - Height			
		10	50	100	8	10	12	
Batch/Capacity	100	**2.848**	3.491	3.846	14.253	19.551	19.511	Uniform
	1000	2.866	3.582	4.054	12.058	12.014	12.138	
	10000	2.875	3.601	3.963	**8.311**	8.313	8.330	
	100	**2.768**	3.470	3.810	10.659	17.209	20.532	Clustered
	1000	2.835	3.558	3.932	10.301	12.312	12.330	
	10000	2.826	3.593	3.970	**9.177**	9.156	9.197	
	100	**2.858**	3.541	3.988	10.469	16.740	20.363	Skewed
	1000	2.947	3.792	3.945	10.037	11.914	11.875	
	10000	2.873	3.618	3.978	**8.914**	8.952	8.948	
	100	**2.151**	2.965	3.477	8.672	12.185	19.170	Roads
	1000	2.149	3.077	3.546	8.550	11.441	12.830	
	10000	2.130	3.121	3.576	**8.369**	9.502	9.362	

4 EXPERIMENTAL EVALUATION

This section presents the results of our evaluation.

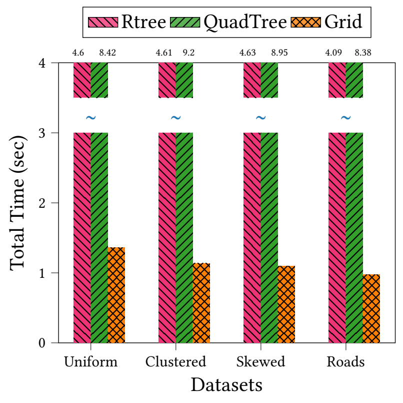
(1) First, we compare methods within each category: static, AKD-based, and grid-based using point and shape data to identify the most robustly performing representative for each category (§4.1).
(2) Next, we thoroughly evaluate the top methods across various types of data (points and ranges), with different distributions of object locations (§4.2) and query access (§4.7).
(3) Then, we assess how the distribution of object extents affects performance (§4.3).
(4) We also examine the effect of dataset size (§4.5) and query selectivity (§4.6).
(5) Finally, we examine the effect of dimensionality using high-dimensional point data (§4.8).

We plot the results in groups to maintain readable plots.

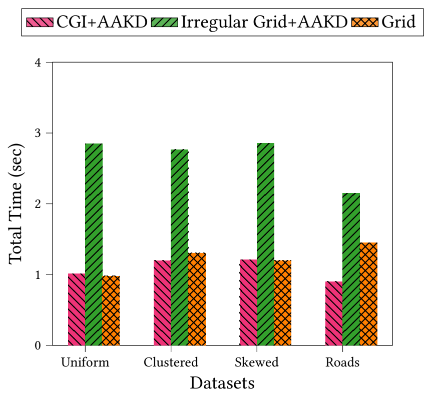
Since QUASII and AKD-based methods are not designed to handle data with spatial extent, we adjust them to achieve proper functioning with the *query window extension* technique [34] on data with spatial extent. We adjust the lower coordinates of the query window by the maximum object extent in the data, in each dimension, so that it overlaps any qualifying object. This extension introduces the overhead of filtering out false positives from the results, as the window then overlaps non-qualifying objects too.

Table 5: AIR, QUASII, AKD, RTree, and GPKD tuning

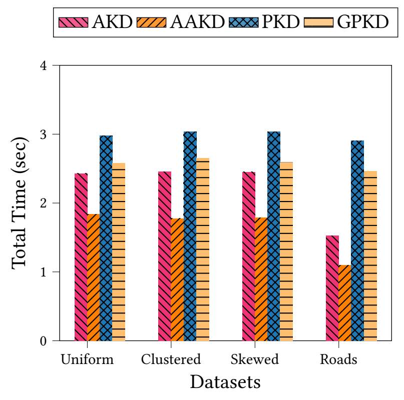
	AIR - Threshold			QUASII - Threshold			AKD - Threshold			R-Tree Node size			GPKD - δ		
	2048	4096	8192	1024	2048	4096	1024	2048	4096	1024	2048	4096	0.4	0.6	0.8
Uniform	2.071	**2.056**	2.060	2.498	**2.408**	2.414	2.515	**2.426**	2.431	4.589	**4.552**	4.641	2.641	**2.582**	2.584
Clustered	2.018	**2.003**	2.002	2.423	**2.335**	2.340	2.539	2.472	**2.455**	4.612	**4.551**	4.561	2.674	**2.616**	2.662
Skewed	2.031	**2.014**	2.179	2.464	**2.377**	2.393	2.509	**2.471**	2.481	4.687	4.641	**4.483**	2.709	**2.562**	2.592
Roads	**1.283**	1.304	1.307	1.881	1.812	**1.808**	1.516	**1.506**	1.570	4.235	**4.061**	4.070	2.575	**2.450**	2.473



(a) Pre-built

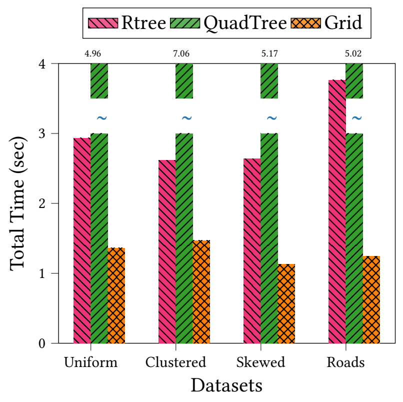


(b) Grid-based

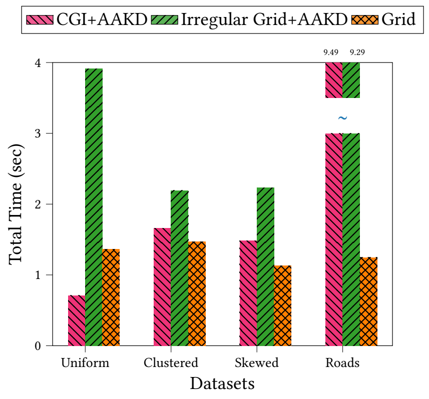


(c) AKD-based

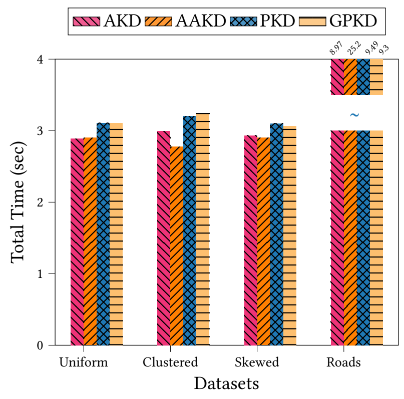
Figure 7: Comparison of indices in each category (cumulative time), point data



(a) Pre-built



(b) Grid-based



(c) AKD-based

Figure 8: Comparison of indices in each category (cumulative time), shape data

4.1 Method Selection

4.1.1 Pre-built Indices. Figures 7a and 8a illustrate the total workload time for index construction and evaluation of 10k queries, for three pre-built indices: R-tree, Quadtree, and a Grid with granularity of 200×200, for points and shapes data respectively. Surprisingly, while simply partitioning the space into equally sized cells, the grid index significantly and consistently outperforms the competition across all datasets and data types, independently of data skew. More complex indexes, such as the R-tree and the quadtree, take longer time to construct, which does not pay off; the simpler and less costly to construct grid index performs better overall. We thus use the grid as the representative of static methods in subsequent experiments.

4.1.2 Grid Indices. Next, we examine the performance of grid-based indices, both static and hybrid, comparing Grid (static) to two hybrid indices: CGI+AAKD (regular grid) and Irregular+AAKD (irregular grid). Recall that the latter two (i) prebuild a coarse grid with spatially equal cells and (ii) adaptively and progressively grow an AAKD adaptive index within each grid cell. We consider the option of complementing CGI with an AIR index later, when dealing with range data. We set the granularity of all coarse grids to 200×200, as the tuning experiments indicated, and measure the cumulative cost for index creation and evaluation of 10k random queries.

Figures 7b and 8b show that CGI+AAKD is the best choice, apart from when dealing with the ROADS dataset, as the inclusion of an AAKD endowed with ample indexing space within each grid cell renders query evaluation faster, with uniform data benefiting the

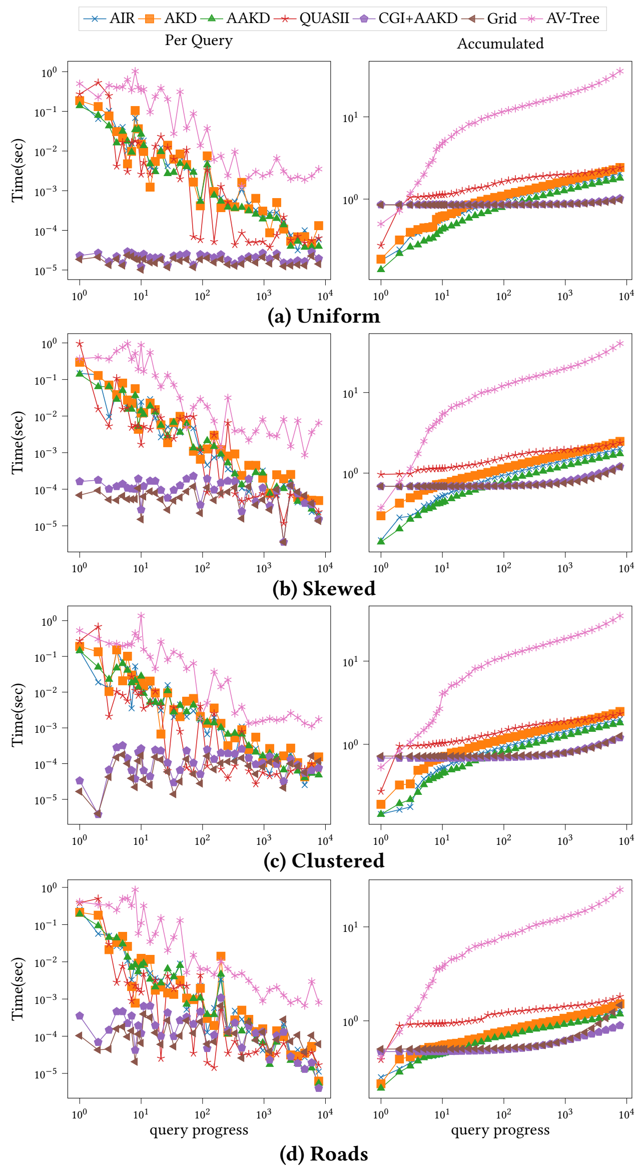


Figure 9: Effect of data distribution on 2D point datasets.

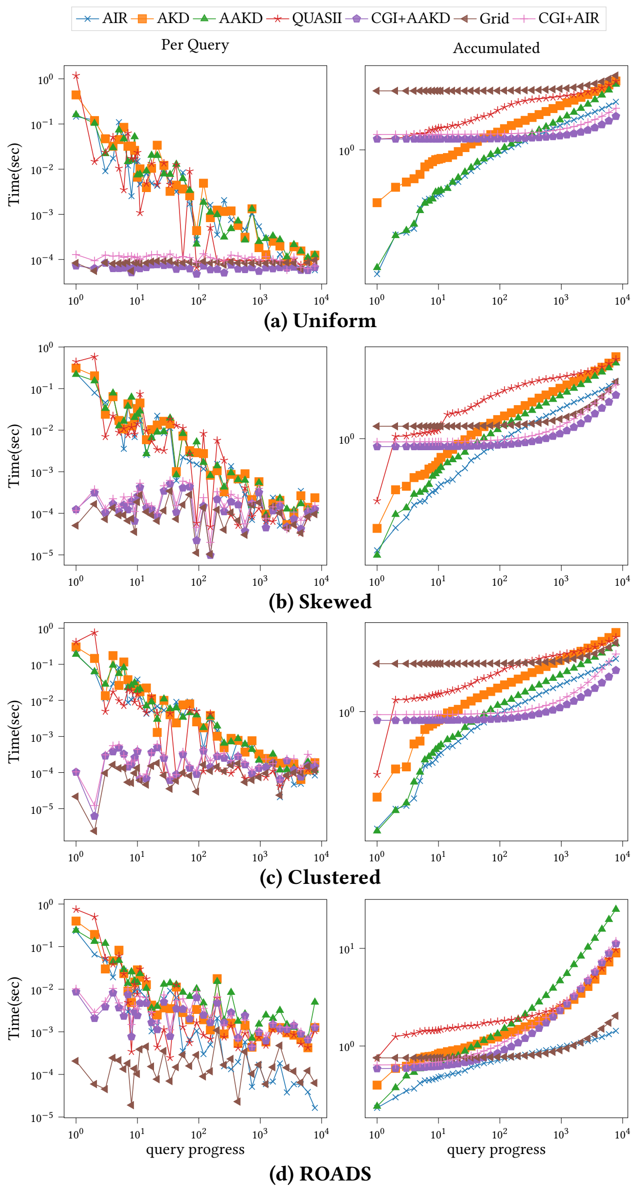
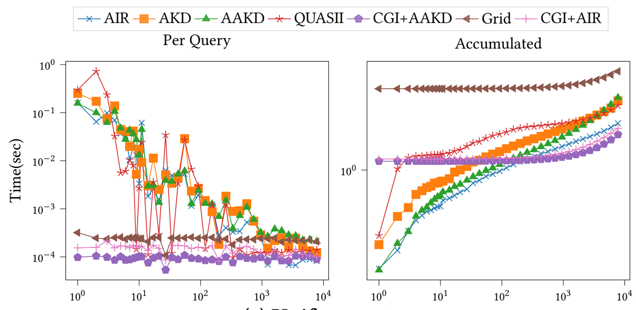


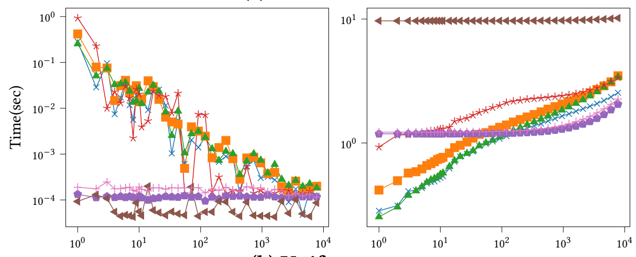
Figure 10: Effect of data distribution on 2D shapes.

most. The AAKD index refinement reduces the number of elements accessed, leading to improved query evaluation performance. Although one might have expected the irregular grid to outperform the regular one on non-uniform data, this is not the case, as the irregular grid incurs a higher cost for data partitioning, compared to the linear-time cost of constructing a regular grid. We thus use CGI+AAKD to represent the CGI index in subsequent experiments.

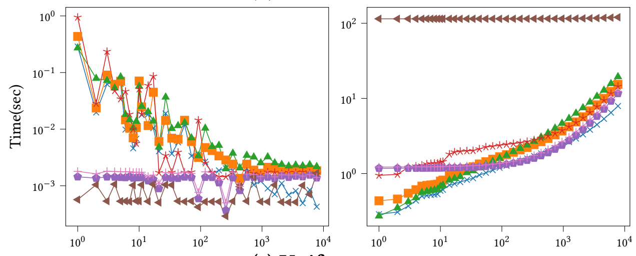
4.1.3 AKD Indices. Three adaptive KD-tree variants are suggested in [29]: the simple adaptive KD-tree (AKD), the progressive adaptive KD-tree (PKD), and the greedy progressive adaptive KD-tree (GPKD). We have formulated another option, the Advanced Adaptive KD-tree (AAKD), which adapts the heuristic ordering of cracks proposed by AIR. Specifically, AKD performs $2d$ cracks along the ends of a query range in each dimension by a fixed order, e.g., x_{low}, x_{high}, y_{low}, then y_{high}. Contrariwise, AIR orders cracks by a heuristic aiming to enhance the *marginality* of the ensuing structure. AAKD applies this heuristic on the AKD structure. We also introduce a stochastic crack on the largest resulting piece. Figures 7c and 8c show the total time to process a workload of 10k queries by these four options on different data distributions, for points and shapes respectively. As expected, GPKD outperforms the non-greedy on points. Additionally, the simple AKD outpaces progressive options. That is expected, as the progressive options are meant to overcome variance in the query time over the workload rather than to achieve low total workload time. Besides, the application of the crack-order-heuristic on the simple AKD proves worthwhile, as it outperforms other variants in all data distributions. In the remaining experiments we only include AKD and AAKD.



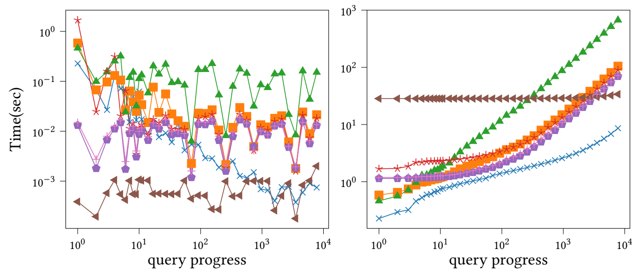
(a) Uniform 0.5%



(b) Uniform 1%



(c) Uniform 5%

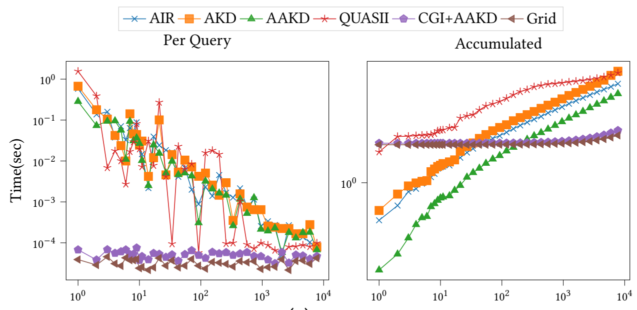


(d) Exponential

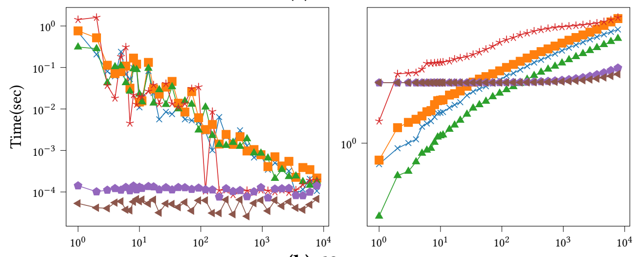
Figure 11: Effect of object extent (2D shapes)

4.2 Effect of object location

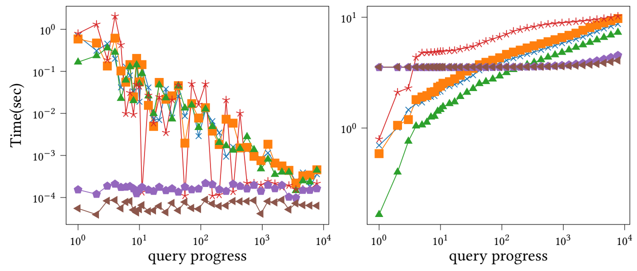
4.2.1 Point Data. We commence our comparative study for static, adaptive, and hybrid indexing options with 2D point data. Figure 9 shows how the methods selected in Section 4.1, i.e., static grid, CGI+AAKD, and AAKD, perform alongside the adaptive indexes: AIR, AKD, AV-tree, and QUASII. We present results for a random access pattern, showing both per-query and accumulated time across various datasets. Adaptive methods exhibit the typical behavior, starting with a high per-query cost, continuing with a declining trend, and eventually settling to the performance of a pre-built index. QUASII has an expensive start and subsequent slow growth, and does not overcome its initial handicap. AIR and AKD exhibit similar performance, with AAKD holding a small edge over both of them. This is expected as the datasets in question are point datasets.



(a) 40m



(b) 60m



(c) 80m

Figure 12: Effect of data cardinality on point data

We discuss experiments with range data in the following. These three indexes also appear largely insensitive to the data distribution.

The coarse granular index option with a regular grid complemented with an AAKD (**CGI+AAKD**) performs robustly across all datasets. So much so, that the per query performance resembles a pre-built static index even though the AAKD is still extending itself. Remarkably, this benefit does not come at an exorbitant initial cost, as the initial cost often resembles QUASII's first query cost, while this investment is vindicated by the total cost remaining competitive. Besides, the cumulative cost of CGI crosses the fully adaptive method latest after 100 queries. Another interesting observation is the comparison among the static grid and CGI. As mentioned in the tuning discussion (§3.4), the grid size of the static and CGI methods are equal, hence their initial building costs match. The total costs are on par with each other in all dataset comparisons except the ROADS data. As Figure 4 shows, the ROADS data has a few areas that are extremely crowded, and queries follow this data pattern. Therefore, the grid cells in those areas are overwhelmed with too many points to manage and many queries to respond to. This is where the extra layer of the adaptive index comes into play and facilitates better performance. Grid-based methods perform best when faced with a uniform data distribution.

On the other hand, the AV-tree performs significantly worse than its competitors. We attribute this gap to two factors: (i) a fundamental change in the query processing mechanism to accommodate

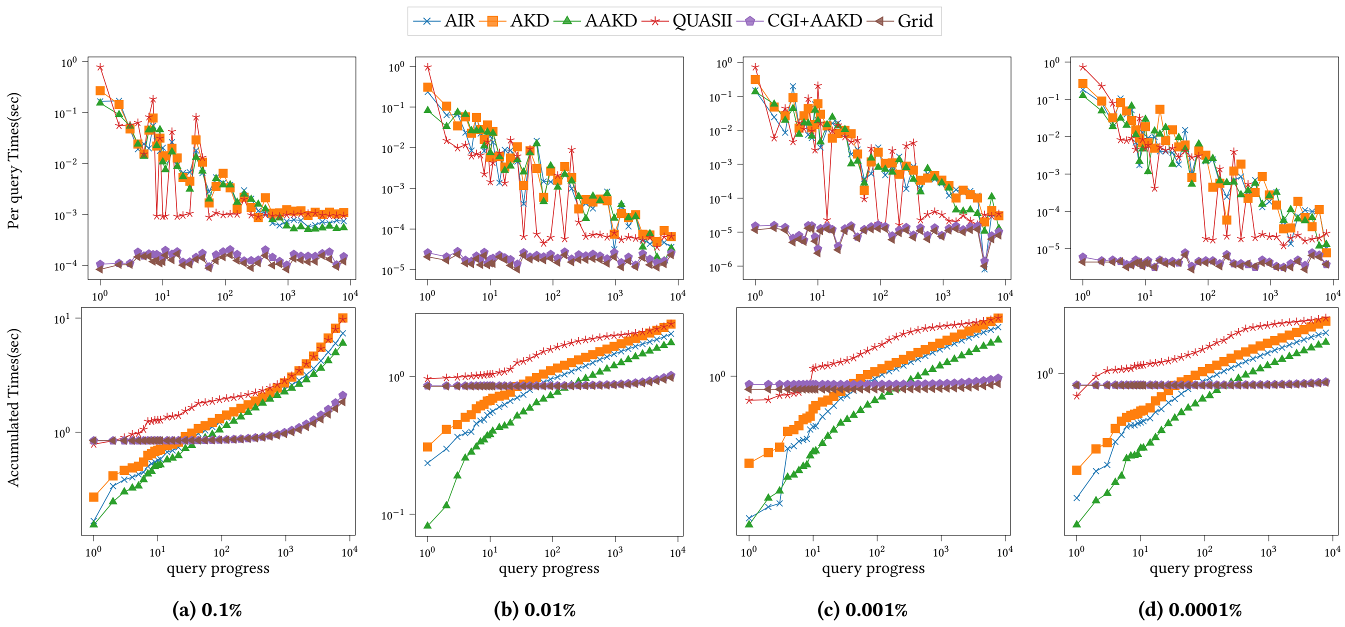


Figure 13: Effect of query selectivity.

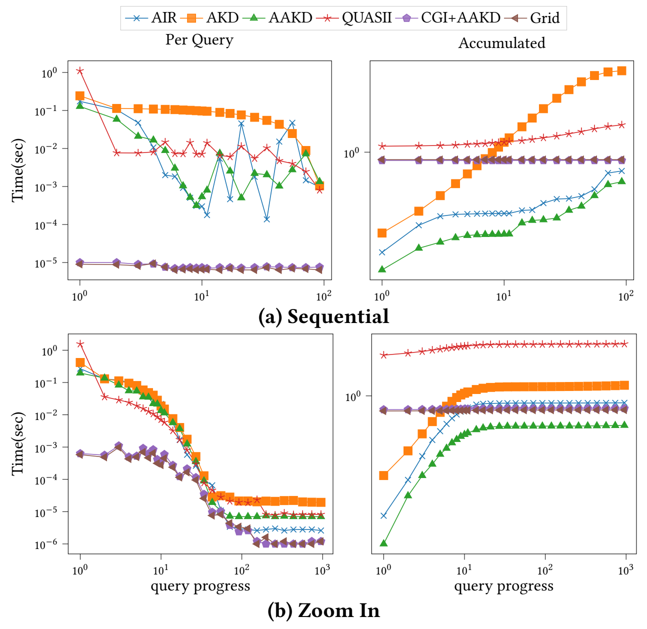


Figure 14: Effect of query access pattern, uniform point data.

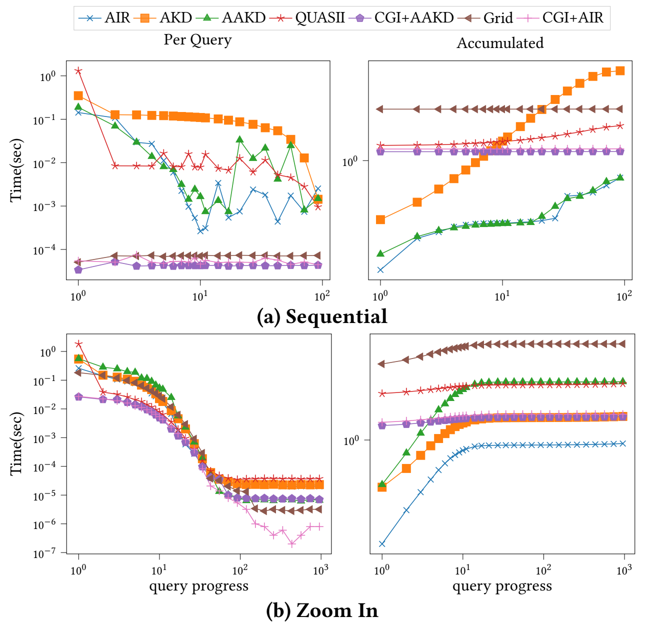


Figure 15: Effect of query access pattern, uniform shape data.

rectangular range queries; as the index was designed to accommodate L_2 distance queries, it creates suboptimal partitions when faced with L_{max} distance queries. And (ii) the low data dimensionality, which renders distance-based partitioning less appropriate. The combination of these two factors contributes to AV-tree's poor results. As its inclusion limits our ability to compare to other lines in the figures, we exclude this method from the remaining plots.

4.2.2 Shape Data. Figure 10 shows the per-query and cumulative workload time for the methods in Section 4.2.1 apart from the excluded AV-tree. CGI+AAKD performs consistently well for our synthetic data but struggles on real data. Interestingly, AIR shows the opposite behavior, which calls for further investigation. In the same context, QUASII, AKD, and AAKD also take a hit with different size distributions. We discuss this matter in detail in Section 4.3. The performance of AAKD, AKD, and QUASII has shifted up compared to the point experiments due to the query-window-extension overhead on shape datasets. This effect makes AAKD and AIR perform similarly for uniformly sized objects.

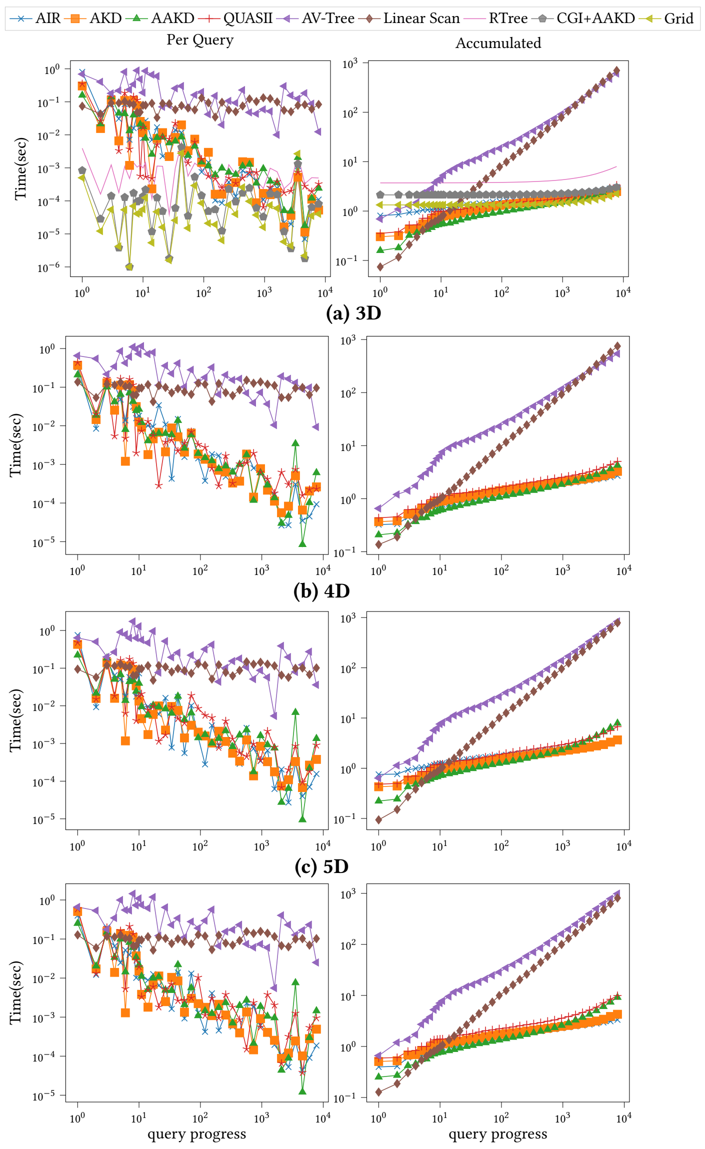


Figure 16: Effect of dimensionality, point data.

While the static grid does not outperform the current state of the art for shape datasets, AIR, in the uniform and ROADS data experiment, it does cross under in the other synthetic datasets, which is an important finding. We create another hybrid to complete the space of investigation: CGI using an AIR index. At first glance it may seem that this is a simple extension; however, we had to make an effectual design choice: We had to choose between either treating the dataset objects as shapes, creating the initial grid with replication, and consequently having shapes in our AIR indices, which would then deal with raw queries, or treat the dataset as points, create the initial grid without replication, inserting points into the AIR indexes, and applying the query window extension technique to reach correct results. The first option has almost the same startup cost and performance as the static Grid; as Figure 10 shows, this cost is much higher compared with the second option denoted by CGI+AAKD. Thus, we decided to use the second option as the standard implementation of CGI for boxes.

Another important finding is the performance of AAKD on real shape data. Several factors need to be considered here. First, the shape data are treated as points and the search simulates retrieval of shape data using the query window extension technique. Second, the heuristic employed by AAKD aims to create partitions that are square-like by cracking on the longest axis on mediocre bounds. This combination limits the pruning ability, resulting in more data being scanned per query and, consequently slower search times. If the data were handled as actual shapes then we would expect a performance more similar to AIR.

Figure 17 illustrates the breakdown of the cumulative running time over a random workload of 10k queries on shape data. The running time consists of several components: the start-up cost (which is the time to build the grid), the adaptation time (the time spent extending the index during query processing), the search time (the time spent traversing the tree until the algorithm reaches leaf nodes), and finally, the filtering time (the time taken to filter false positives for CGI+AAKD).

When dealing with uniform data, adaptation time is negligible and due to the uniformity of data, search and filtering times show minimal variation across the different grid-based methods. The experiment on the non-uniform Roads data reveal more interesting insights. Search time is substantially higher for both methods, primarily because these methods utilize the query window extension. This extension results in a much bigger search area compared to the original window query. Consequently, filtering becomes considerably more expensive, again due to the query window extension.

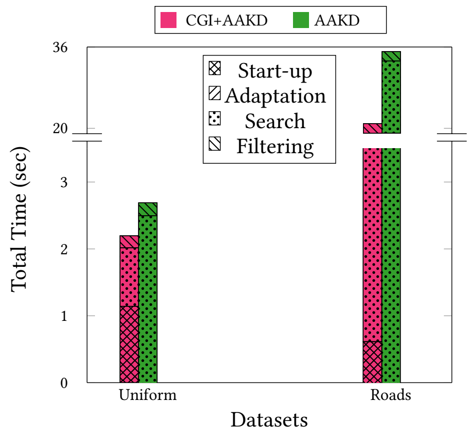


Figure 17: Cumulative time breakdown.

4.3 Effect of object size

AKD-based methods are not tailored handle shapes, and as a result, the extent of a single object in the dataset can significantly impact performance, as queries have to be extended by the largest object extent. Table 2 indicates that our synthetically generated datasets have more uniformly sized shapes, while the real data have shapes of varying size distributions. To further improve our argument that the distribution of the shapes can affect the performance the indexes under study, we vary the sizes in two manners: First we changed the parameters of the widths and heights of the shapes from a uniform distribution with an average of 0.2% of the dimension range, to 0.5%, 1%, and 5% of the dimension range. The details of these new distributions are found in Table 6. And second we synthetically generated a dataset where shapes are uniformly placed in the domain space but have extents that follow an exponential distribution, as described in section 3.1.1. Figure 11 shows the performance of the different indexes under these settings.

Table 6: Extent of datasets with different object sizes.

Extent	AVG	MAX	MIN
Uniform 0.5%	[0.0049, 0.0049]	[0.009, 0.009]	[$4.64e^{-10}$, $1.92e^{-9}$]
Uniform 1%	[0.009, 0.009]	[0.019,0.019]	[$5.62e^{-10}$, $1.31e^{-10}$]
Uniform 5%	[0.045,0.045]	[0.09,0.09]	[$2.05e^{-8}$, $1.39^{e}-9$]

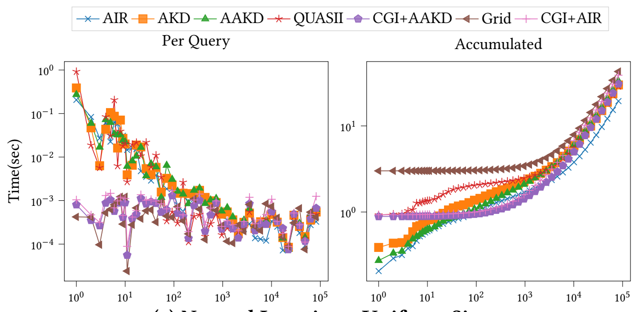
The static grid's index creation cost grows as the average size of the objects grows. This is due to the rising number of objects that are replicated in the grid cells. The performance of AIR remains immune to changes in the size of the data objects. AKD, AAKD, and QUASII get more and more hindered with larger data objects as a consequence of the extensions. An interesting pattern to notice is the behaviour of AAKD when dealing with particularly large objects, as in Figure 11d, and even in the ROADS dataset in Figure 10d. We can see that the AAKD performs worse than than both of its parents, AIR and AKD. This is because trying to fit left corner points of large objects into square-like partitions, actually backfires, and creates a badly shaped index. We notice this in the number of shapes scanned per query, which is the most fundamental measure for index performance. The coarse granular indexes try to resist the unusual sizes and have best performance for objects with sizes up to 1% of the space, but eventually for larger and exponentially sized objects, AIR prevails as the best choice.

4.4 Complex distributions

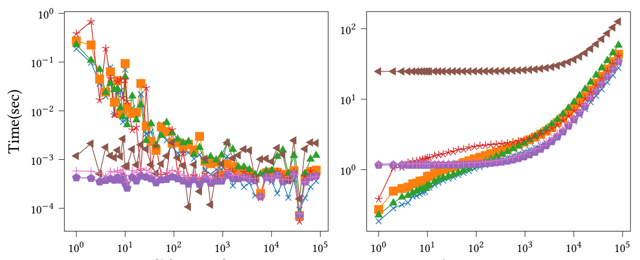
In the next experiment, test various combinations of shape size and location distribution parameters that accommodate more diverse scenarios. First, we synthetically generate shapes that are placed in the domain using a normal distribution, while their size follows a uniform distribution, and vice versa. In addition, we generate a dataset of 10M shape objects where their location and size are correlated, such that shapes become smaller and denser as they move towards the bottom right corner and bigger and sparser as they move towards the upper right corner. Figure 18 shows how the methods under study perform in these new and diverse settings. Similarly to the previous findings, CGI+AAKD performs consistently well in these complex distributions. AIR outperforms CGI+AAKD in Figure 18(c), where object density and size are anti-correlated, which can be attributed to the large objects that appear in the upper right corner, causing large query extensions and extensive post-filtering in CGI+AAKD.

4.5 Effect of object cardinality

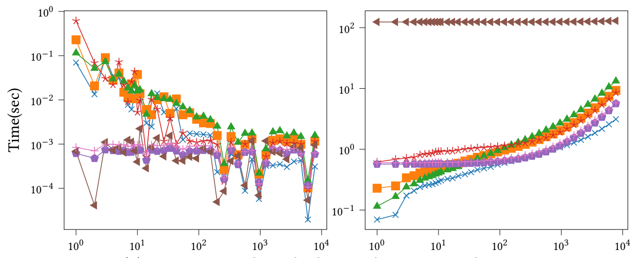
Figure 12 illustrates the scalability performance of the indices across datasets of varying cardinality. In this experiment, we use uniform point datasets of varying size. All other experiments use 20 million objects, in this section and examine the performance of the methods for larger sizes, namely 40, 60, and 80 million objects. Consistent with previous studies [25, 29, 38], the results demonstrate that data



(a) Normal Location - Uniform Size



(b) Uniform Location - Normal Size



(c) Anti-correlated object density and size

Figure 18: Complex distributions

cardinality has little to no impact on the relative performance of the indices under evaluation, highlighting their robustness and scalability. As in other experiments, we see that using a simple **static grid** seems to be the best choice for point datasets. The performance of the **course granular index** in also equivalent.

4.6 Effect of query selectivity

This subsection includes experiments showcasing the effect of query selectivity on point data. Figure 13 shows the per-query and cumulative performance of the indices when faced a workload of varying selectivity. When the selectivity is very high, adaptive indices struggle to create good enough indexes that can filter out future searches. This is evident in the per query times displayed in the 0.1% selectivity that do not decrease down to the static index performance. A similar phenomenon can be seen in too small selectivities, although not as harshly for the 0.0001% selectivity. We have set 0.001% as our default selectivity for all other experiments as a reasonable size. Overall, the **coarse granular index**, or equivalently the **static grid** performs reliably even while affected by the query size.

4.7 Effect of query pattern

Figures 14 and 15 show the performance of the methods in workloads with non-random access patterns for points and shapes respectively. The number of queries are lower for these experiments, as the normalized data space limits the number of sequential non-overlapping queries and meaningful zoom-in queries we can create. As expected, the stochastic crack employed by AIR and AAKD helps mitigate the negative effects of a sequential workload. Note that CGI+AAKD is extremely robust and not affected by the query access pattern when dealing with points.

While a decreasing trend in the per-query times is expected for adaptive indexes, we see a similar trend for static indexes in the zoom-in patterns. This is due to the decrease in the extent and selectivity of queries as we zoom in. As we access fewer grid cells as the workload goes on, the benefits of the CGI with AAKD over the plain AAKD, and CGI with AIR over the plain AIR become moot. As seen in the object size experiments in section 4.3, the performance of AKD, AAKD, CGI with AAKD, and QUASII are affected by the query windows extension and the static grid by its replication. As such, AIR seems to be performing better. However, it is important to remember these workloads are very short therefore the benefit of the coarse grid does not have time to shine.

4.8 Effect of dimensionality

Figure 16 shows results on the effect of dimensionality on point data, performing the same set of experiments on uniform data of higher dimensionality. Most of the indexing methods in this study are designed to support multidimensional data. We have included more indexes in this experiment to make the results come across more clearly. Grids do not scale to higher than 3 dimensions, so they were excluded. The AV-tree still struggles due to the L_{max} distances. The AAKD becomes worse with higher dimensional data, while the plain AKD is less affected. Evidently, the heuristic choice becomes less worthwhile and instead just becomes a burden with higher dimensions. Given the datasets are points, **AKD** and **AIR** have similar reliable behavior regardless of dimensionality as expected.

Table 7: Memory Usage (MB)

	Before 1st query	After workload
AAKD	320	323.16
AIR	320	326.06
AKD	320	322.67
QUASII	320	320.77
Grid	623	623
CGI+AAKD	623	626

4.9 Memory usage

Table 7 shows the memory usage of various indices for 20 million shape data objects uniformly distributed, measured both before any queries are evaluated and after the workload has been completed. Initially, all adaptive indices consume the same amount of memory (accounting for storing the data objects), as no indexing has occurred at that point. In contrast, the grid index allocates discrete memory space for each cell, since the partition is out-of-place. Since the objects are shapes, they may span across multiple cells, resulting in replicated entries across these cells. The grid combined with AAKD (CGI+AAKD) shows that the overhead of the adaptive index in minimal. That is the case for uniform data where the data are equally distributed among the cells and their cardinality is not much bigger than the threshold used.

5 CONCLUSIONS & FINDINGS

We evaluated the performance and robustness of multidimensional adaptive indexes in diverse scenarios, tuning factors such as object size and distribution, workload pattern, and more. Our comparison includes an advanced AKD implementation which adopts cracking heuristics from AIR [38]. We also included a first-time implementation of multidimensional coarse granular indexing (CGI) [32]. We found that grid-based methods perform best on uniformly distributed data and are robust on non-uniform point data, yet their advantage is limited to low-dimensional spaces. Highly clustered collections of real shape data collections pose challenges arising from object sizes and data density in small areas. On such data, the query window extension method impairs KD-based methods, whereas those ingesting data with extent, such as AIR, are more resilient. On shape data, AIR is unaffected by object size, while the hybrid CGI+AAKD method maintains good performance with object size up to 1% of the dimension range. Linearly growing dataset size (i.e., cardinality) has a minor effect on efficiency, towards which static grids and coarse-granular indexes on point datasets are particularly robust. We experimented with up to 10k queries, as we observed that larger workloads did not hamper performance, as the best adaptive indexes eventually match the performance of the best pre-build indices. On the other hand, query selectivity significantly affects adaptive indexing. With high selectivity, adaptive indexes struggle to filter searches effectively. Low selectivity also impacts performance, to a lesser extent. In contrast, static grid methods maintain stable performance regardless of query selectivity. Query patterns also affect performance. Adaptive indexes benefit from zoom-in query patterns that progressively refine search areas. CGI+AAKD remains robust and is largely unaffected by different query access patterns. Dimensionality presents challenges, particularly for grid-based approaches, which do not scale well beyond 3D. AAKD struggles with higher-dimensional data, whereas AIR and AKD deliver reliable performance as dimensionality increases. The AV-tree falters due to its reliance on cumbersome L_{max} distance queries to express ranges.

Our main experimental findings can be summarized as follows:

- On *point* data CGI+AAKD performs best.
- On data of small objects, CGI is most effective; still, for larger and less regular objects, AIR supercedes other methods.
- In space above 3D, AKD and AIR are the best choices.
- On irregular query patterns, no method consistently performs best, yet AIR and AAKD are the most robust.

Our key conclusion is that a static grid index, which had not been considered as a competitor of adaptive indexes in previous studies, is highly effective and robust on point data. Our findings challenge the assumption that adaptive indexes always outperform static ones with a reasonable workload. One of our proposed extensions, the coarse-granular index, is based on this observation, aiming to create a *coarse* grid that gets refined via adaptation throughout the query workload. However, we found the best-performing grid sizes to be equal, thus the simplicity and inexpensive build of the grid is hard for adaptive indexes to overcome on point data. Nonetheless, on shape data, which incur replication in grids, inherently shape-oriented indexes like AIR perform the best, especially so on oddly-sized shape data.

REFERENCES

[1] Victor Alvarez, Felix Martin Schuhknecht, Jens Dittrich, and Stefan Richter. 2014. Main memory adaptive indexing for multi-core systems. In *DaMoN*. 3:1–3:10.

[2] Peter A. Boncz and Martin L. Kersten. 1999. MIL Primitives for Querying a Fragmented World. *VLDB J.* 8, 2 (1999), 101–119.

[3] Boost. 2015. Boost C++ Libraries. http://www.boost.org/. Last accessed 2025-01-10.

[4] Natalia Chaudhry, Muhammad Murtaza Yousaf, and Muhammad Taimoor Khan. 2020. Indexing of real time geospatial data by IoT enabled devices: Opportunities, challenges and design considerations. *Journal of Ambient Intelligence and Smart Environments* 12, 4 (2020), 281–312.

[5] Jialin Ding, Vikram Nathan, Mohammad Alizadeh, and Tim Kraska. 2020. Tsunami: A Learned Multi-dimensional Index for Correlated Data and Skewed Workloads. *Proc. VLDB Endow.* 14, 2 (2020), 74–86.

[6] Quang-Tu Doan, A. S. M. Kayes, Wenny Rahayu, and Kinh Nguyen. 2022. A Framework for IoT Streaming Data Indexing and Query Optimization. *IEEE Sensors Journal* 22, 14 (2022), 14436–14447.

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

[8] Yasmin Fathy, Payam Barnaghi, and Rahim Tafazolli. 2018. Large-Scale Indexing, Discovery, and Ranking for the Internet of Things (IoT). *ACM Comput. Surv.* 51, 2 (2018).

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

[10] Goetz Graefe, Felix Halim, Stratos Idreos, Harumi Kuno, and Stefan Manegold. 2012. Concurrency control for adaptive indexing. *Proc. VLDB Endow.* 5, 7 (2012), 656–667.

[11] Goetz Graefe, Felix Halim, Stratos Idreos, Harumi Kuno, Stefan Manegold, and Bernhard Seeger. 2014. Transactional support for adaptive indexing. *The VLDB Journal* 23 (04 2014), 303–328.

[12] Goetz Graefe and Harumi A. Kuno. 2010. Self-selecting, self-tuning, incrementally optimized indexes. In *EDBT*. 371–381.

[13] Antonin Guttman. 1984. R-trees: a dynamic index structure for spatial searching. In *SIGMOD*. 47–57.

[14] Felix Halim, Stratos Idreos, Panagiotis Karras, and Roland H. C. Yap. 2012. Stochastic Database Cracking: Towards Robust Adaptive Indexing in Main-Memory Column-Stores. *Proc. VLDB Endow.* 5, 6 (2012), 502–513.

[15] Pedro Holanda and Stefan Manegold. 2021. Progressive Mergesort: Merging Batches of Appends into Progressive Indexes. In *EDBT*. 481–486.

[16] Pedro Holanda, Matheus Nerone, Eduardo C. de Almeida, and Stefan Manegold. 2018. Cracking KD-Tree: The First Multidimensional Adaptive Indexing (Position Paper). In *DATA*. 393–399.

[17] Pedro Holanda, Mark Raasveldt, Stefan Manegold, and Hannes Mühleisen. 2019. Progressive indexes: indexing for interactive data analysis. *Proc. VLDB Endow.* 12, 13 (2019), 2366–2378.

[18] Stratos Idreos. 2010. Database Cracking: Towards Auto-tuning Database Kernels. *CWI, PhD Thesis* (2010).

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

[20] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. 2007. Updating a cracked database. In *SIGMOD*. 413–424.

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

[22] 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. *Proc. VLDB Endow.* 4, 9 (2011), 585–597.

[23] Anders Hammershøj Jensen, Frederik Lauridsen, Fatemeh Zardbani, Stratos Idreos, and Panagiotis Karras. 2021. Revisiting Multidimensional Adaptive Indexing [Experiment & Analysis]. In *EDBT*. 469–474.

[24] Martin L. Kersten and Stefan Manegold. 2005. Cracking the Database Store. In *CIDR*. 213–224.

[25] Konstantinos Lampropoulos, Fatemeh Zardbani, Nikos Mamoulis, and Panagiotis Karras. 2023. Adaptive Indexing in High-Dimensional Metric Spaces. *Proc. VLDB Endow.* 16, 10 (2023), 2525–2537.

[26] Mingliu Liu, Deshi Li, Qimei Chen, Jixuan Zhou, Kaitao Meng, and Song Zhang. 2018. Sensor Information Retrieval From Internet of Things: Representation and Indexing. *IEEE Access* 6 (2018), 36509–36521.

[27] Nikos Mamoulis. 2011. *Spatial Data Management.* Morgan & Claypool Publishers.

[28] Vikram Nathan, Jialin Ding, Mohammad Alizadeh, and Tim Kraska. 2020. Learning Multi-Dimensional Indexes. In *SIGMOD*. 985–1000.

[29] Matheus Agio Nerone, Pedro Holanda, Eduardo C. de Almeida, and Stefan Manegold. 2021. Multidimensional Adaptive & Progressive Indexes. In *ICDE*. 624–635.

[30] Mirjana Pavlovic, Darius Sidlauskas, Thomas Heinis, and Anastasia Ailamaki. 2018. QUASII: QUery-Aware Spatial Incremental Index. In *EDBT*. 325–336.

[31] Holger Pirk, Eleni Petraki, Stratos Idreos, Stefan Manegold, and Martin Kersten. 2014. Database cracking: fancy scan, not poor man's sort!. In *DaMoN*. 4:1–4:8.

[32] Felix Martin Schuhknecht, Alekh Jindal, and Jens Dittrich. 2013. The uncracked pieces in database cracking. *Proc. VLDB Endow.* 7, 2 (2013), 97–108.

[33] Felix Martin Schuhknecht, Alekh Jindal, and Jens Dittrich. 2016. An experimental evaluation and analysis of database cracking. *VLDB J.* 25, 1 (2016), 27–52.

[34] 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. *Int. J. Geogr. Inf. Sci.* 11, 6 (1997), 529–554.

[35] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alex Rasin, Nga Tran, and Stanley B. Zdonik. 2005. C-Store: A Column-oriented DBMS. In *VLDB*. 553–564.

[36] Sivaprasad Sudhir, Michael J. Cafarella, and Samuel Madden. 2021. Replicated Layout for In-Memory Database Systems. *Proc. VLDB Endow.* 15, 4 (2021), 984–997.

[37] Dimitrios Tsitsigkos, Konstantinos Lampropoulos, Panagiotis Bouros, Nikos Mamoulis, and Manolis Terrovitis. 2021. A Two-layer Partitioning for Non-point Spatial Data. In *ICDE*. 1787–1798.

[38] Fatemeh Zardbani, Nikos Mamoulis, Stratos Idreos, and Panagiotis Karras. 2023. Adaptive Indexing of Objects with Spatial Extent. *Proc. VLDB Endow.* 16, 9 (2023), 2248–2260.